Homology of moving points

Sweep, R.

Award date:
2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Geometric Algorithms Research Group

Homology of Moving Points

Master Thesis

Rik Sweep

Supervisors:
Kevin Buchin
Kevin Verbeek
Jesper Nederlof

Eindhoven, Wednesday 16th October, 2019

Abstract

Topological data analysis is an area of research that is concerned with the shape of data. It has applications in for instance machine learning, biology and geographic data analysis. Homology tries to capture the shape of data by computing and characterizing the holes in the data. In this thesis, we study the shape of moving data. We present a data structure that captures the homology of the points once, and maintains it throughout a simulation of the points by updating the structure whenever it should change. In this thesis we present a kinetic data structure for computing and maintaining the homology of a set S of n moving balls of fixed radius in $\mathbb{R}^d$.

The kinetic data structure for maintaining the homology of this set of moving balls consists of two parts. The first part is a kinetic data structure to maintain the d-dimensional α-complex, which is a shape equivalent to the union of balls. The second part is a dynamic data structure that computes the homology of the α-complex. For the dynamic data structure we maintain the boundary matrices and we introduce the notion of homology matrices, which we also maintain. We handle $\mathcal{O}\left(n^d \lambda_\delta(n)\right)$ events, where $\lambda_\delta(n)$ denotes the maximum length of a (n, δ)-Davenport-Schinzel sequence and $\delta \geq 0$ is a parameter that describes the complexity of the movement of the points. Each event is handled in $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time, where n_d denotes the complexity of the α-complex, β denotes the maximum number of holes and a is a small constant integer. We need $\mathcal{O}\left(n_d \beta\right)$ space to store the data structure and there are $\mathcal{O}\left(n_d\right)$ certificates associated with a single vertex.

We present another data structure with corresponding algorithms for speeding up the maintenance of parts of the homology. We improve the dynamic data structure for maintaining the homology by not storing and maintaining the lowest and highest dimensional boundary matrix and the lowest and highest dimensional homology matrix. We instead use that the 0-homology group is equivalent to the connected components of the α-complex and that the $(d-1)$-homology group is equivalent to the connected components of the dual of the α-complex. We also improve the kinetic data structure for maintaining the α-complex by ignoring points in the center of clusters and the events associated with these points. In this way, the data structures need fewer events, handle events faster and need less space to be stored.

Contents

Chapter 1

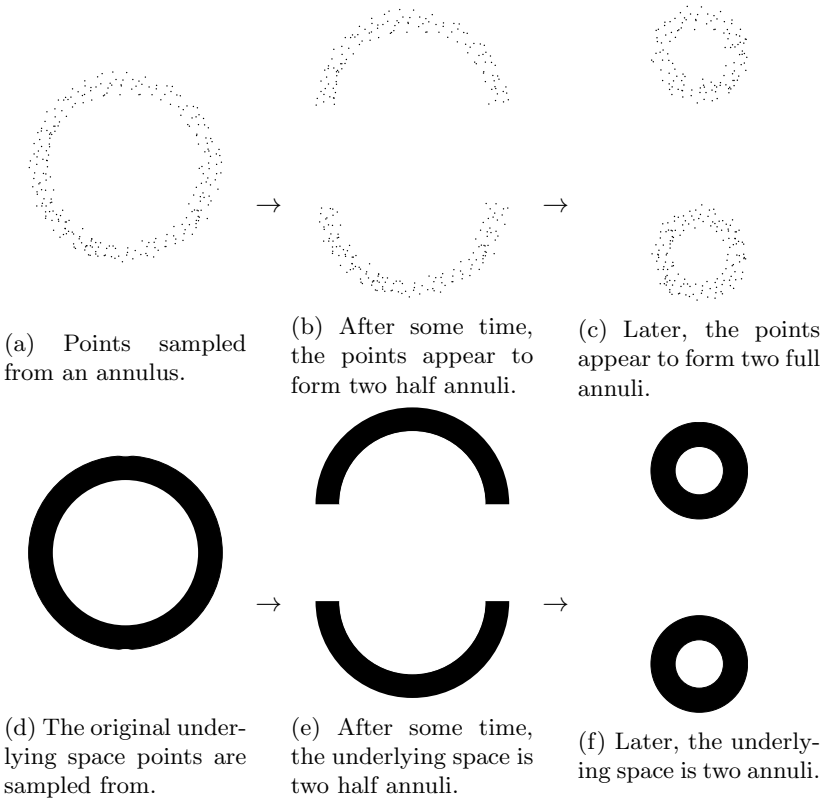
Introduction

Motivation Topological data analysis is a fast growing area of research with many applications in several areas of research [22]. Topological data analysis is a means to capture the 'shape' of data. More precisely, it assumes that there is sampling data from some underlying topological space X and it tries to approximate the shape of X from this data. An example is the data from Figure 1.1a, which is sampled from the underlying annulus, depicted in Figure 1.1d. The goal of topological data analysis in the case of Figure 1.1 is to find that X is an annulus while we are only given the data from Figure 1.1a.

In this thesis, we are interested in data that moves. We are given a set of points where each point moves along a trajectory and has a flight plan that prescribes where it will move next. During such motions, topological structures that are defined by the location of the points may change. For example, the convex hull of the points changes shape and position as the points move. At some moments in time, a new point may even appear on the convex hull of the data. One can imagine that the shape of the data also changes as the data moves. In this thesis, we try to capture the shape of data that moves and to track the changes in its shape over time. An example of such a situation can be seen in Figure 1.1. In Figure 1.1a, the shape of our data is the annulus of Figure 1.1d. However, if the top half of the data points move upwards, and the bottom half of the points move downwards, we obtain the configuration of Figure 1.1b. The shape of this configuration is no longer captured by the annulus, but rather by two half annuli, as shown in Figure 1.1e. The points may continue to move even further, to acquire the position shown in Figure 1.1c. The shape of this data is best captured using two annuli, As shown in Figure 1.1f.

Background An important part of topological data analysis is *homology*. Homology is a way to capture the shape of data by looking at 'holes' in the data. These holes exist in multiple dimensions. A k-dimensional hole is essentially the absence of a k-dimensional object so we say a k-dimensional hole can be *filled* by adding that k-dimensional object. For example, the hole in the annulus in Figure 1.1d can be filled using a 2-dimensional surface; in this case, a disk. A 1-dimensional hole is the gap between two connected components. It can be filled by connecting the two components with a line segment, which is indeed a 1-dimensional object. This generalizes into higher dimensions. For each dimension k, the k^{th} *Betti number* is defined as the number of independent holes. The objective of homology in the setting of topological data analysis is to find the holes in a space X, while we are only given data sampled from X. The data from Figure 1.1a, for example, does not form a 2-dimensional hole as they are, since they are just points. One way to solve this is by choosing a radius α and drawing a ball of radius α around each data point. For a suitable choice of α, the union of these balls tends to have the same number of holes as X as we can see when we compare Figure 1.1d to Figure 1.2. To determine the value of the radius α one may use the theory of *persistent homology*. However, this is beyond the scope of this thesis and we assume such a radius is readily available.

Much work exists on the computational aspects of homology. The classical algorithms related to homology can be found in the book by Munkres. [33]. An introduction to the topic can be found



(a) Points sampled from an annulus.

(b) After some time, the points appear to form two half annuli.

(c) Later, the points appear to form two full annuli.

(d) The original underlying space points are sampled from.

(e) After some time, the underlying space is two half annuli.

(f) Later, the underlying space is two annuli.

Figure 1.1: The points of some underlying space move along with the underlying space.

in [19], where many aspects of homology are discussed including some intuition, and algorithms to compute homology. Over the years, topological data analysis has expanded to also include co-homology, persistent homology, zig-zag persistent homology and many more. For an overview of these topics, we refer the reader to [29]. Also, several different techniques exist for computing homology. The classical way to compute homology comes down to computing the *Smith-normal form* of a matrix [19]. There are several ways to compute the Smith-normal form of a matrix [16]. Other ways to compute homology is with the use of Laplacians [20], using reduction of chain complexes [26] or using reduction of cohomology [15]. However, to the best of our knowledge, none of the techniques provably obtain better running time bounds than the classical algorithm.

To study situations where the data moves over time in a live fashion, the framework of *kinetic data structures* was developed [5]. By moving in a live fashion, we mean that we know a *flight plan* for each of the vertices but these flight plans may change at any point in time. A flight plan is a continuous trajectory which the point is going to follow. Also, the history of the property of interest is not stored, and we only know what the situation is at this point in time. This is different from investigating a movie, where the full path of the points is known beforehand and we are probably interested in the full history of the property of interest. We call the time that we are monitoring the movement of the data the *simulation*. At the start of the simulation, a kinetic data structure computes the information we would like to know and maintain and makes sure this information stays correct at all times. This means that when the points start moving, the information may become incorrect due to the updated position of the points and the data structure will update its information accordingly. We explicitly compute when the changes happen, so that we can adjust the information at the same time the change happens. This is different from computing the information from scratch repeatedly, as might be the case when we compute the information at certain time frames. The advantage of this method is that it is independent of a choice of the time frames. The desired information is known at all times, and not just at specific moments in time.

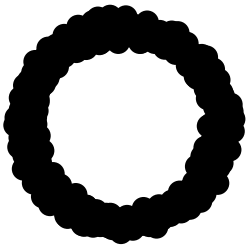


Figure 1.2: The union of disks around each data point of Figure 1.1a.

Also, we are certain that only one change happens at each computation, so we may compute the new information using the old information whereas in between two time-frames, multiple changes may have occurred, and we cannot be certain any of the old information is still relevant.

The framework of kinetic data structures was developed by Basch et al. [5]. It provides a new perspective on the design and analysis of data structures for moving points which caused this area of research to grow rapidly. Already one year after its introduction, kinetic data structures existed for maintaining the kinetic convex hull [5], maintaining the closest pair of a set of moving points [6], maintaining a kinetic $(1 + \varepsilon)$-minimum spanning tree [6], maintaining the diameter, width and smallest area or perimeter bounding rectangle of a set of moving points [2] and maintaining a kinetic minimum spanning tree [1]. Surveys on kinetic data structures can be found in several books, such as Chapter 23 in the first edition of the Handbook of Data Structures and Applications [31] and Chapter 24 in its second edition [32]. Also in the Handbook of Discrete and Computational Geometry (second edition) [21] Chapter 50 discusses kinetic data structures and Chapter 53 in the third edition [22].

Our Contributions We consider a set S of n points $x_1, \ldots, x_n$, moving in $\mathbb{R}^d$ and we assume we are given a radius $\alpha > 0$. The goal of this thesis is to present a kinetic data structure that maintains the homology and Betti numbers for the union of balls of radius α around the points in S.

Some work already exists that considers the homology of moving points. An example of this is the trajectory grouping structure [11], which considers a set of moving points and computes when they form so-called *groups*. A set of moving points is called a group if there are enough points in this set and they are close enough during the simulation for a duration that is long enough. These aspects are formalized using the trajectory grouping structure. The aspect of closeness makes this a special case of keeping track of part of the homology, namely the connected components. The lifespan of a group is related to persistent homology, since persistent homology concerns the lifespan of holes. The trajectory grouping structure is restricted in the sense that it only considers 2-dimensional trajectories and connected components, which is only a part of homology. Other work on kinetic homology studies a kinetic data structure to maintain the connected components of a set of unit disks [23]. Though this is a part of homology, this work restricts itself in the sense that it does not consider holes of any other type. It also restricts itself in the sense that it only considers 2-dimensional moving points. More closely related to this thesis is previous work on the kinetic α-complex [27].

The α-complex is a shape that is equivalent to the union of balls but has lower complexity. However, previous work on the kinetic α-complex is restricted to three dimensions and does not consider the question of also maintaining the homology of the shape. In our work, however, we are primarily interested in kinetically maintaining the homology, and we want to do so for points in arbitrary dimension.

In this thesis we present algorithms for maintaining all parts of the homology of the union of balls around a set of moving points in d dimensions. For this, we maintain the α-complex which is a shape that is equivalent to the union of balls. We adapt the classical algorithm for computing homology to a dynamical setting, which we then use to maintain the homology of a topologically

changing shape over time.

After covering the underlying notions in Section 2, we present in Section 3 algorithms to kinetically maintain the α-complex. To only maintain the α-complex, we need $\mathcal{O}\left(n^d \lambda_\delta(n)\right)$ changes where n is the number of vertices, d is the dimension of the points, δ is a parameter describing the flight plans of the vertices and $\lambda_\delta(n)$ is the maximum length of a (n, δ)-Davenport-Schinzel sequence. Whether this bound on the number of changes is sharp, is unknown. However, we do know that the number of changes processed by the algorithm is optimal in the worst case. Each change can be processed in $\mathcal{O}(\rho \log n)$ time and we need $\mathcal{O}(n_d)$ space to store the structure, where ρ is the maximum degree of a vertex in S and n_d is the complexity of the α-complex.

In Section 4 we present the algorithms that maintain the homology of a shape. We will show that we process the same number of changes as for the α-complex, that each change is processed in $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time and that we need $\mathcal{O}(n_d \beta)$ space to store the structures, where a is a small constant integer and β denotes the maximum number of holes. In Section 5 we present improvements for maintaining certain parts of the homology. Some parts of the homology can be maintained more efficiently than using the algorithms mentioned above. We will show that the 0-dimensional gaps are essentially the connected components of a graph and that they can maintained using $\mathcal{O}\left(\log^2 n\right)$ amortized time per change and $\mathcal{O}\left(n^2\right)$ storage space. We will also show that the $(d-1)$-dimensional gaps can be maintained more easily. Using a structure dual to the original shape, the maintenance of the $(d-1)$-dimensional gaps is reduced to maintaining the connected components of a graph which can be done in $\mathcal{O}\left(\log^2 n\right)$ amortized time per change and $\mathcal{O}(n_d)$ storage space. In Section 6 we present the kernel optimization technique, which may be used to eliminate some of the vertices, thus possibly improving the efficiency of maintaining the homology. This results in additional changes for each vertex because we maintain more information about the position and surroundings of each vertex. However, we show that these changes can be processed in $\mathcal{O}(\log n)$ time. The benefit of the kernel optimization technique is that under some conditions, several moving points may be replaced by one or more static points. This may save us considerable time if many points are replaced.

Chapter 2

Preliminaries

The preliminaries are divided into three topics. The first topic explains some of the basic principles of simplicial complexes, the second deals with homology and the third deals with kinetic data structures. These elements are the building blocks for the rest of the thesis.

2.1 Simplicial Complexes

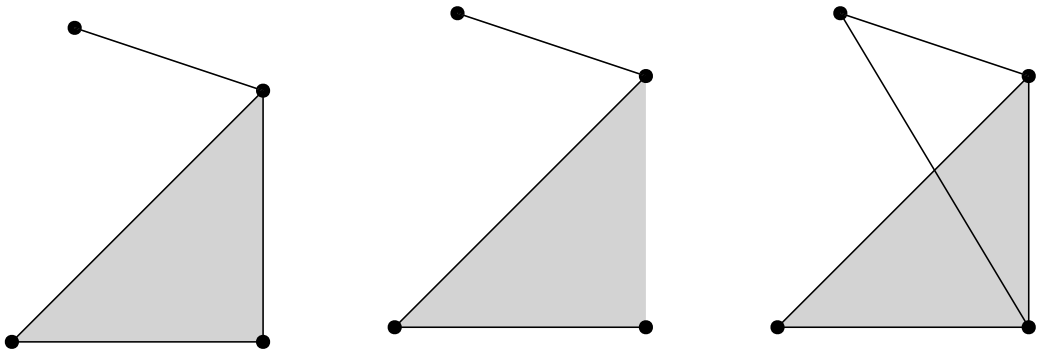
Consider a set S of n points $x_1, \ldots, x_n$ in $\mathbb{R}^d$. A finite collection of points is called *affinely independent* if no affine space of dimension k contains more than $k+1$ of the points, and this is true for every k. A *k-simplex* σ is the convex hull of a set of $k+1$ affinely independent points and k is called the *dimension* of σ. This means that a 1-simplex is a line segment, a 2-simplex is a triangle and a 3-simplex is a tetrahedron. We call σ a *simplex* if its dimension is unimportant. In $\mathbb{R}^d$, the largest number of affinely independent points is $d+1$, so we have simplices of dimension $-1, 0, \ldots, d$ where the (-1)-simplex is the empty set [18]. The *affine hull* of a set of points (or, equivalently, of a simplex) is the affine space with smallest dimension containing the set of points (or simplex). Let σ be a k-simplex consisting of $k+1$ points $x_0, \ldots, x_k$. We write $\sigma = (x_0, \ldots, x_k)$. Let $s = \{x_0, \ldots, x_k\}$ be the set of vertices that span σ. The simplex τ formed by taking the convex hull of a subset s' of s is called a *face* of σ. We call τ an *l-face* of σ if it is a face of σ and τ is an l-simplex. A *proper face* τ of σ is an l-face with $0 \leq l < k$. We call simplex ρ a *coface* of σ if σ is a face of ρ. Similarly to faces, ρ is called an *l-coface* of σ if it is a coface of σ and ρ is an l-simplex. Also, we call ρ a *proper coface* of σ if it is an l-coface with $l > k$. A face or coface τ of σ is called *improper* if $\tau = \emptyset$ or $\tau = \sigma$.

A *simplicial complex* C is a set of simplices with the following properties.

1. If $\sigma \in C$ and τ is a face of σ then $\tau \in C$.
2. If $\sigma, \tau \in C$, then $\rho = \sigma \cap \tau$ is a face of σ and of τ.

So all faces of a simplex are present in the complex and two simplices in the complex only intersect in a common face (note that the empty set may be this common face if two simplices do not intersect). An example of a valid simplicial complex can be found in Figure 2.1a. An example of a set of simplices where the first property is violated can be found in Figure 2.1b and an example of a set of simplices where the second property is violated can be found in Figure 2.1c. A *subcomplex* of a simplicial complex is a subset of the simplices that is also a simplicial complex. An example of a subcomplex is the *l-skeleton*. The l-skeleton of a simplicial complex C is the set of k-simplices in C with $k \leq l$.

If we replace each simplex $\sigma = (x_0, \ldots, x_k)$ by the set $s = \{x_0, \ldots, x_k\}$, a simplicial complex can be represented as a system of subsets of the point set S. If we also disregard the position of the point set, such a set C' of subsets is called an *abstract simplicial complex*. The two properties of a simplicial complex are replaced by the property that if $s \in C'$ and $t \subseteq s$ then $t \in C'$. Each simplicial complex corresponds to an abstract simplicial complex which you can construct by



(a) A valid simplicial complex.

(b) A simplicial complex that violates the first property.

(c) A simplicial complex that violates the second property.

Figure 2.1: A valid simplicial complex and two invalid complexes.

storing the simplices as sets in the abstract simplicial complex. Once the vertices in an abstract simplicial complex have coordinates in some space, a simplicial complex is constructed (assuming the vertices are placed such that the second property of simplicial complexes holds). Such a simplicial complex is called a *geometric realization* of C' [18].

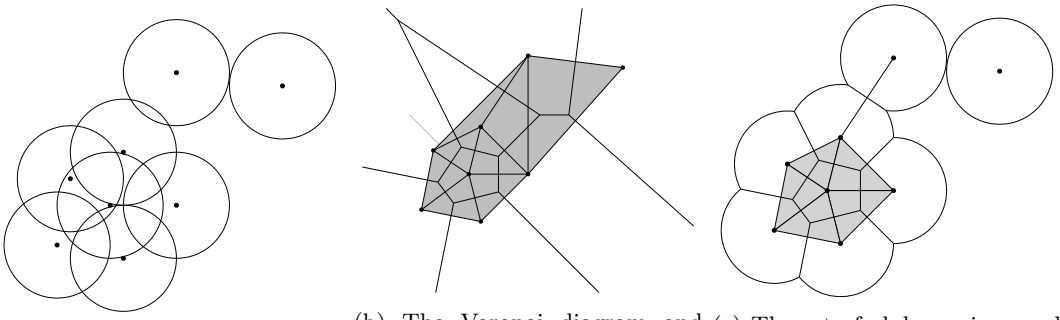
Let C be a collection of sets U_i. The *nerve* of C is defined as the collection of index sets J such that $\bigcap_{j\in J} U_j \neq \emptyset$ [4]. It is easy to see that the nerve of a collection of sets is always an abstract simplicial complex. A well-known example is the *Delaunay complex*, which is the nerve of the *Voronoi diagram*. The Voronoi diagram of a set of points S is a partitioning of the space in so-called *Voronoi cells*. A Voronoi cell V_i of a point $x_i \in S$ is the set of points for which x_i is the closest point from S, that is, $V_i = \{x \in \mathbb{R}^d | \, \|x - x_i\| \leq \|x - x_j\| \, \forall x_j \in S \setminus \{x_i\}\}$ for all $i \in \{1, \ldots, n\}$. So, essentially, in a Voronoi diagram, each point x in the space is assigned to the point in S which is closest to x. An example of a Voronoi diagram can be found in Figure 2.2b. The Delaunay complex is indicated using gray triangles. The *α-ball* $B_i(\alpha)$ around a point $x_i \in S$ is the set of points with distance to x_i less than or equal to α. So $B_i(\alpha) = \{x \in \mathbb{R}^d | \, \|x - x_i\| \leq \alpha, x_i \in S\}$. An example of a set of α-balls can be found in Figure 2.2a. The nerve of a set of α-balls around the points of S is called a *Čech complex*. The *α-region* $R_i(\alpha)$ around a point $x_i \in S$ is defined as the intersection between V_i and $B_i(\alpha)$. The nerve of the set of α-regions around the points of S is called the *α-complex* of S. We will denote the α-complex of a point set S by $K_\alpha(S)$. An example of a set of α-regions with its corresponding α-complex can be found in Figure 2.2c. If the point set is clear from the context, we will also write $K_\alpha = K_\alpha(S)$. Since each α-region is a subset of the corresponding Voronoi cell, the α-complex is a subcomplex of the Delaunay complex. The α-complex will be used throughout the thesis. Its use is made clear in the following lemma.

Lemma 1. *The set of α-regions and the union of balls of radius α contain the same points.*

Proof. [18] We show that the set of α-regions and the union of balls of radius α cover each other. Since $R_i(\alpha) = V_i \cap B_i(\alpha)$, we know that $R_i(\alpha)$ is covered by $B_i(\alpha)$ for each x_i, so the the set of α-regions is covered by the union of balls.

To prove the reverse, let x be a point in the union of balls. Let $x_i \in S$ be the closest point to x of all points in S. Since x is in the union of balls, we know there exists a $x_j \in S$ such that $\|x - x_j\| \leq \alpha$. Now, $\|x - x_i\| \leq \|x - x_j\| \leq \alpha$ so x is in x_i's Voronoi region V_i and in x_i's α-ball $B_i(\alpha)$. Therefore, it is in x_i's α-region. Since we did not make any assumptions on x, we know that the union of balls is covered by the set of α-regions. □

The *circumsphere* of a k-simplex is the $(k-1)$-sphere going through the $k+1$ vertices that span σ. Given α, a simplex is called *short* if the radius of its circumsphere is smaller than, or equal to α. The ball of maximal dimension d with the same radius and center as the circumsphere of a simplex is called the *circumball* of that simplex. Note that the circumsphere of a d-simplex then becomes the boundary of the circumball of that simplex. As an example, a triangle embedded in



(a) The set of α-balls around S. (b) The Voronoi diagram and Delaunay complex of S. (c) The set of alpha regions and the α-complex around S.

Figure 2.2: A point set S with the union of balls, Voronoi diagram, Delaunay complex, set of α-regions and α-complex.

$\mathbb{R}^3$ has a circle going through the three cornerpoints of the triangle as its circumsphere and its circumball is a 3-dimensional ball with the same radius and center as its circumsphere. We call a simplex *Gabriel* if its circumball contains no points in its interior. We now have the following result.

Lemma 2. *A simplex σ from the Delaunay complex is included in the alpha complex with parameter α if and only if it is short and Gabriel or it has a coface that is short and Gabriel.*

The proof for this is easily obtained from generalizing Lemma 1 in [27].

2.2 Homology

Consider a simplicial complex C. A *k-chain* c is a formal sum of k-simplices in C. We write $c = \sum a_i\sigma_i$ where $a_i \in \mathbb{Z}/2\mathbb{Z}$ are *coefficients* and σ_i are k-simplices. For example, Figure 2.3 shows a simplicial complex and any set of edges forms a 1-chain of this complex. In our situation, the coefficients are computed modulo 2. Essentially, a simplex σ_i is part of the chain ($a_i = 1$) or it is not ($a_i = 0$). Addition of k-chains happens componentwise. This means that if $c = \sum a_i\sigma_i$ and $c' = \sum b_i\sigma_i$, then $c + c' = \sum(a_i + b_i)\sigma_i$. Since the a_i and b_i are computed modulo 2, they are essentially bits, and addition can be seen as the XOR operation '$\otimes$'. The *k^{th} chain group*, $(C_k, +)$, is the group formed by the k-chains together with the addition operation. We will write $C_k = (C_k, +)$ when we mean addition modulo 2. The addition operation is associative because addition modulo 2 is associative. The neutral element of the chain group is $0 = \sum 0\sigma_i$. The inverse of a k-chain c is c itself since $c + c = \sum(a_i + a_i)\sigma_i = \sum(a_i \otimes a_i)\sigma_i = 0$. We find that C_k is indeed a group. The group is Abelian because addition modulo 2 is Abelian [19].

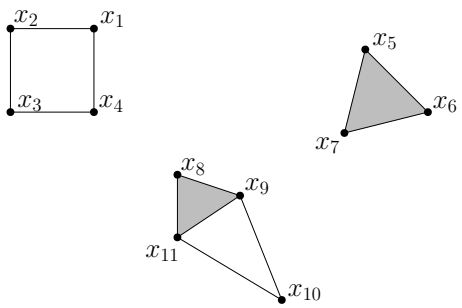


Figure 2.3: A simplicial complex consisting of three connected components and two cycles.

To relate the different chain groups, we introduce the *boundary* of a k-simplex as the sum of its $(k-1)$-faces. In the example of Figure 2.3, the boundary of triangle (x_5, x_6, x_7) is the set of edges (x_5, x_6), (x_6, x_7) and (x_5, x_7). Now, the k^{th} *boundary map* $\partial_k : C_k \to C_{k-1}$ maps a k-chain to its boundary. Following our previous example, we get $\partial_2(x_5, x_6, x_7) = (x_5, x_6) + (x_6, x_7) + (x_5, x_7)$. The boundary of a k-chain is defined as the sum of the boundaries of the simplices in the chain. Note that $\partial_k(c + c') = \partial_k(c) + \partial_k(c')$. A k-chain c is called a *k-cycle* if $\partial_k(c) = 0$. For example, the three edges (x_9, x_{10}), (x_9, x_{11}) and (x_{10}, x_{11}) from Figure 2.3 form a 1-cycle because $\partial_1((x_9, x_{10}) + (x_9, x_{11}) + (x_{10}, x_{11})) = x_9 + x_9 + x_{10} + x_{10} + x_{11} + x_{11} = 0$. The *$k^{th}$ cycle group*, $Z_k = (Z_k, +)$, is a subgroup of the k-chains and consists of those k-chains that get mapped to zero, i.e. $Z_k = \ker(\partial_k)$. A k-chain c is called a *k-boundary* if ∂_{k+1} maps some $(k+1)$-chain to c. That is, $\exists d \in C_{k+1}$ such that $\partial_{k+1}(d) = c$. For example, the 1-chain (x_5, x_6), (x_5, x_7) and (x_6, x_7) is a 1-boundary because $\partial_2((x_5, x_6, x_7)) = (x_5, x_6) + (x_5, x_7) + (x_6, x_7)$. Similar to the k^{th} cycle group, the *k^{th} boundary group*, $B_k = (B_k, +)$, is a subgroup of the k-chains and it consists of those k-chains c for which there exists a $(k+1)$-chain d such that $\partial_{k+1}(d) = c$, i.e. $B_k = \text{Im}(\partial_{k+1})$. As it happens, every k-boundary is necessarily a k-cycle or, equivalently, B_k is a subgroup of Z_k. This property is captured in the following lemma.

Lemma 3 (Fundamental Lemma of Homology)**.** *$\partial_k(\partial_{k+1}(d)) = 0$ for every integer k and every $(k+1)$-chain d.*

Proof. [19] We only need to show that $\partial_k(\partial_{k+1}(\tau)) = 0$ for a $(k+1)$-simplex τ. The boundary, $\partial_{k+1}(\tau)$, consists of all k-faces of τ. Every $(k-1)$-face of τ belongs to exactly two k-faces, so $\partial_k(\partial_{k+1}(\tau)) = 0$. □

Two k-cycles c and c' are said to be *homologous* if $c = c' + d$ where $d \in B_k$. For example, in Figure 2.3, the 1-cycles $((x_9, x_{10}), (x_{10}, x_{11}), (x_9, x_{11}))$ and $((x_9, x_{10}), (x_{10}, x_{11}), (x_8, x_{11}), (x_8, x_9))$ are homologous because $((x_9, x_{10}) + (x_{10}, x_{11}) + (x_8, x_{11}) + (x_8, x_9)) = ((x_9, x_{10}) + (x_{10}, x_{11}) + (x_9, x_{11})) + ((x_8, x_9) + (x_8, x_{11}) + (x_9, x_{11}))$. The homologous relation is an equivalence relation, which means we can partition the k-cycles into *homology classes*, which are the sets of homologous cycles. Now, the *k^{th} homology group* H_k is defined as the k^{th} cycle group modulo the k^{th} boundary group, $H_k = Z_k / B_k$. The *k^{th} Betti number* β_k is defined as the rank of the k^{th} homology group. We say we *know* the homology of a simplicial complex if we know a *representative k-cycle* for each homology class in H_k. A representative k-cycle is a k-cycle from its homology class.

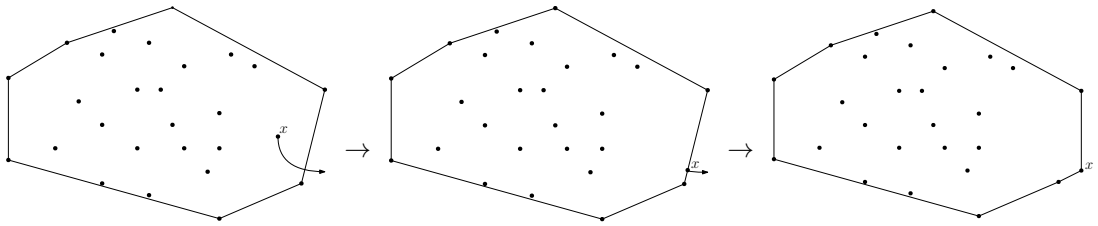
Two simplicial complexes C and C' are said to be *homotopy equivalent* if there exist two continuous maps $g : C \to C'$ and $g' : C' \to C$ such that $g' \circ g = \text{id}_C$ and $g \circ g' = \text{id}_{C'}$. An example is the set of α-regions and the union of balls. Because these two sets cover each other, they are homotopy equivalent and their homology groups are the same. The following theorem gives us more information about the homology of sets [19].

Theorem 1 (nerve theorem)**.** *Let C be a collection of closed, convex sets U_i. Then C is homotopy equivalent to its nerve: the collection of sets J such that $\bigcap_{j \in J} U_j \neq \emptyset$.*

In particular, This means that the union of balls is homotopy equivalent to its nerve: the Čech complex. Also, the set of α-regions is homotopy equivalent to the α-complex. Because homotopy equivalence is an equivalence relation, the union of balls is homotopy equivalent to the α-complex. We will use this fact later in the thesis.

2.3 Kinetic data structures

We now consider the case where the points $x_i \in S$ are moving. We assume the movement of these points is modeled as polynomials in time t, so $x_i(t) = (f_i^1(t), f_i^2(t), \ldots, f_i^d(t))$ where the $f_i^j(t)$ are polynomials of degree δ. These polynomials are called *coordinate functions*. Such a set of coordinate functions determine a flight plan for the point. At certain times, the motion of x_i may be updated. What happens is that one or more of the coordinate functions f_i^j of point x_i are changed into a new function $\widehat{f}_i^j$. Each coordinate function can be changed into a new coordinate



(a) Convex hull of a point set where x moves.

(b) Convex hull at the moment an event occurs.

(c) Convex hull after the movement of x.

Figure 2.4: An example of an exterior event for a kinetic data structure maintaining the convex hull on a point set.

function at any point in time. However, because the coordinate functions are part of the input, so are these changes and we have no influence on them.

To handle a situation in which the location of vertices is known as a polynomial function in time, the framework of *kinetic data structures* was developed [5]. A kinetic data structure maintains a (usually topological) structure on the input data and it uses *certificates* to certify the correctness of the current state of that structure. In Figure 2.4 we see how a kinetic data structure that maintains the convex hull of a point set evolves over time. A certificate can be seen as a predicate on the input that holds true for some period of time. All certificates together prove that the current state of the data structure is indeed correct. A certificate is said to *fail* after the period of time in which it holds true. The time at which a certificate fails is known as the *failure time* of that certificate. At any failure time we say that an *event* occurs. For example, in Figure 2.4 we see that the convex hull first does not include point x. Then, as time progresses, x reaches the hull at Figure 2.4b and an event occurs. After this event, the convex hull does include x. If the structure that is maintained is topological, an event is also known as a *topological event*. All events are stored in an *event queue* Q where they are sorted by their corresponding failure times. This event queue is implemented as a priority queue where the event that will happen soonest is popped from the queue when time has reached the failure time of that event. To compute the failure times of the certificates, each certificate has an *event function*. The certificate fails when the event function reaches zero. Typically, the event functions have the coordinates of vertices as their variables, and, since these coordinates are polynomials in time, the times at which the event function reaches zero can be computed.

The quality of a kinetic data structure is assessed differently than the quality of a normal data structure. For a normal data structure, we are interested in the running time of a query, the space needed to store the data structure, and possibly the construction time. For dynamic data structures, we are additionally interested in the update time, e.g., the time it takes to insert or delete a point. For kinetic data structures, we check if the data structure does not handle too many events, if an event is handled quickly, if the data structure does not take too much space on disk and if not too many certificates may fail at the same time [5]. We now formalize these principles. An event is called an *internal event* if the property we would like to maintain does not change during the event. Similarly, an event is called an *external event* if the property we would like to maintain does change and the event is caused by this change. The event in Figure 2.4b is an external event because the topological structure of the convex hull changed during the event.

Efficiency A kinetic data structure is called *efficient* if number of internal events / number of external events $= \mathcal{O}(\log^c n)$ for some constant c.

Responsiveness A kinetic data structure is called *responsive* if it is updated in $\mathcal{O}(\log^c n)$ time when an event occurs.

Compactness A kinetic data structure is called *compact* if it requires only $\mathcal{O}(n \log^c n)$ space on disk.

Locality A kinetic data structure is called *local* if the number of certificates associated with a single input point is at most $\mathcal{O}(\log^c n)$.

The goal is to find a kinetic data structure that is efficient, responsive, compact and local.

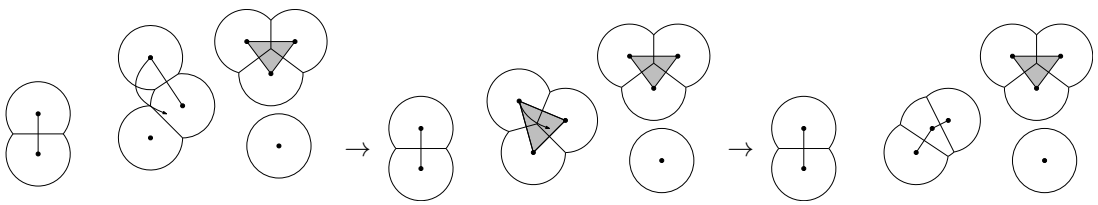
Chapter 3

Maintaining the Alpha Complex

In this chapter, we present a kinetic data structure to maintain the α-complex of a moving point set in d dimensions. As mentioned before, we are interested in the homology of an underlying space from which available data is sampled. However, since that space is not readily available, we approximate its homology by computing the homology of the union of balls around the sample points. So our goal is to maintain the homology of the union of balls around a vertex set S with radius α. By the nerve theorem, the union of balls is homotopy equivalent to the Čech complex. However, the Čech complex can have simplices of dimension up to $n-1$. In fact, If all vertices of S are close together such that all balls overlap, the Čech complex is an $(n-1)$-simplex with all its faces. The number of simplices in this case is exponential in n. However, since we know that the union of balls is homotopy equivalent to the α-complex, it also suffices to maintain the α-complex and compute its homology. We know that for constant dimension d, the number of simplices in the α-complex is polynomial in n so it is profitable to keep track of the α-complex and its homology since its homology is the same as the homology of the union of balls but its complexity is polynomial while the complexity of the Čech complex may be exponential.

At any point in time, the points in S form an α-complex for any $\alpha > 0$. As the points x_i move through space, the α-regions change location and shape. An example of a kinetic α-complex can be found in Figure 3.1. At some points in time, a topological event occurs. This is when a simplex gets added or removed from the alpha complex or when several simplices get removed and added at the same time. This happens for instance when two α-regions that did not share a face before now do share a face, thus creating the corresponding simplex in its nerve: the α-complex. It might also happen when some points move so far apart that the α-balls around those points no longer intersect. For example, if we compare Figure 3.1a to Figure 3.1b we see that a triangle and an edge were added to the complex. Later in the simulation, a triangle and an edge were deleted again, as depicted in Figure 3.1c.

We know that the α-complex is a subcomplex of the Delaunay complex. If both the Delaunay complex and the α-complex contain a simplex σ, we say that σ is *present* in the α-complex or



(a) An α-complex where the flight plan of one vertex is indicated.

(b) The α-complex after an event.

(c) The α-complex after another event.

Figure 3.1: Three frames of the simulation of a kinetic α-complex.

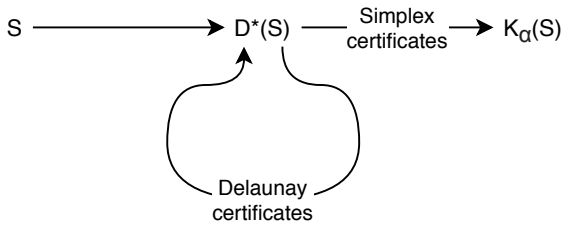


Figure 3.2: The control flow for maintaining the α-complex.

simply that σ is in the α-complex. By Lemma 2 we know when a simplex from the Delaunay complex is in the α-complex. This leads us to maintain the Delaunay complex and for each simplex in this complex we maintain whether this simplex is also present in the α-complex. Let $D(S)$ be the Delaunay complex of a point set S. If the point set used is clear from the context, we will also write D instead of $D(S)$. Using this method, we get two types of certificates. The first type are *Delaunay certificates* which are used to maintain the Delaunay complex. The second type are *simplex certificates* which are used to maintain the α-complex, given the Delaunay complex. A schematic overview of the control flow for maintaining the α-complex can be found in Figure 3.2. We will discuss both types of certificate separately.

3.1 Delaunay Certificates

The failures of Delaunay certificates induce *flip events* and these events are all we need to maintain the Delaunay complex. A flip event can be seen as the multidimensional equivalent of an edge-flip in planar graphs. A flip event occurs either when $d+2$ vertices lie on the same hypersphere and no points of S lie inside this hypersphere or when $d+1$ vertices lie on the same hyperplane and this hyperplane contains a face of the convex hull of S.

To see how many Delaunay certificates should be created we augment S with an extra, virtual point at 'infinity'. We call this extra point x_∞ and say that each $(d-1)$-simplex σ on the convex hull of S is a face of the d-simplex created by taking the convex hull of $\sigma \cup \{x_\infty\}$. We define S^* to be $S^* = S \cup \{x_\infty\}$. We call the standard Delaunay complex together with these new d-simplices the *augmented Delaunay complex*. We denote the augmented Delaunay complex by D^*. An example of a 2-dimensional augmented Delaunay complex can be found in Figure 3.3. In this figure, the Delaunay complex is denoted in solid lines and the augmented Delaunay complex is shown with dashed lines. Note that the Delaunay complex is a subcomplex of the augmented Delaunay complex

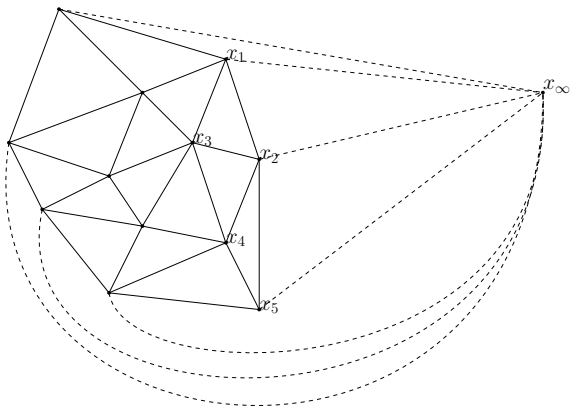


Figure 3.3: A 2-dimensional augmented Delaunay complex. The dashed lines show the difference between the normal Delaunay complex and the augmented Delaunay complex.

and therefore, the α-complex is a subcomplex of the augmented Delaunay complex. For our purposes, we will maintain the augmented Delaunay complex and for each simplex σ that contains x_∞ as a vertex, we know that σ is not present in the α-complex. The augmented Delaunay complex partitions the whole of $\mathbb{R}^d$. Therefore, a $(d-1)$-simplex in the augmented Delaunay complex always has exactly two d-cofaces. Each $(d-1)$-simplex in the augmented Delaunay complex is associated with exactly one Delaunay certificate and each Delaunay certificate maintains a list of the $d+2$ associated vertices that span this $(d-1)$-simplex and its two d-cofaces. Now, if x_∞ is one of the vertices in the list of associated vertices of a Delaunay certificate, then that Delaunay certificate describes the event when $d+1$ vertices lie on the same hyperplane on the convex hull of S. If x_∞ is not in the list of associated vertices of a Delaunay certificate, then the Delaunay certificate describes the event when $d+2$ vertices lie on the same (empty) hypersphere. For example, in Figure 3.3, the Delaunay certificate associated with edge (x_2, x_3) stores a list containing x_1, x_2, x_3 and x_4. Since x_∞ is not in this list, the Delaunay certificate describes the event that these four points lie on the same hypersphere. The Delaunay certificate associated with edge (x_1, x_2) stores a list containing x_1, x_2, x_3 and x_∞. Now, x_∞ is in this list so the Delaunay certificate describes the event that x_1, x_2 and x_3 lie on the same hyperplane on the convex hull of S. Also the edge (x_2, x_∞) is associated with a Delaunay certificate. This Delaunay certificate stores a list containing x_1, x_2, x_5 and x_∞.

To compute when a Delaunay certificate fails, we need to find the zeroes of its event function. If the list of a Delaunay certificate does not contain x_∞, the event function is $\textsc{Outside}(x_1, \ldots, x_{d+2})$, where $x_1, \ldots, x_{d+2}$ are the vertices in the list of the certificate. If the list of a Delaunay certificate does contain x_∞, the event function is $\textsc{Colinear}(x_1, \ldots, x_{d+1})$ where $x_1, \ldots, x_{d+1}$ are the other vertices in the list of the certificate. Now, Outside and Colinear can be defined as follows [3].

$$\textsc{Outside}(x_1, \ldots, x_{d+2}) = \det \begin{bmatrix} 1 & f_1^1 & \cdots & f_1^d & \|x_1\|^2 \\ 1 & f_2^1 & \cdots & f_2^d & \|x_2\|^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & f_{d+2}^1 & \cdots & f_{d+2}^d & \|x_{d+2}\|^2 \end{bmatrix}$$

$$\textsc{Colinear}(x_1, \ldots, x_{d+1}) = \det \begin{bmatrix} 1 & f_1^1 & \cdots & f_1^d \\ 1 & f_2^1 & \cdots & f_2^d \\ \vdots & \vdots & \ddots & \vdots \\ 1 & f_{d+1}^1 & \cdots & f_{d+1}^d \end{bmatrix}$$

Here, we have omitted the functionality on time. Note that Outside is a polynomial of degree $(d+2)\delta$ and Colinear is a polynomial of degree $d\delta$. We assume that d and δ are constant so the zeroes of Outside and Colinear can be computed in constant time using exact computation [37].

Once a Delaunay certificate fails, a flip event happens. The handling of such a flip event is the same when $d+2$ points lie on the same hypersphere as when $d+1$ points lie on the same hyperplane on the convex hull of S. To handle a flip event, we execute Algorithm 1. The notation $\widehat{x}$ means that x is excluded from a list. Essentially, this algorithm deletes all existing d-simplices formed by the $d+2$ concerned vertices and creates all other possible simplices. The correctness of this algorithm is proven in [3]. This means i simplices are deleted and j simplices are created, where $i+j = d+2$. Note that i and j are at least 2, since the cavity inside the convex hull of $x_1, \ldots, x_{d+2}$ cannot be filled by a single d-simplex. In 2 dimensions we know that $i = j = 2$ so all flip events are relatively simple, but in higher dimensions, they become more complex. In Algorithm 1, the sets F and G are used to make sure all deletions are executed before all the creations of simplices. During the deletions and creations, we make sure we keep the correct Delaunay certificates and simplex certificates. Which simplices should have a simplex certificate will be discussed later.

Algorithm 1 Handle flip event of $x_1, \ldots, x_{d+2}$

FLIP($x_1, \ldots, x_{d+2}$)
1: $G, F \leftarrow \emptyset$
2: **for** $i = 1$ **to** $d + 2$ **do**
3: $\quad \sigma \leftarrow (x_1, \ldots, x_{i-1}, \widehat{x_i}, x_{i+1}, \ldots, x_{d+2})$
4: $\quad$ **if** $\sigma \in D^*$ **then**
5: $\quad\quad G \leftarrow G \cup \{\sigma\}$
6: $\quad$ **else**
7: $\quad\quad F \leftarrow F \cup \{\sigma\}$
8: $\quad$ **end if**
9: **end for**
10: **for** $\sigma \in G$ **do**
11: $\quad$ Delete σ and its faces from D^*.
12: $\quad$ remove any certificates concerning σ or a face of σ from Q
13: **end for**
14: **for** $\sigma \in F$ **do**
15: $\quad$ add σ and its faces to D^*
16: **end for**
17: Compute which simplices from F or their faces are short and Gabriel and mark the corresponding simplices as part of the α-complex.
18: Find the simplices from F or their faces that need a simplex certificate and add its simplex certificate to Q.
19: Add the Delaunay certificate of each $(d-1)$-face of simplices in F to Q.

3.2 Simplex Certificates

Simplex certificates certify the correctness of the α-complex. They assume an augmented Delaunay complex is known and maintain which simplices are part of the α-complex and which are not. Simplices from the Delaunay complex change from being part of the α-complex to not being part of the α-complex at discrete moments. Such a moment always coincides with the failure of a simplex certificate. The failure of a simplex certificate is known as a *radius event*. Geometrically, a radius event is when a simplex turns from being short to being non-short or vice versa. So for a k-simplex $\sigma = (x_0, \ldots, x_k)$, this is when the radius of the $(k-1)$-sphere going through the points $x_0, \ldots, x_k$ becomes larger or smaller than α. By Lemma 2 we know that simplices are in the α-complex when they are short and Gabriel, or when they have a coface that is short and Gabriel. One might expect therefore that simplices may be added or deleted whenever simplices change from non-short to short (or vice versa) or when they change from non-Gabriel to Gabriel (or vice versa). However, Kerber et al. [27] prove that simplices may only be added or deleted when a simplex changes its shortness and not when a simplex changes its Gabrielity. This lemma and its proof are stated in three dimensions but they are also valid in any number of dimensions.

Simplex certificates may revolve around any simplex in the Delaunay complex. In theory, each simplex in the Delaunay complex could have a simplex certificate. However, this is not necessary since the (non)-shortness of some simplices is implied by the (non)-shortness of other simplices. For any simplex we know that the radius of its circumsphere is larger than, or equal to the radius of the circumsphere of its faces. So for any k-simplex σ with l-face τ, $l \leq k$, we have RADIUS(CIRCUMSPHERE(τ)) $\leq$ RADIUS(CIRCUMSPHERE(σ)). The result is that if a simplex is short, all its faces are short. It also means that if a simplex is non-short, all its cofaces are non-short. So only the shortness of simplices of which all faces are short and all cofaces are non-short is not implied by other simplices. This is why we only have simplex certificates for simplices with only short faces and non-short cofaces. In Figure 3.4 we see an example of an abstract simplicial

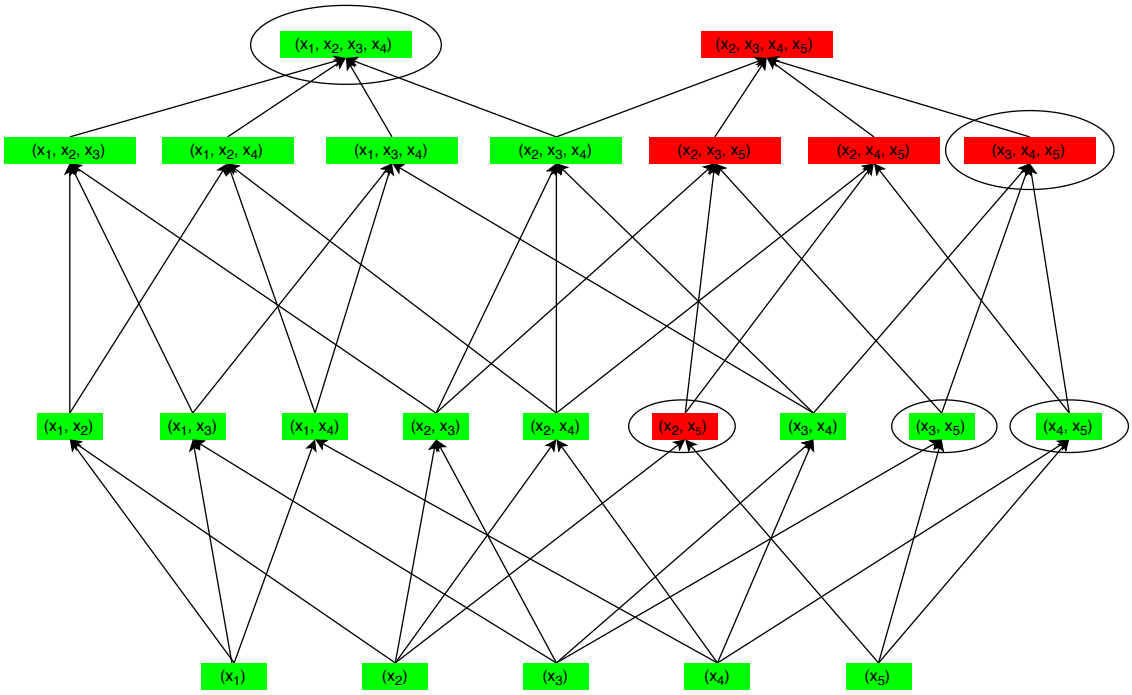


Figure 3.4: Example of an abstract simplicial complex. Short simplices are marked green, non-short simplices are marked red and the simplices with a simplex certificate are circled. The arrows denote that simplices are the face of other simplices.

complex in which the shortness of simplices is marked with colors and the simplices that have a simplex certificate are circled. Not associating a certificate with each simplex significantly reduces the number of simplex certificates.

The simplex certificate of a simplex fails if the radius of the circumsphere of that simplex equals α. The radius $\mathcal{R}$ of the circumsphere of a k-simplex $\sigma = (x_0, \ldots, x_k)$ satisfies $-2\mathcal{R}^2 = \frac{\det(L)}{\det(M)}$ [12] where

$$M = \begin{pmatrix} 0 & 1 & 1 & \ldots & 1 \\ 1 & \|x_0 - x_0\|^2 & \|x_0 - x_1\|^2 & \ldots & \|x_0 - x_k\|^2 \\ 1 & \|x_1 - x_0\|^2 & \|x_1 - x_1\|^2 & \ldots & \|x_1 - x_k\|^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \|x_k - x_0\|^2 & \|x_k - x_1\|^2 & \ldots & \|x_k - x_k\|^2 \end{pmatrix}$$

and

$$L = \begin{pmatrix} \|x_0 - x_0\|^2 & \|x_0 - x_1\|^2 & \ldots & \|x_0 - x_k\|^2 \\ \|x_1 - x_0\|^2 & \|x_1 - x_1\|^2 & \ldots & \|x_1 - x_k\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|x_k - x_0\|^2 & \|x_k - x_1\|^2 & \ldots & \|x_k - x_k\|^2 \end{pmatrix}$$

The event function of a simplex certificate is called AlphaRadius(σ) where σ is the simplex the certificate revolves around and AlphaRadius is defined as follows.

$$\text{AlphaRadius}(\sigma) = \det(L) + 2\alpha^2 \det(M)$$

Note that the right-hand side of this equation is a polynomial of degree $2\delta(k+1)$. We know that $k \leq d$ and we assume d and δ to be constant. Therefore, computing the zeroes of AlphaRadius can be done in constant time for any dimension k.

Algorithm 2 Simplex turns short.

SHORT(σ)
1: $k \leftarrow$ DIM(σ)
2: **for** all $(k-1)$-faces τ of σ **do**
3: remove radius certificate of τ from Q
4: **end for**
5: Add radius certificate of σ to Q.
6: **for** all $(k+1)$-cofaces ρ of σ **do**
7: **if** $\nexists$ a k-face σ' of ρ such that σ' is non-short **then**
8: add radius certificate of ρ to Q.
9: **end if**
10: **end for**
11: **if** σ is Gabriel **then**
12: $K_\alpha \leftarrow K_\alpha \cup \{\sigma\}$
13: $\forall$ faces τ of σ: $K_\alpha \leftarrow K_\alpha \cup \{\tau\}$
14: **end if**

Algorithm 3 Simplex turns non-short.

NON-SHORT(σ)
1: $k \leftarrow$ DIM(σ)
2: **for** all $(k+1)$-cofaces ρ of σ **do**
3: remove radius certificate of ρ from Q
4: **end for**
5: add radius certificate of σ to Q.
6: **for** all $(k-1)$-faces τ of σ **do**
7: **if** $\nexists$ a k-coface σ' of τ such that σ' is short **then**
8: add radius certificate of τ to Q.
9: **end if**
10: **end for**
11: **if** $\sigma \in K_\alpha$ **then**
12: $K_\alpha \leftarrow K_\alpha \setminus \{\sigma\}$
13: determine which faces of σ are short and Gabriel and remove the appropriate simplices from K_α
14: **end if**

If a simplex certificate fails, a radius event happens. All we know at that point is that the radius of the circumsphere of a simplex σ equals α. We distinguish two cases. In the first case, σ changes from non-short to short. In the second case, σ changes from short to non-short. In the first case we execute Algorithm 2 and in the second case we execute Algorithm 3. We consider first the case that a k-simplex σ turns from non-short to short. Denote by $\tau_1, \ldots, \tau_{k+1}$ all the $(k-1)$-faces of σ and by $\rho_1, \ldots, \rho_w$ all the $(k+1)$-cofaces of σ. We assume non-degeneracy in the sense that a simplex and its face cannot turn (non-)short at the same time. This implies that $\tau_1, \ldots, \tau_{k+1}$ are all short at the time that σ changes and $\rho_1, \ldots, \rho_w$ are all non-short. Since σ was non-short, the τ_i might have a simplex certificate. Since σ is short after the radius event, the shortness of the τ_i is implied by the shortness of σ, and the simplex certificates of the τ_i can be deleted. This is what happens on lines 2 - 4 of Algorithm 2. Since all cofaces of σ are necessarily non-short and all faces of σ are necessarily short, we need to keep the simplex certificate of σ in the event queue. Depending on the choice of implementation the other radius events of the certificate concerning σ are already in Q, still have to be computed or only have to be inserted into Q. For every ρ_i we know that all their cofaces are non-short. We also know that one of their faces turned from non-short to short. It might be that σ was the last non-short face of ρ_i. In that case, the non-shortness of ρ_i is no longer implied by one of its faces and ρ_i should get a simplex certificate. This is exactly what lines 6 to 10 do. Lastly, it might be the case that some simplices get added to the α-complex. If σ is Gabriel then it should be added to the α-complex along with all its faces. This does not change any certificates.

We now consider the case that a k-simplex σ changes from short to non-short and we execute Algorithm 3. We denote the $(k-1)$-faces and the $(k+1)$-cofaces of σ again by $\tau_1, \ldots, \tau_{k+1}$ and $\rho_1, \ldots, \rho_w$, respectively, and we again assume non-degeneracy. Before the change, the shortness of σ implied the shortness of the τ_i and after the change, the non-shortness of σ implies the non-shortness of the ρ_i. This means that the simplex certificates of the ρ_i should be deleted (lines 2 - 4) and the simplex certificate of τ_i needs to be added if σ was the only simplex implying the shortness of τ_i. Again, we need to keep the simplex certificate of σ, since all its faces are short and all its cofaces are non-short. These steps are executed in lines 2 - 10. If σ was part of the α-complex at the time it changes to non-short, it should be deleted from the α-complex. All faces of σ that were part of the α-complex but are not short and Gabriel, nor have another coface that is short and Gabriel should also be deleted from the α-complex.

3.3 Updating of Vertex Motion

Changing the motion of a vertex means that at least one of the coordinate functions f_i^j that describe the motion of a vertex x_i is replaced by a new function, say, $\widehat{f}_i^j$. When the path of a vertex is updated, we need to adapt all the certificates that contain this vertex and take into account the new coordinate function $\widehat{f}_i^j$. For each Delaunay certificate that contains x_i in its list we recompute the failure times using the new $\widehat{f}_i^j$, delete the old certificates from the event queue and add the new ones. For each simplex that contains x_i and has a simplex certificate associated with it, we recompute its failure times, delete the old certificates from the event queue and add the new ones.

3.4 Quality Analysis of Maintaining the Alpha Complex

The quality of a kinetic data structure is measured by its efficiency, responsiveness, compactness and locality. We will now address these features individually. Some properties of the data structure that we need for this analysis are the maximum number of cofaces of a simplex σ, the total number of simplices in the Delaunay complex and the number of certificates in event queue Q. We denote by ρ_σ the set of $(k+1)$-cofaces of σ and by $|\rho_\sigma|$ the number of $(k+1)$-cofaces of σ. In theory, given a k-simplex σ, all vertices from S that do not span σ could form a coface together with the vertices from σ. This means that $|\rho_\sigma|$ could be as large as $n-k-1$. However, in many practical

applications, such a configuration is unlikely. In fact, the maximum number of cofaces of a simplex σ is bounded by the maximum degree of its vertices. Let ρ denote the maximum degree of a vertex in S, then $|\rho_\sigma| = \mathcal{O}(\rho)$. For example, if the vertices of S are distributed uniformly at random in a convex shape, ρ is expected to be only $\mathcal{O}\left(\log^{2+\varepsilon} n\right)$ and therefore, $|\rho_\sigma| = \mathcal{O}\left(\log^{2+\varepsilon} n\right)$ where $\varepsilon > 0$ is an arbitrarily small constant [10]. The total number of simplices in the Delaunay complex is $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ [30, 35]. This upper bound is only attained in some special cases. Again, in many applications the distribution of the points is such that the number of simplices is expected to be much smaller. For example, if the points are chosen independently from a uniform distribution on the interior of a d-ball, one may expect the number of simplices to be only linear in n [17]. Denote by n_k the number of k-simplices in the Delaunay complex. We find that $n_k = \mathcal{O}(n_{k+1})$ since each $(k+1)$-simplex can have only a constant number of k-faces. Therefore, the total number of simplices $\sum_{k=0}^{d} n_k = \mathcal{O}(n_d)$ and $n_d = \mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$. The number of certificates in the event queue Q is linear in the number of simplices in the Delaunay complex. Since Q is implemented as a priority queue, we can look up, insert and delete certificates in Q in time logarithmic in the number of elements in Q. Therefore, looking up, inserting and deleting certificates from Q takes $\mathcal{O}(\log n)$ time.

3.4.1 Efficiency

A kinetic data structure is called efficient if the worst case number of internal events divided by the worst case number of external events is $\mathcal{O}(\log^c n)$ where c is a constant. In our case, the external events are those events that cause a topological change in the α-complex. The internal events are the flip events and the radius events that do not incur a topological change in the α-complex. We now show that the worst case number of external events is always of the same order of magnitude as the worst case number of internal events. Let S' be a simulation that is run on the time interval $t \in [0, 1]$ with parameter α and let S' be such that the worst case number of internal events is attained. Let ξ be the number of events handled in the simulation, ξ^f the number of flip events and ξ^r the number of radius events so that $\xi = \xi^f + \xi^r$. Because a simplex certificate can only fail a constant number of times, each simplex in the simulation can have a constant number of radius events. Each flip event may create a constant number of simplices so the number of radius events $\xi^r = \mathcal{O}\left(\xi^f + n_d\right)$ and, therefore, $\xi = \mathcal{O}\left(\xi^f + n_d\right)$. However, since simulations exist where $\xi^f = \Omega\left(n^{\lceil d/2 \rceil + 1}\right)$, we find that $\xi = \mathcal{O}\left(\xi^f\right)$ [3].

We bound the simulation with $d+1$ additional points such that all points of S' stay within the convex hull of these extra points and each flip event in the simulation still happens. This is possible by putting these additional points far enough from the points of S'. Each flip event that involved x_∞ will now involve one of these additional points so the total number of flip events remains the same. Since no flip event happens that involves x_∞, the radius of the circumspheres of the simplices never approach infinity. Therefore, we can choose a new parameter α' such that α' is larger than the largest radius of a circumsphere of a simplex throughout the entire simulation. If we now run the simulation with this new parameter α' and this new point set, each simplex from the Delaunay complex will also be in the α-complex. Also, this new simulation will have $\Omega\left(\xi^f\right)$ events. Because the α-complex now equals the Delaunay complex, each event is an external event. Therefore, we have created a simulation with $\Omega(\xi)$ external events. We conclude that the worst case number of internal events is of the same order of magnitude as the worst case number of external events and their ratio is $\mathcal{O}(1)$.

3.4.2 Responsiveness

A kinetic data structure is called responsive if an event can be handled in poly-logarithmic time. We first analyze the handling of a flip event and then the handling of a radius event. When a flip event happens, we execute Algorithm 1, which is called the Flip algorithm. To compute the running time of the Flip algorithm, we go through it line by line. The first line clearly takes constant time. In the loop initiated on line 2, we pass $d+2$ iterations which is a constant number. The creation of the simplex σ takes constant time. Checking the presence of an element in a

set takes logarithmic time in the number of elements in the set so checking the presence of σ in D^* takes logarithmic time in the number of simplices in D^*. Since the number of simplices in D^* is always polynomial in n, checking the presence of σ takes $\mathcal{O}(\log n)$ time. In the loop initiated on line 10, there are a constant number of iterations, since the size of G cannot exceed d. Finding the simplices to delete in the data structure that stores D^* again takes $\mathcal{O}(\log n)$ time. Removing the certificates from Q takes $\mathcal{O}(\log n)$ time. The loop initiated on line 14 contains a constant number of iterations, since the size of F cannot exceed d. Each addition of a simplex to D^* takes $\mathcal{O}(\log n)$ time. In the lines after the loop we check which simplices are short and add the appropriate certificates to Q. To compute whether a simplex is short, we need constant time. There are only a constant number of simplices in F, so computing which simplices need a simplex certificate takes constant time. Adding these certificates to Q takes $\mathcal{O}(\log n)$ time. To compute whether a k-simplex σ is Gabriel we need to check whether there are vertices in the circumball of σ. One way to do this is to check all vertices in S and see if there are any in the circumball of σ. However, it suffices to check only vertices that are contained in cofaces of σ [8]. This way, checking whether σ is Gabriel takes $\mathcal{O}(|\rho_\sigma|)$ time. Computing the Delaunay certificates of the $(d-1)$-faces of simplices in F takes constant time because there are only a constant number of simplices in F. Adding these certificates to Q takes $\mathcal{O}(\log n)$ time. So in total, Algorithm 1 runs in $\mathcal{O}(\rho + \log n)$ time where ρ denotes the maximum degree of a vertex.

When a radius event happens, we execute Algorithm 2 or Algorithm 3, depending on whether σ turns short or non-short. These algorithms are called the SHORT algorithm and the NON-SHORT algorithm respectively. To compute the running time of SHORT and NON-SHORT we analyze these algorithms line by line. The first line of the SHORT algorithm clearly takes constant time. The loop initiated at line 2 contains a constant number of iterations, in which $\mathcal{O}(\log n)$ time is spent in each iteration. Line 5 also takes $\mathcal{O}(\log n)$ time. The loop initiated on line 6 has $|\rho_\sigma|$ iterations. Each iteration in the loop might take $\mathcal{O}(\log n)$ time because checking the guard in the if-statement takes constant time and adding the certificate to Q takes $\mathcal{O}(\log n)$ time. So in total, this loop takes $\mathcal{O}(\rho \log n)$ time. Checking whether σ is Gabriel takes $\mathcal{O}(|\rho_\sigma|)$ time. Marking σ present in K_α takes constant time and marking all faces of σ also takes constant time because a simplex can only have a constant number of faces. It total, Algorithm 2 runs in $\mathcal{O}(\rho \log n)$ time.

We now analyze the running time of the NON-SHORT algorithm. The first line clearly takes constant time. The loop initiated on line 2 has $|\rho_\sigma|$ iterations and each iteration takes $\mathcal{O}(\log n)$ time. Line 5 takes $\mathcal{O}(\log n)$ time. Line 6 passes through a constant number of iterations but checking the guard on line 7 might take $\mathcal{O}(|\rho_\tau|)$ iterations, where ρ_τ denotes the set of k-cofaces of τ. Adding certificates on line 8 takes $\mathcal{O}(\log n)$ time. Checking whether σ is part of K_α on line 11 takes constant time. Marking σ on line 12 also takes constant time. To find which simplices are short and Gabriel and which simplices have to be removed from K_α, we have to do a constant number of Gabriel checks. Each Gabriel check of a simplex takes time proportional to the number of cofaces of that simplex so this takes $\mathcal{O}(\rho)$ time. In total, executing Algorithm 3 takes $\mathcal{O}(\rho \log n)$ time. So processing any event in the event queue takes $\mathcal{O}(\rho \log n)$ time.

3.4.3 Compactness

A kinetic data structure is called compact if it takes $\mathcal{O}(n \log^c n)$ storage. We need to store all simplices in the augmented Delaunay complex. Also, the number of certificates is at most linear in the number of simplices in D^*. Therefore, the size of the data structure will be $\mathcal{O}(n_d)$. We recall that in the worst case $n_d = \mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$, but under some assumptions on the point set distribution we may expect that $n_d = \mathcal{O}(n)$.

3.4.4 Locality

A kinetic data structure is called local if the maximum number of certificates associated with a single vertex is at most $\mathcal{O}(\log^c n)$. The number of certificates associated with a vertex x_i is dependent on the number of simplices that contain that vertex. The number of simplices that contain a vertex is bounded by the degree of that vertex so the number of certificates associated

with a vertex is $\mathcal{O}(\rho)$. In theory, each d-simplex in the Delaunay complex could contain x_i as a vertex. In that case, the number of certificates associated with x_i is $\mathcal{O}(n_d)$. However, if we assume the points to be distributed uniformly at random in a smooth, convex shape we may expect that $n_d = \mathcal{O}(n)$ and that $\rho = \mathcal{O}\left(\log^{2+\varepsilon} n\right)$ [10]. Away from the boundary of the convex shape we may even expect that $\rho = \Theta(\log n / \log\log n)$ [7]. Whether ρ is expected to be $\Theta(\log n / \log\log n)$ throughout the entire shape is, to the best of our knowledge, still an open problem [10].

If we combine the results of the four aspects we obtain the following theorem.

Theorem 2. *Let $\alpha > 0$ and a set S of n moving points $x_1, \ldots, x_n$ be given, where the movement of the x_i is modeled by coordinate functions which are polynomials of degree δ. Then, the α-complex of S can be maintained with an $\mathcal{O}(1)$ ratio between the internal and external events, $\mathcal{O}(\rho \log n)$ update time, $\mathcal{O}(n_d)$ storage space and $\mathcal{O}(\rho)$ certificates per vertex. In the worst case, these bounds may be $\mathcal{O}(1)$, $\mathcal{O}(n \log n)$, $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ and $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ respectively. If we assume the points to be chosen uniformly at random from a convex shape, these bounds are expected $\mathcal{O}(1)$, $\mathcal{O}\left(\log^{3+\varepsilon} n\right)$, $\mathcal{O}(n)$ and $\mathcal{O}\left(\log^{2+\varepsilon} n\right)$ respectively.*

We find that the bounds vary strongly between different point sets based on their distribution. Two point sets that both contain n points may have completely different responsiveness, locality and compactness. In some simulations, the kinetic data structure may be expected to be efficient, responsive, local and compact but in more unfortunate configurations it may only be efficient.

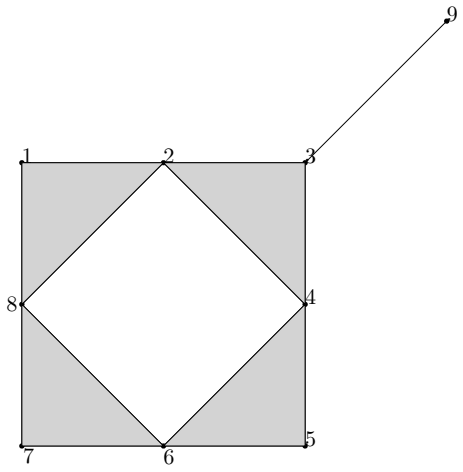
Chapter 4

Dynamically Maintaining Homology

The result of the algorithms from Chapter 3 is a dynamically changing simplicial complex. By that we mean that we have an abstract simplicial complex which undergoes topological changes at discrete moments in time. In this chapter we present the algorithms that maintain the homology of this simplicial complex. The input for the algorithms in this chapter is an augmented Delaunay complex D^* where each simplex bears a mark that tells us if it is in the α-complex or not. This input is subject to dynamical changes that may involve flip events that change D^* and radius events that may change the marks on the simplices. When a simplex was not in the α-complex before an event and it is in the α-complex after the event, we say the simplex gets *added* to the α-complex. Similarly, when a simplex was in the α-complex before an event and it is not in the α-complex after the event, we say the simplex gets *removed* or *deleted* from the α-complex. Throughout the maintenance of the homology of the α-complex, we may assume the Delaunay property holds on D^*. This means that the circumball of each d-simplex has empty interior so each d-simplex is Gabriel. We may also assume that the α-complex is a valid simplicial complex so we know that the two properties of a simplicial complex hold. The desired result of the algorithms is the homology of the α-complex. That is, we would like to have a representative cycle readily available for each homology class. On top of that, we would like to know the number of independent homology classes per homology group: the Betti numbers.

The data structure we propose makes use of so-called *k-boundary matrices* and *k-homology matrices*. For each $1 \leq k \leq d$, the k-boundary matrix $\delta^k \in (\mathbb{Z}/2\mathbb{Z})^{n_k \times n_{k-1}}$ stores which $(k-1)$-simplices from the α-complex form the boundary of each k-simplex. Each column in δ^k represents a k-simplex and each row in δ^k represents a $(k-1)$-simplex. Now, $\left(\delta^k\right)_{ij} = 1$ if the j^{th} $(k-1)$-simplex is a face of the i^{th} k-simplex and $\left(\delta^k\right)_{ij} = 0$ otherwise. Each k-simplex has $k+1$ $(k-1)$-faces so there are exactly $n_k(k+1)$ ones in δ^k. We assume that $n_{k-1} \gg k$ so the δ^k are sparse matrices and they are stored in sparse format. We assume that we can add elements to δ^k in constant time, delete rows and columns in time proportional to the number of nonzero entries in that row or column and multiply δ^k with any test vector using time proportional to the number of nonzero entries in δ^k. These assumptions are met if the δ^k are stored as a list of rows and a list of columns where each row or column is stored as a list of nonzero entries.

For each $0 \leq k \leq d-1$, The k-homology matrix $H^k \in (\mathbb{Z}/2\mathbb{Z})^{n_k \times \beta_k}$ provides us with a basis for the k-homology group by storing representative cycles of a set of independent k-homology classes. Because we store only independent k-homology classes, the size of the k-homology matrix tells us β_k. A homology class of which no representative cycle is stored in the matrix can be found by adding together homology classes of which a representative cycle is stored in H^k. So for any homology class h of the simplicial complex we either store a representative cycle c or we store a set of representative cycles $c_1, \ldots, c_r$ of homology classes $h_1, \ldots, h_r$ such that $c = c_1 + \cdots + c_r$ is a representative cycle of h. A k-homology matrix stores which k-simplices from the α-complex



(a) A simplicial complex with numbered vertices

δ^2	128	234	456	678
12	1			
18	1			
23		1		
24		1		
28	1			
34		1		
39				
45			1	
46			1	
56			1	
67				1
68				1
78				1

(b) The 2-boundary matrix of the simplicial complex.

δ^1	12	18	23	24	28	34	39	45	46	56	67	68	78
1	1	1											
2	1		1	1	1								
3			1			1	1						
4				1		1		1	1				
5								1		1			
6									1	1	1	1	
7											1		1
8		1			1							1	1
9							1						

(c) The 1-boundary matrix of the simplicial complex

H^1	12	18	23	24	28	34	39	45	46	56	67	68	78
1				1	1				1			1	

(d) A possible 1-homology matrix of the simplicial complex.

H^0	1	2	3	4	5	6	7	8	9

(e) The 0-homology matrix of the simplicial complex.

Figure 4.1: A simplicial complex with its corresponding boundary matrices and homology matrices.

are part of the representative cycles of the homology classes of the α-complex. Each column in H^k represents a k-simplex and each row in H^k represents a representative cycle of a k-homology class. Now, $\left(H^k\right)_{ij} = 1$ if the i^{th} k-simplex is a simplex in the j^{th} representative k-cycle and $\left(H^k\right)_{ij} = 0$ otherwise. Note that any representative cycle can be chosen for each homology class in H^k and any basis of homology classes can be chosen. This means that the k-homology matrices are, in general, not unique. The H^k could be non-sparse in theory, but we store it in sparse format anyway because that will be faster in many cases. We use the same implementation for the homology matrices as for the boundary matrices. We assume that initially, the boundary matrices and the homology matrices are computed correctly and we will only show how to update them. Figure 4.1a shows an example of a simplicial complex and Tables 4.1c, 4.1b, 4.1e and 4.1d show δ^1, δ^2, H^0 and H^1 belonging to that simplicial complex respectively. A schematic overview of the control flow including the maintenance of the homology can be found in Figure 4.2.

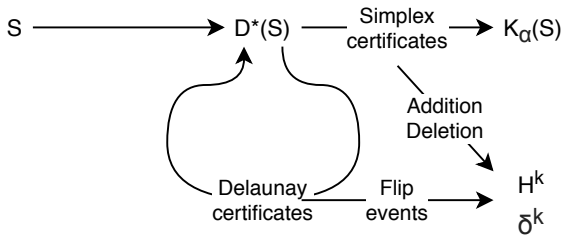


Figure 4.2: The control flow for maintaining the α-complex and its homology.

4.1 Algorithms

We need to be able to maintain the homology of the α-complex during a radius event or a flip event. In a radius event, only the addition or deletion of a simplex may change the homology of the α-complex. A flip event can be modeled as the deletion of all simplices involved before the event and the addition of all simplices involved after the event. This means that we can maintain the homology of the α-complex during radius and flip events if we can maintain the homology of the α-complex during the addition or deletion of a simplex. That is why the algorithms described below will focus on the addition and deletion of simplices.

We slightly abuse the notation $\partial_k(\sigma)$ and say that $\partial_k(\sigma)$ is a vector of length n_{k-1} with a 1 on the i^{th} position if the i^{th} $(k-1)$-simplex is a face of σ and a 0 otherwise. When a k-simplex σ gets added, we check whether $\partial_k(\sigma)$ is a linear combination of the existing columns of δ^k. If $\partial_k(\sigma)$ is a linear combination on the existing columns of δ^k, we add the k-chain consisting of σ and all the simplices from the linear combination as a new k-homology class to H^k. If $\partial_k(\sigma)$ is not a linear combination on the existing columns of δ^k, we find a different linear combination. We find rows of H^{k-1} and columns of δ^k such that $\partial_k(\sigma)$ equals the sum of these rows and columns. Once we have such a linear combination, we delete one of the rows of H^{k-1} that was in this linear combination. We then add $\partial_k(\sigma)$ as a new column to δ^k. This column will contain $k+1$ ones, since σ has $k+1$ $(k-1)$-faces, which are all present in the α-complex. We also add a row to δ^{k+1} that represents σ. This row contains only zeroes, since σ has no cofaces in the α-complex.

When a k-simplex σ gets deleted from the α-complex, the inverse of an addition happens. To update the boundary matrices, we remove the corresponding row of δ^{k+1} and column of δ^k. We check if the column i in H^k that represents σ contains a nonzero entry. If column i of H^k contains a nonzero entry, let j be the row of H^k such that $H^k_{ij} = 1$. Now, for all other rows j' such that $H^k_{ij'} = 1$ we replace row j' by the sum of row j' and row j, thus creating a zero at $H^k_{ij'}$. Now, we simply remove row j and column i from H^k. If column i of H^k does not contain a nonzero entry, we simply add $\partial_k(\sigma)$ as a new row to H^{k-1}.

4.2 Correctness Proof

We rely on the observation that the addition of a k-simplex σ always either creates a k-homology class, or deletes a $(k-1)$-homology class [13]. The deletion of σ is the inverse of the addition of σ so the deletion of a k-simplex will either cause the deletion of a k-homology class, or the creation of a $(k-1)$-homology class. This means that whenever we add a simplex, we have to determine whether it creates a k-homology class or deletes a $(k-1)$-homology class. To do this, we use the following lemma.

Lemma 4. *Let $\sigma_1, \ldots, \sigma_{n_k}$ be the k-simplices of an α-complex. The addition of a new k-simplex σ creates a new k-homology class if and only if $\partial_k(\sigma)$ is a linear combination of $\partial_k(\sigma_1), \ldots, \partial_k(\sigma_{n_k})$.*

Proof. Assume $\partial_k(\sigma)$ is a linear combination of $\partial_k(\sigma_1), \ldots, \partial_k(\sigma_{n_k})$. Then, $\partial_k(\sigma) = \sum_{i \in I} \partial_k(\sigma_i)$ for some index set I. Since our arithmetic is modulo 2, this implies $\partial_k\left(\sigma + \sum_{i \in I} \sigma_i\right) = 0$. Therefore, $c = \{\sigma\} \cup \{\sigma_i | i \in I\}$ is a cycle. Also, since σ was just added, no cofaces of σ are in

the α-complex. This means that c is not the boundary of some $(k+1)$-chain and it must be a representative cycle of a valid k-homology class.

To prove the reverse, we assume σ creates a new k-homology class h. Let I be the index set of the simplices in a representative cycle c of h so that $c = \{\sigma\} \cup \{\sigma_i | i \in I\}$. Since c is a representative cycle of a homology class, we know its boundary is trivial so $\partial_k(c) = 0$. We conclude that $\partial_k(\sigma) = \sum_{i \in I} \partial_k(\sigma_i)$ and we are done. □

When we delete a simplex we need to determine whether it deletes a k-homology class or creates a $(k-1)$-homology class. To do this, we use the following lemma.

Lemma 5. *The deletion of a k-simplex σ from the α-complex deletes a k-homology class if σ is part of one of the representative cycles in H^k and it creates a $(k-1)$-homology class if it is not.*

Proof. Because σ does not have a coface in the α-complex, it is not part of the boundary of a $(k+1)$-chain. This means that a cycle c containing σ cannot be part of the boundary group and must be a representative cycle of a homology class. It also means that such a cycle c can only be homologous to other cycles that contain σ. Therefore, if a homology class contains a representative cycle that contains σ, all representative cycles of that homology class contain σ. This means we can speak of homology classes that contain σ. If there is a homology class h that contains σ then there must be a homology class in H^k that contains σ because otherwise, the basis of representative cycles from H^k cannot create h and we assume that H^k is correct before the deletion. This means that σ is part of a cycle in a homology class if and only if there is a nonzero element in the column representing σ in H^k.

If there are only zeroes in the column representing σ and there are no homology classes that contain σ, then the boundary of σ must be a representative cycle of a $(k-1)$-homology class. By Lemma 3 we know that $\partial_k(\sigma)$ is a cycle. Also, $\partial_k(\sigma)$ cannot be the boundary of a k-chain c^* other than σ because that would imply that the boundary of σ with c^* is 0 and $h' = \sigma \cup c^*$ would be a representative cycle of a k-homology class. Therefore, $\partial_k(\sigma)$ must be a representative cycle of a $(k-1)$-homology class.

Assume there is a nonzero entry in the column i representing σ in H^k. Let j be a row of H^k such that $H^k_{ij} = 1$. Let c be the representative cycle of row j. Let index set I be such that $c = \{\sigma\} \cup \{\sigma_i | i \in I\}$. Now, because c is a cycle, we know that it has trivial boundary and therefore we know that $\partial_k(\sigma) = \sum_{i \in I} \partial_k(\sigma_i)$. By Lemma 4 we now know that the difference between K_α and $K_\alpha \setminus \{\sigma\}$ is one k-homology class. Therefore, we should delete a k-homology class. Using that the deletion of a k-simplex can either create a $(k-1)$-homology class or delete a k-homology class we know that these two situations are mutually exclusive.

□

Now we know that we can determine whether the addition of a k-simplex σ creates a k-homology class or deletes a $(k-1)$-homology class and whether the deletion of a k-simplex deletes a k-homology class or creates a $(k-1)$-homology class. Next, we show that each of these four cases is handled correctly. The proof of Lemma 4 constructs a homology class from the linear dependency of the $\partial(\sigma_i)$. This homology class contains σ, so it could not exist before the addition of σ and it is correct to add it to H^k.

If the addition of k-simplex σ deletes a $(k-1)$-homology class, we know that $\partial_k(\sigma)$ is already present in the α-complex. We know that $\partial_k(\sigma)$ is a cycle because of Lemma 3. The boundary $\partial_k(\sigma)$ cannot be the boundary of a k-chain c, because then, $c \cup \{\sigma\}$ would be a representative cycle of a k-homology class, and we had assumed the addition of σ deletes a $(k-1)$-homology class so it cannot create a k-homology class. Therefore, $\partial_k(\sigma)$ is a representative cycle of a $(k-1)$-homology class h. After the addition of σ, every representative cycle of h is homologous to the empty set so h must be deleted from the set of homology classes. We know that h is represented in H^{k-1}. To be precise, there exists a representative cycle c' of h such that $c' = \sum_{j \in J} H^{k-1}_j$ for some index set J, where H^{k-1}_j denotes the j^{th} row of H^{k-1}. Since c' is homologous to $\partial_k(\sigma)$, we know that $\partial_k(\sigma) + \sum_{i \in I} \partial_k(\sigma_i) = \sum_{j \in J} H^{k-1}_j$ for some index set I, where the σ_i denote the other k-simplices in the α-complex. This means that after the addition of σ, $H^{k-1}_{j^*}$ is homologous to $\sum_{j \in J \setminus \{j^*\}} H^{k-1}_j$

for any $j^* \in J$. If we remove $H^{k-1}_{j^*}$ from H^{k-1} then we can still construct a representative cycle homologous to $H^{k-1}_{j^*}$ because $\sum_{j \in J \setminus \{j^*\}} H^{k-1}_j$ is homologous to $H^{k-1}_{j^*}$. This means that we can also still construct the homology classes we could construct using $H^{k-1}_{j^*}$ except for h and classes including h. So effectively, we have deleted h.

If the deletion of k-simplex σ creates a $(k-1)$-homology class, we add $\partial_k(\sigma)$ to H^{k-1}. The proof of Lemma 5 shows that $\partial_k(\sigma)$ is a representative cycle of a valid homology class. This homology class could not have existed before the deletion of σ, because the presence of σ would make sure that all elements of this homology class are homologous to the empty set. Therefore, it is correct to add it to H^{k-1}. If the deletion of k-simplex σ deletes a k-homology class, we first add one representative cycle to several other representative cycles. We now show that these additions do not change the set of homology classes that we can construct. We consider the addition of a row j^* to another row i^* and note that if this addition does not change the set of homology classes that we can construct, then all consecutive additions also do not change this. Let H^k be the homology matrix before the addition. Let H^k_i denote the i^{th} row of H^k. Let $\widehat{H^k}$ be the homology matrix after the addition so $\widehat{H^k_i} = H^k_i$ for all $i \neq i^*$ and $\widehat{H^k_{i^*}} = H^k_{i^*} + H^k_{j^*}$. Now, any representative cycle c that was represented in H^k, can be written as $c = \sum_{i \in I} H^k_i$ or as $c = \sum_{i \in I} H^k_i + H^k_{i^*}$ for some index set I that does not contain i^*. In the first case we find that $c = \sum_{i \in I} \widehat{H^k_i}$, so c is represented in the new matrix. In the second case we find that

$$\begin{aligned} c &= \sum_{i \in I} H^k_i + H^k_{i^*} \\ &= \sum_{i \in I} \widehat{H^k_i} + H^k_{i^*} + H^k_{j^*} + H^k_{j^*} \\ &= \sum_{i \in I} \widehat{H^k_i} + \widehat{H^k_{i^*}} + \widehat{H^k_{j^*}}. \end{aligned}$$

So c is still represented in the new matrix.

We now prove the correctness of deleting the row of H^k that contains a 1 in the column representing σ.

Lemma 6. *Let H^k be such that column j^* contains a nonzero entry only at row i^*, where j^* represents k-simplex σ. Then deleting row i^* and column j^* from H^k yields a correct k-homology matrix for $K_\alpha \setminus \sigma$.*

Proof. Let $\widehat{H^k}$ be the k-homology matrix after the deletion of row i^* and column j^*. We first show that all homology classes that contained σ in H^k are no longer represented in $\widehat{H^k}$. Let $c = \sum_{i \in I} \widehat{H^k_i} + \sum_{j \in J} \delta^{k+1}_j$ be a representative cycle of a homology class represented using the new matrices. Since σ does not have any cofaces in the α-complex, σ is not represented in any of the δ^{k+1}_j. Also, none of the $\widehat{H^k_i}$ contain σ because of the operations that were just executed. Therefore, c does not contain σ, and none of the homology classes represented by $\widehat{H^k}$ contains σ. Finally, we show that all homology classes that did not contain σ in H^k are still represented in $\widehat{H^k}$. Let $c = \sum_{i \in I} H^k_i + \sum_{j \in J} \delta^{k+1}_j$ be a representative cycle of a homology class that does not contain σ. Again, none of the δ^{k+1}_j can contain σ, because σ does not have any cofaces in the α-complex. Also, only row i^* of H^k contains σ so this row cannot be present in the sum defining c. Therefore, for all $i \in I$ we find that $H^k_i = \widehat{H^k_i}$ and c is still represented in the new matrix $\widehat{H^k}$. □

So in total we have proven six different parts of the algorithms.

1. If a k-simplex gets added, we correctly decide whether it creates a k-homology class or deletes a $(k-1)$-homology class.
2. If a k-simplex gets added and this creates a k-homology class, we correctly find a representative cycle of this class and add it to H^k.

3. If a k-simplex gets added and this removes a $(k-1)$-homology class, we correctly determine which representative cycle to remove from H^{k-1}.
4. If a k-simplex gets removed, we correctly decide whether it deletes a k-homology class or creates a $(k-1)$-homology class.
5. If a k-simplex gets removed and this removes a k-homology class, we correctly change H^k such that the correct homology classes are removed.
6. If a k-simplex gets removed and this creates a $(k-1)$-homology class, we correctly add a representative cycle to H^{k-1}.

4.3 Quality Analysis

If we combine the algorithms from Chapter 3 with the algorithms from this chapter, we get a kinetic data structure that maintains the homology of the union of balls. We will call this the *combined data structure*. To evaluate the quality of the combined data structure, we address its efficiency, responsiveness, compactness and locality.

4.3.1 Efficiency

The events of the combined data structure coincide with those of the kinetic data structure that maintains the α-complex. However, not every event that changes the topological structure of the α-complex also changes the homology of the α-complex so fewer events are classified as external events. For example, a flip event where all simplices involved are short does not create, destroy or change any homology groups. Another example is a radius event in which a k-simplex and its $(k-1)$-face are both added to the α-complex at the same time. The $(k-1)$-simplex creates a $(k-1)$-homology class that is immediately deleted by the k-simplex. So in the end, none of the homology groups have changed. Naturally, the deletion of a k-simplex together with one of its faces would yield the same result.

To find the efficiency of the combined data structure we need to know how many internal events take place. The number of internal events equals the number of events, minus the number of external events. We have seen that the number of events $\xi = \mathcal{O}\left(\xi^f + n_d\right)$. For a kinetic Delaunay complex in d dimensions we know that the worst case number of flip events is upper bounded by $\mathcal{O}\left(n^d \lambda_\delta(n)\right)$ where $\lambda_\delta(n)$ is the maximum length of a (n, δ)-Davenport-Schinzel sequence [3]. We also know that the worst case number of flip events is lower bounded by $\Omega\left(n^{\lceil d/2 \rceil + 1}\right)$ [3]. To the best of our knowledge, these two bounds have only been improved for $d = 2$ dimensions, where the upper bound is known to be $\mathcal{O}\left(n^{2+\varepsilon}\right)$ for any $\varepsilon > 0$ and assuming that the vertices move with unit speed along straight line trajectories [34].

For the worst case number of external events we note that it is naturally bounded by the worst case number of events, which is $\mathcal{O}\left(n^d \lambda_\delta(n)\right)$. For a lower bound on the worst case number of external events, we construct an example where $\Omega\left(n^{\lceil d/2 \rceil}\right)$ homology changes occur. One can construct a Delaunay complex that contains $\Omega\left(n^{\lceil d/2 \rceil}\right)$ simplices [28]. After we do this, we choose α such that all d-simplices are short. We choose the trajectories of the vertices such that the entire data set expands, but the relative positions remain the same. If we would scale the situation, this basically comes down to decreasing α while the point set remains static. This can be done in such a way that all simplices get removed from the α-complex one by one. Because the deletions happen one at a time, each simplex deletion causes a change in the homology groups so the number of homology changes (and the number of external events) is $\Omega\left(n^{\lceil d/2 \rceil}\right)$.

So in total, the number of internal events is upper bounded by $\mathcal{O}\left(n^d \lambda_\delta(n)\right)$ and the worst case number of external events is lower bounded by $\Omega\left(n^{\lceil d/2 \rceil}\right)$. This means that the ratio between the two is upper bounded by $\mathcal{O}\left(n^{\lfloor d/2 \rfloor} \lambda_\delta(n)\right)$. However, we believe this bound is very pessimistic and better bounds can be obtained. Only in $d = 2$ dimensions we know that the ratio is upper bounded by $\mathcal{O}\left(n^\varepsilon\right)$.

4.3.2 Responsiveness

In a flip event or a radius event there can be a constant number of deletions and a constant number of additions of simplices to the α-complex. We will first analyze the running time of adding a simplex to the α-complex and then we analyze the running time of deleting a simplex. When a k-simplex σ gets added, we add a column to δ^k, a row to δ^{k+1} and we either add a row to H^k or we delete a row from H^{k-1}. The additions to δ^k and δ^{k+1} take constant time, because these matrices are stored in sparse format. The addition to H^k or deletion from H^{k-1} takes time proportional to the size of the homology class involved. To see if $\partial_k(\sigma)$ is a linear combination of existing columns of δ^k we solve a linear system of equations $\delta^k h = \partial_k(\sigma)$. The entries of δ^k are elements of $\mathbb{Z}/2\mathbb{Z}$ and δ^k is sparse, so this linear system can be solved in $\mathcal{O}\left(m_k^2 \log^a(n)\right)$ time, where $m_k = \max(n_k, n_{k-1})$ and a is a small constant integer [36]. We have seen that $n_{k-1} = \mathcal{O}(n_k)$ so this system can be solved in $\mathcal{O}\left(n_k^2 \log^a(n)\right)$ time

Now, there are two options. If the linear system has a solution, this solution is added to H^k and the algorithm terminates. If the linear system has no solution, we solve another linear system $\left[\delta^k, \quad \left(H^{k-1}\right)^T\right] h = \partial_k(\sigma)$ where the $[,]$ notation means concatenation of two matrices. This linear system can be solved in $\mathcal{O}\left(m'_k(m'_k + \beta_{k-1}n_{k-1}) \log^a(n)\right)$ time, where $m'_k = \max(n_k + \beta_{k-1}, n_{k-1})$ and a is a small constant integer [36]. This is roughly in $\mathcal{O}\left(\beta n_d^2 \log^a(n)\right)$ time, where β denotes the maximum Betti number.

We know that n_k may be in the order of $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ or be expected to be in the order of $\mathcal{O}(n)$ if S is distributed uniformly. In both cases, the running time of handling the addition of a simplex is at least quadratic in n. However, an algorithm that can handle addition of a simplex faster than quadratic in the number of simplices would allow us to construct an incremental algorithm for computing homology faster than in time cubic in the number of simplices. The standard algorithm for computing the k-homology of a general point set in d dimensions takes $\mathcal{O}\left(n_{k-1}n_k \min(n_k, n_{k-1})\right)$ time which is approximately cubic in the number of simplices. To the best of our knowledge, this standard algorithm has not yet been improved to anything faster than cubic [19]. Therefore, handling addition of a simplex faster than in time quadratic in the number of simplices would be major breakthrough.

If we delete a k-simplex σ, we delete a row in δ^{k+1} and a column in δ^k. Both matrices are stored in sparse format and the row and column both contain a constant number of nonzero entries, so these operation can be executed in constant time. We then check if there is a nonzero entry in the column representing σ in H^k. There are β_k entries to check, so this will take $\mathcal{O}(\beta_k)$ time. Now, there are two options. If there are no nonzero entries in this column, we add $\partial_k(\sigma)$ to H^{k-1}. This takes constant time because H^{k-1} is stored in sparse format and $\partial_k(\sigma)$ contains $k+1 = \mathcal{O}(1)$ nonzero entries. We delete the column representing σ in constant time. If there are one or more nonzero entries in the column representing σ, we choose a row and subtract it from the other rows. In the worst case, we need to do $\beta_k - 1$ row subtractions and each row subtraction takes $\mathcal{O}(n_k)$ operations. Therefore, this case takes $\mathcal{O}(\beta_k n_k)$ time. To delete a row and a column from H^k takes time proportional to the number of simplices in the chosen homology class. The number of simplices in the chosen homology class is $\mathcal{O}(n_k)$. Handling this last case may be improved by choosing the optimal row from H^k. If we choose the row with the least number of nonzero entries, the number of operations for each row subtraction will be minimized, thus reducing the running time.

We recall that a radius event takes $\mathcal{O}(\rho \log n)$ time to maintain the α-complex and a flip event takes $\mathcal{O}(\rho + \log n)$ time. In both types of event we may add or delete a constant number of simplices from the α-complex. Each addition will take $\mathcal{O}\left(m'_k(m'_k + \beta_{k-1}n_{k-1}) \log^a(n)\right)$ time and each deletion takes $\mathcal{O}(\beta_k n_k)$ time. Because $n_k = \mathcal{O}(n_d)$ we find that addition takes $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time and deletion takes $\mathcal{O}(\beta n_d)$ time, where $\beta = \max_{0 \leq k < d} \beta_k$.

4.3.3 Compactness

The combined data structure stores the same objects the data structure for maintaining the α-complex does, only it also stores the δ^k and the H^k. Each δ^k contains exactly $(k+1)n_k$ nonzero

entries, so storing δ^k takes $\mathcal{O}(n_k)$ space. This means that we can store all the boundary matrices in space proportional to the number of simplices in the α-complex. Each k-homology matrix can be stored in $\mathcal{O}(\beta_k n_k)$ space so all homology matrices together can be stored in $\mathcal{O}(\beta n_d)$.

4.3.4 Locality

Locality is about the number of certificates associated with a single vertex. Since there are no extra certificates in the combined data structure with respect to maintaining the α-complex, the locality also does not change. We conclude that the locality is of the same order as the maximal degree of a vertex. In the worst case, the maximal degree of a vertex is $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ but under some uniformity assumptions, the expected maximum degree is $\mathcal{O}\left(\log^2 n\right)$.

If we combine the results of the four quality aspects, we obtain the following theorem.

Theorem 3. *Let $\alpha > 0$ and a set S of n moving points $x_1, \ldots, x_n$ be given, where the movement of the x_i is modeled by coordinate functions which are polynomials of degree δ. Then, the homology of the α-complex of S can be maintained with an $\mathcal{O}\left(n^{\lfloor d/2 \rfloor}\lambda_\delta(n)\right)$ ratio between the internal and external events, $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ update time, $\mathcal{O}(\beta n_d)$ storage space and $\mathcal{O}(\rho)$ certificates per vertex where $\lambda_\delta(n)$ denotes the maximum length of a (n, δ)-Davenport-Schinzel sequence.*

What stands out in this result are the bound on the ratio between the internal and external events and the update time of the combined kinetic data structure. The bound on the ratio between the internal and external events may seem very poor. However, we are unsure if the bound on the worst case number of internal events is sharp, since there is still a gap between the worst case number of events achieved in a simulation and this upper bound. Also, we are unsure about the sharpness of the worst case number of external events, as there may possibly exist simulations with asymptotically more external events. It is possible these bounds can be improved and if that happens, the bound on the ratio will also improve. The combined kinetic data structure is far from responsive because the update time is $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$. However, a dynamic data structure that maintains the homology of a changing simplicial complex in $o\left(n_d^2\right)$ time per update would result in a data structure that can compute the homology of a static simplicial complex incrementally in $o\left(n_d^3\right)$ time, which would be a major breakthrough. The space required to store the data structure may seem poor when expressed in n but it makes intuitive sense to need this much space since the α-complex simply has this many simplices. The locality is still $\mathcal{O}(\rho)$.

Chapter 5

Optimization of low (co-)dimension

The 0-homology group and the $(d-1)$-homology group can be maintained using faster algorithms than the deletion and insertion algorithms presented in Chapter 4. For the 0-homology group we will use that the homology classes are essentially the connected components of the α-complex. For the $(d-1)$-homology group we will use that each class can be seen as the absence of some d-simplices, while their boundary is present.

5.1 Maintaining the Connected Components

We note that the 0-homology classes are essentially the connected components of the α-complex. Also, the connected components of the α-complex equal the connected components of the 1-skeleton of the α-complex. So to know the 0-homology classes of the α-complex, it suffices to maintain the connected components of the 1-skeleton of the α-complex, which is a graph with vertex set S. Maintaining the connected components of a fully dynamic graph can be done in $\mathcal{O}\left(\log^2 n\right)$ amortized time per insertion or deletion of an edge by maintaining a so-called *spanning forest*. We use the data structure presented in [25] to maintain the spanning forest on the 1-skeleton of the α-complex. This data structure can be stored using $\mathcal{O}\left(n_1 + n \log n\right)$ space and it can be updated using $\mathcal{O}\left(\log^2 n\right)$ amortized time. This greatly decreases the costs of maintaining the connected components. Each connected component has one representative vertex, called the *root* of that component. For each vertex in the component, a path to the root through edges and vertices in the component is easily accessible.

When an edge is added to the α-complex, we maintain the spanning forest in $\mathcal{O}\left(\log^2 n\right)$ amortized time and that is all we need to do to maintain the connected components. As we have seen before, the addition of an edge either increases the number of 1-homology classes or decreases the number of connected components. Therefore, if the number of connected components decreases because of the addition of an edge, nothing happens to the 1-homology matrix and we are done with handling this addition. If the number of connected components did not change, a new cycle needs to be added to the 1-homology matrix. In this case, the two vertices that are connected by the new edge were already connected via another route. The path from the first end of the new edge to the root of the component, together with the path from the other end of the new edge to the root, together with the new edge must form a cycle that was not represented in the 1-homology matrix yet. We concatenate the two paths and add the new edge to obtain a homology class h that can readily be added to the 1-homology matrix. Finding these paths takes time proportional to the number of edges in the path. Adding h to the 1-homology matrix takes time proportional to the size of h since H^1 is stored in sparse format. In either case we update the 2-boundary matrix, which can be done in $\mathcal{O}\left(1\right)$ time since this matrix is stored in sparse format.

When an edge is deleted from the α-complex we maintain the spanning forest accordingly. This also takes $\mathcal{O}\left(\log^2 n\right)$ amortized time. Then, we again distinguish between two cases. If the number of connected components increases, nothing happens to the 1-homology matrix and we do not need to update it. If the number of connected components does not increase, the number of 1-homology classes must decrease. This means there is a cycle that contains the deleted edge. As we have seen before, the column representing that edge in H^1 must contain at least one nonzero entry. We find that nonzero entry and execute the algorithms presented in Chapter 4. In either case we update the 2-boundary matrix, which again takes constant time.

We summarize the results in the following theorem.

Theorem 4. *Let C be a dynamically changing simplicial complex. We can maintain the homology groups of C by storing $H^1, \ldots, H^{d-1}$, $\delta^2, \ldots, \delta^d$ and the data structure presented in [25] with vertex set S and the edges of C as edge set. The addition or deletion of an edge may be handled in $\mathcal{O}\left(log^2 n\right)$ amortized time when the number of connected components of C changes and the data structure can be stored using $\mathcal{O}\left(n_1 + n \log n\right)$ space.*

We find that only in specific cases, this other data structure saves us time. However, compared to the running time of the standard dynamical data structure, the space needed to store this data structure and the time needed to update this data structure are very cheap.

5.2 Maintaining the $(d-1)$-Homology Group

To maintain the $(d-1)$-homology group we introduce the notion of the *dual graph*. We represent each d-simplex of D^* by a vertex and whenever two d-simplices share a $(d-1)$-simplex, we connect the two corresponding dual vertices by an edge. Since $(d-1)$-simplices always connect two d-simplices of D^*, each dual edge corresponds to a unique $(d-1)$-simplex. Now, whenever a $(d-1)$-simplex or d-simplex is present in the α-complex, we remove its corresponding dual edge or vertex from the dual graph so that only dual vertices and edges of simplices not in the α-complex remain. We maintain the connected components of this graph using the same data structure as for the connected components of the α-complex itself. We maintain a spanning forest on the dual graph of the α-complex. We call this spanning forest the *dual spanning forest*. Whenever a d-simplex gets added to the α-complex, its dual vertex is deleted from the dual graph. If a d-simplex gets removed from the α-complex, its dual vertex is added to the dual graph. Similarly, the addition and deletion of a $(d-1)$-simplex to the α-complex is handled by deleting and adding the dual edge from the dual graph respectively. Using the data structure from [25], we can maintain the dual spanning forest during these additions and deletions in $\mathcal{O}\left(\log^2 n_d\right)$ amortized time.

Algorithms The algorithms presented in this section replace the algorithms presented in Chapter 4 for the d-simplices and the $(d-1)$-simplices. Therefore, we will only explain how the addition and deletion of d-simplices and $(d-1)$-simplices is handled. Whenever a d-simplex gets added or removed from the α-complex, we only update the dual graph and maintain the dual spanning forest. When a $(d-1)$-simplex gets added to the α-complex, we delete its dual edge from the dual graph and maintain the dual spanning forest. We know that the addition of a $(d-1)$-simplex either creates a $(d-1)$-homology class or deletes a $(d-2)$-homology class. If the number of connected components of the dual graph increases, a $(d-1)$-homology class is created and we are done. If the number of connected components of the dual graph remains the same, a $(d-2)$-homology class is deleted. This means we need to delete a row of H^{d-2}. To do this, we execute the standard algorithm for addition of a $(d-1)$-simplex that deletes a $(d-2)$-homology class, as presented in Chapter 4. When a $(d-1)$-simplex gets deleted from the α-complex, we add its dual edge to the dual graph and maintain the dual spanning forest. Similarly to addition, the deletion of this simplex either deletes a $(d-1)$-homology class or it creates a $(d-2)$-homology class. If the number of connected components of the dual graph decreases, a $(d-1)$-homology class is deleted and we are done. We do not have to do anything for that. If the number of connected components of the

dual graph remains the same, a $(d-2)$-homology class is created. Again, we handle this last case as presented in Chapter 4.

Correctness Proof The difference between a $(d-1)$-cycle and a cycle of any other dimension is that with $(d-1)$-cycles, there is a sense of being 'inside' this cycle. For example, in a 2D α-complex, a 1-cycle is a closed curve. In such a case, it is clear what is inside of this curve and what is outside. In three dimensions we can imagine a cavity in a set of triangles and tetrahedrons and it is immediately clear when you are inside the cavity, and when you are not. Also in higher dimensions there is a notion of inside and outside of a $(d-1)$-cycle. We use this notion in the following lemma.

Lemma 7. *A $(d-1)$-cycle c can only be the boundary of a d-chain that is completely inside c.*

Proof. It is clear that the d-chain c^* consisting of all d-simplices inside c satisfies $\partial_d(c^*) = c$. Now, assume another d-chain $c' \neq c^*$ exists, such that $\partial_d(c') = c$. We find that $\partial_d(c' + c^*) = 0$ so $c^* + c'$ is a d-cycle. Also, no $(d+1)$-simplices exist, so this d-cycle must be a representative cycle of a d-homology class. However, a d-homology class cannot exist in $\mathbb{R}^d$ so this is a contradiction. □

We may now use this lemma to prove the following lemma.

Lemma 8. *Let s be the set of vertices of a connected component of the dual graph that does not contain a vertex that is dual to a d-simplex that contains x_∞. Let c be the d-chain dual to s. Then $\partial_d(c)$ is a representative cycle of a $(d-1)$-homology class.*

Proof. We show that $\partial_d(c)$ is present in the α-complex, that it is a $(d-1)$-cycle and that it is not the boundary of a d-chain that is present in K_α. We know that the dual edges corresponding to $\partial_d(c)$ are not in the dual graph, because that would connect s to other dual vertices. Therefore, we know they must be in the α-complex. We know that $\partial_d(c)$ is a cycle by Lemma 3. By Lemma 7, $\partial_d(c)$ cannot be the boundary of a d-chain outside of $\partial_d(c)$. Also, $\partial_d(c)$ cannot be the boundary of a d-chain inside $\partial_d(c)$ because c is the inside of $\partial_d(c)$ and since s is present in the dual graph, c cannot be present in the α-complex. Therefore, $\partial_d(c)$ is a representative cycle of a $(d-1)$-homology class. □

Note that the connected component of the dual graph that contains a vertex dual to a d-simplex that contains x_∞ always exists because none of the d-simplices that contain x_∞ are present in the α-complex so their dual vertices are present in the dual graph. Also, there can be only one connected component that contains a vertex dual to a d-simplex that contains x_∞ because all d-simplices that contain x_∞ are incident in D^*. We conclude from Lemma 8 that we may find a basis for the $(d-1)$-homology group by considering the boundaries of the connected components of the dual graph.

Quality Analysis Maintaining the dual spanning forest takes $\mathcal{O}\left(\log^2 n_d\right)$ amortized time for each addition and deletion of a d-simplex or $(d-1)$-simplex because the dual graph may contain $\mathcal{O}\left(n_d\right)$ vertices and edges. Unfortunately, when adding or deleting a $(d-1)$-simplex, this may not be enough and we also need to update the $(d-2)$-homology matrix. If we add a $(d-1)$-simplex and we need to update the $(d-2)$-homology matrix, we need to solve a linear system which takes $\mathcal{O}\left(\beta n_d^2 \log^a(n)\right)$ time. If we delete a $(d-1)$-simplex σ and we need to update the $(d-2)$-homology matrix, we add $\partial_d(\sigma)$ to H^{d-2} which takes constant time. The storage of the dual graph and the data structure that maintains the dual spanning forest uses $\mathcal{O}\left(n_d \log n_d\right)$ space. However, we do not need to store H^{d-1} or δ^d so $\mathcal{O}\left(n_d^2\right)$ space is saved.

We summarize the results in the following theorem.

Theorem 5. *Let K_α be a dynamically changing α-complex and D^* a dynamically changing augmented Delaunay complex defined on vertex set S. We can maintain the homology groups of K_α by storing $H^1, \ldots, H^{d-2}$, $\delta^2, \ldots, \delta^{d-1}$, the data structure presented in [25] with vertex set S and the edges of K_α as edge set and the data structure presented in [25] with the d-simplices of D^* minus*

the d-simplices of K_α as vertex set and the $(d-1)$-simplices of D^* minus the $(d-1)$-simplices of K_α as edge set. The addition or deletion of a d-simplex may be handled in $\mathcal{O}\left(\log^2 n\right)$ amortized time, the addition or deletion of a $(d-1)$-simplex may be handled in $\mathcal{O}\left(\log^2 n\right)$ amortized time if the $(d-1)$-homology group does not change and the data structure may be stored in $\mathcal{O}\left(n_d \log n_d\right)$ space.

We find that this alternative data structure for the $(d-1)$-homology group only saves us time if a very specific simplex is added or deleted. Also, the space saved by using this data structure does not asymptotically improve the space needed to maintain all the homology groups. However, only considering the $(d-1)$-simplices and d-simplices, it needs much less update time and much less storage space. Also, if we combine this alternative data structure for the $(d-1)$-homology group with the alternative data structure for the connected components, we may save quite a significant amount of time and space. Relatively, this may save us even more time if d is small and a higher percentage of the additions and deletions involves an edge, $(d-1)$-simplex or d-simplex. However, only if $d = 2$, we can handle every addition and deletion in $\mathcal{O}\left(\log^2 n\right)$ amortized time and maintain all the homology groups using little storage space and update time.

Chapter 6

Kernel Optimization

Maintaining the alpha complex is efficient in the sense that we keep track of a number of events that is optimal in the worst case. However, in some cases, the alpha complex might undergo many topological changes while the homology of the alpha complex does not change at all. In Figure 6.1 we see a 2-dimensional example of a cluster of points where the previous algorithms perform too many actions. In this figure, imagine that all the points in the red circles are moving but the points outside of these circles do not move. The connected components and the 1-homology classes do not change as the points in the red circles move. This means we are dealing with a (possibly quadratic) number of topological changes, while the homology of the complex (which is what we are interested in) does not change at all.

To reduce the number of topological changes we need to deal with, we would like to maintain the presence of some sort of 'cluster' instead of maintaining what the alpha complex exactly looks like. We will also need a criterion that the cluster is dense enough everywhere in the cluster so that there cannot exist holes inside. To facilitate this, we will need the following definitions and notation.

We will partition the ambient space into *cells*. A cell is defined as a convex part of the ambient space with nonempty interior. Each cell c will maintain an integer $c.\textsc{number}$ that shows how many vertices are in the interior of c. A *kernel* is a set of cells of which one is marked as the *center cell*. Also, the cells in the kernel need to be such that if all cells contain at least one vertex, then the union of balls around the vertices in the kernel has to be homotopy equivalent to the union of balls around the vertices in the kernel cells except the center cell, and including something covering the center cell. Three example kernels in 2 dimensions can be found in Figure 6.2. A kernel is called *filled* if all cells in the kernel contain at least one vertex. A kernel with center cell c is denoted by K_c. We also define a *reverse kernel* with a center cell c to be the set of cells c' such that c is a cell in $K_{c'}$. We denote the reverse kernel with center cell c as $\bar{K}_c$.

If K_c is filled for a certain center cell c, we say that c is *marked*. Denote by χ_c all the vertices in the interior of c. We abuse this notation and say that $\chi_{C'} = \cup_{c \in C'} \chi_c$ for a set of cells C'. Let C be the set of all cells in the partitioning. Let $C^m \subseteq C$ be the set of all marked cells. Let x_c be a dummy vertex placed somewhere in c. Again, we abuse this notation and say that $x_{C'} = \cup_{c \in C'} x_c$ for a set of cells C'. We call the union of balls around the vertices in S the *normal union of balls* and we call the union of balls around the vertices in $x_{C^m} \cup S \setminus \chi_{C^m}$ the *adjusted union of balls*. The following theorem allows us to use a kernel consisting of hypercubical cells in $\mathbb{R}^d$.

Theorem 6. *Let the partitioning of $\mathbb{R}^d$ consist of d-cubes such that each side of the cell has length l and the intersection of two cells is always a hypercube or the empty set. Let K be a kernel consisting of $(1+2\lceil\sqrt{d}\rceil)$ cells in each direction so that it consists of a total of $(1+2\lceil\sqrt{d}\rceil)^d$ hypercubes. The center cell of K is defined to be the cell in the middle of K. Then, the normal union of balls of radius $\alpha \geq l\sqrt{d}$ is homotopy equivalent to the adjusted union of balls.*

Proof. If there are no filled kernels, the positions of the vertices are the same and the statement becomes trivial. We now prove that the two unions are homotopy equivalent if there is exactly

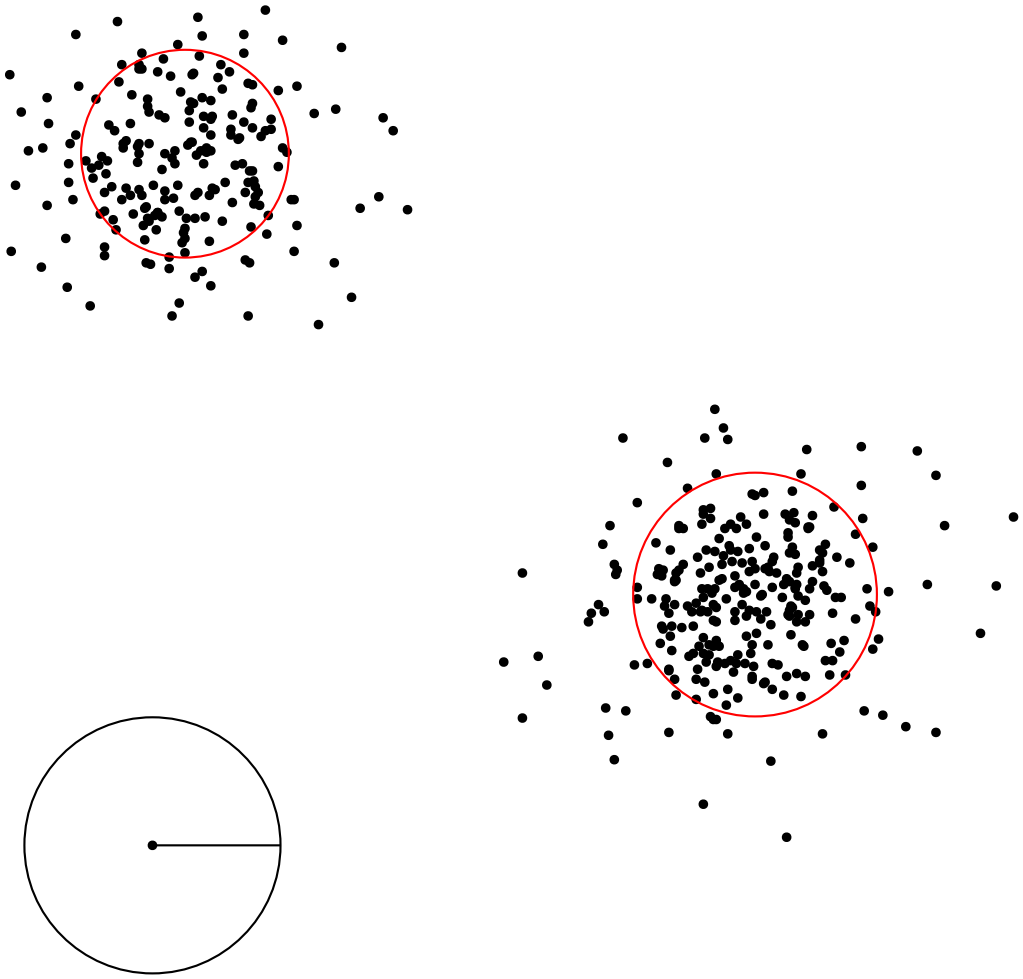


Figure 6.1: Example where there can be many topological changes but the homology of the alpha complex does not change. The circle bottom-left shows the size of α and the points outside of the red circles do not move, while the points inside the red circles do.

one kernel filled. In the case that there are multiple kernels filled, we can apply equivalence to each kernel consecutively.

To prove that the two unions are homotopy equivalent if there is exactly one kernel filled we show that both unions cover each other. First we show that the adjusted union of balls covers the normal union of balls. Each ball from the normal union of balls in the non-center cells are covered by the same ball in the adjusted union of balls. To show that a ball with its center in the center cell c is covered, we partition the space as follows. For ease of writing, let $b = \lceil \sqrt{d} \rceil$. Each coordinate can fall in one of the following ranges.

$$\left(-\infty, -(b+\frac{1}{2})l\right], \quad \left(-(b+\frac{1}{2})l, -(b-\frac{1}{2})l\right], \quad \dots \quad , \left((b-\frac{1}{2})l, (b+\frac{1}{2})l\right], \quad \left((b+\frac{1}{2})l, \infty\right)$$

where the origin is chosen to be the center of c. This means the partitioning contains exactly $(2b+3)^d$ parts. Let $x = (x_1, \dots, x_d)$ be a vertex in the center cell and let $y = (y_1, \dots, y_d) \in B(x, \alpha)$. Let P be the part in the partitioning that contains y.

If P coincides with a cell, let z be the vertex in that cell. Now,

$$\begin{aligned}\|y-z\| &= \sqrt{(y_1-z_1)^2+\cdots+(y_d-z_d)^2}\\ &\leq \sqrt{l^2+\cdots+l^2}\\ &\leq \sqrt{d}l\\ &\leq \alpha\end{aligned}$$

Where $|y_i - z_i| < l$ because both points lie in the same cell. Hence, y is covered by the ball with center z.

If P does not coincide with a cell, let c' be the cell from K_c that shares the highest dimensional face with P. That means that whenever $y_i > (b+\frac{1}{2})l$ or $y_i < -(b+\frac{1}{2})l$, c' will be the cell with its i^{th} coordinate ranging between $(b-\frac{1}{2})l$ and $(b+\frac{1}{2})l$, or between $-(b+\frac{1}{2})l$ and $-(b-\frac{1}{2})l$ respectively. Whenever $-(b+\frac{1}{2})l < y_i < -(b-\frac{1}{2})l$, c's i^{th} coordinate ranges between $-(b+\frac{1}{2})l$ and $-(b-\frac{1}{2})l$ as well, and similarly for the other intervals. Let z be a vertex in c'. For convenience, we sort the coordinates such that $y_1,\ldots,y_m \notin \left(-(b+\frac{1}{2})l,(b+\frac{1}{2})l\right]$ and $y_{m+1},\ldots,y_d \in \left(-(b+\frac{1}{2})l,(b+\frac{1}{2})l\right]$ and without loss of generality we assume that $y_1,\ldots,y_m > 0$. We distinguish several cases with respect to m. If $m = 0$, P coincides with a cell, in which case we have seen that $\|y-z\| < \alpha$. If $m > 0$, we need to do a more rigorous analysis. Assume $m = 1$. During the following computations, we use the following facts. We use that $|y_i - z_i| < l$ for all $m < i \leq d$ and that $0, x_1 \leq \frac{1}{2}l < (b-\frac{1}{2})l \leq z_1 \leq (b+\frac{1}{2})l \leq y_1$.

$$\begin{aligned}\|y-z\|^2 &= (y_1-z_1)^2+(y_2-z_2)^2+\cdots+(y_d-z_d)^2\\ &\leq (y_1-z_1)^2+(d-1)\,l^2\\ &\leq \left(y_1-\left(b-\frac{1}{2}\right)l\right)^2+(d-1)\,l^2\\ &= y_1^2-(2b-1)\,ly_1+\left(b-\frac{1}{2}\right)^2 l^2+(d-1)\,l^2\\ &= y_1^2-ly_1+\frac{1}{4}l^2-\frac{1}{4}l^2-(2b-2)\,ly_1+\left(b-\frac{1}{2}\right)^2 l^2+(d-1)\,l^2\\ &= \left(y_1-\frac{1}{2}l\right)^2-\frac{1}{4}l^2-(2b-2)\,ly_1+\left(b-\frac{1}{2}\right)^2 l^2+(d-1)\,l^2\\ &\leq \left(y_1-\frac{1}{2}l\right)^2-\frac{1}{4}l^2-(2b-2)\left(b+\frac{1}{2}\right)l^2+\left(b-\frac{1}{2}\right)^2 l^2+(d-1)\,l^2\\ &= \left(y_1-\frac{1}{2}l\right)^2-\left(b^2-d\right)l^2\\ &= \left(y_1-\frac{1}{2}l\right)^2-\left(\lceil\sqrt{d}\rceil^2-d\right)l^2\\ &\leq \left(y_1-\frac{1}{2}l\right)^2-\left(\sqrt{d}^2-d\right)l^2\\ &= \left(y_1-\frac{1}{2}l\right)^2\\ &\leq (y_1-x_1)^2\\ &\leq \|y-x\|^2\\ &\leq \alpha^2\end{aligned}$$

So if $m = 1$, y is covered by the ball around z, with radius α.

For $m \geq 2$ we get the following.

$$\begin{aligned}
\|y-z\| &= \sqrt{(y_1-z_1)^2 + \cdots + (y_m-z_m)^2 + (y_{m+1}-z_{m+1})^2 + \cdots + (y_d-z_d)^2} \\
&\leq \sqrt{(y_1-z_1)^2 + \cdots + (y_m-z_m)^2 + l^2 + \cdots + l^2} \\
&= \sqrt{((y_1-x_1)-(z_1-x_1))^2 + \cdots + ((y_m-x_m)-(z_m-x_m))^2 + (d-m)l^2} \\
&\leq \sqrt{((y_1-x_1)-(b-1)\,l)^2 + \cdots + ((y_m-x_m)-(b-1)\,l)^2 + (d-m)l^2}
\end{aligned}$$

Now, since $y_i - x_i \geq bl > (b-1)l$, we have that $((y_i-x_i)-(b-1)\,l)^2 < (y_i-x_i)^2 - (b-1)^2\,l^2$ for all $i \in \{1,\ldots,m\}$. This means we can approximate

$$\begin{aligned}
\|y-z\| &\leq \sqrt{(y_1-x_1)^2 - (b-1)^2\,l^2 + \cdots + (y_m-x_m)^2 - (b-1)^2\,l^2 + (d-m)l^2} \\
&= \sqrt{(y_1-x_1)^2 + \cdots + (y_m-x_m)^2 + \left((d-m) - m\,(b-1)^2\right) l^2}
\end{aligned}$$

Now, if $\left((d-m) - m\,(b-1)^2\right) l^2$ is negative, we know that $\|y-z\|$ is less than $\|x-y\|$ which, in term, is less than α. Notice that $\left((d-m) - m\,(b-1)^2\right) l^2$ decreases in m. This means that if $\left((d-m) - m\,(b-1)^2\right) l^2$ is negative for $m=2$, then it is also negative for any value of $m > 2$. We substitute $m=2$ and obtain

$$\begin{aligned}
\left((d-m) - m\,(b-1)^2\right) l^2 &= \left((d-2) - 2\,(b-1)^2\right) l^2 \\
&= d - 2\left(1 + \left(\lceil\sqrt{d}\rceil - 1\right)^2\right) \\
&\leq d - 2\left(1 + \left(\sqrt{d} - 1\right)^2\right) \\
&= -\left(\sqrt{d} - 2\right)^2 \\
&\leq 0
\end{aligned}$$

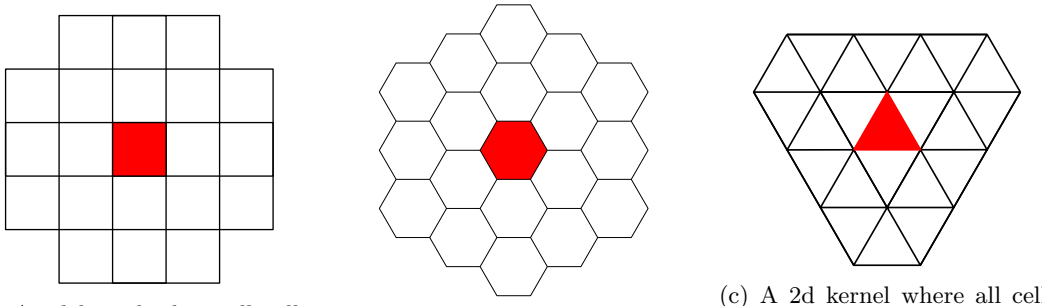
So y is covered by the ball around z with radius α if $m \geq 2$. We already knew y is covered if $m = 0$ or $m = 1$, so we know y is always covered. Since we did not make any assumptions on y we know that the entire ball around x is covered by balls around the vertices in the other cells in the kernel. So the normal union of balls is covered by the adjusted union of balls.

To prove that the adjusted union of balls is covered by the normal union of balls we note that the positions of the balls in the adjusted union of balls is a special case of the position of the balls in the normal union of balls. Therefore, we know that the ball around the center point is covered by the balls around the vertices in the other cells of the kernel. Therefore, the adjusted union of balls is covered by the normal union of balls and the two unions are homotopy equivalent. □

6.1 Cell Certificates

The idea of kernel optimization is to maintain the alpha complex on $x_{C^m} \cup S \setminus \chi_{C^m}$. How to do this is explained using the example kernel from Theorem 6. The initialization of the adjusted union of balls is as follows. Even before the Delaunay complex is computed, the space is partitioned into hypercubical cells with edges of length l, where $l \leq \alpha/\sqrt{d}$. Now, for each vertex $x_i \in S$ we check in which cell it lies. Also, for each cell that contains a vertex, we check how many vertices it contains. Then, for each cell c that contains a vertex, we check whether K_c is filled. If K_c is filled, we mark c. A set T is initialized with $T = S$. For each marked cell c, the vertices in c are deleted

from T and we add a dummy vertex x_c in the center of c to T. So we set $T \leftarrow \{x_c\} \cup T \setminus \chi_c$. An example of these different steps in the euclidean plane can be found in Figure 6.4. A schematic overview of these steps including the maintenance of the α-complex and its homology can be found in Figure 6.3. Now, we can compute the Delaunay complex on vertex set T and execute the standard algorithms using T.



(a) A 2d kernel where all cells are squares.

(b) A 2d kernel where all cells are hexagons

(c) A 2d kernel where all cells are triangles.

Figure 6.2: Three different kernels with their center cells marked red.

To maintain the adjusted union of balls we need a new type of certificate. These new certificates store in which cell each vertex is, so we need d certificates for each vertex $x_i \in S$. These certificates are called *cell certificates*. Recall that the movement of x_i is modelled by its coordinate functions $\big(f_i^1(t), \ldots, f_i^d(t)\big)$. We consider the case where a vertex x_i is in a cell c at a certain time t_{current}. Let the j^{th} coordinates of c range between $(i_j - 1)l$ and $i_j l$. We know that $f_i^j(t_{\text{current}}) > (i_j - 1)l$ and that $f_i^j(t_{\text{current}}) < i_j l$. If $f_i^j(t) = (i_j - 1)l$ or $f_i^j(t) = i_j l$ for some $t > t_{\text{current}}$ then we know that x_i leaves c and the cell certificate fails. So we solve $f_i^j(t) = (i_j - 1)l$ and $f_i^j(t) = i_j l$ for t and add the one that will happen soonest as a *cell event* to the event queue. This is done for each vertex x_i and each dimension j, so there are nd cell events in total in the queue.

When a cell certificate fails, we need to update several data structures. We consider the case where a vertex x_i moves from a cell c_1 to a neighbouring cell c_2 and c_1 and c_2 differ only in the j^{th} coordinate. First of all, we need to update the number of vertices in each cell. So c_1.NUMBER decreases by one and c_2.NUMBER increases by one. We also add a new cell certificate for the j^{th} coordinate function of x_i where we check when $f_i^j(t)$ will cross the j^{th} bounding coordinates of c_2.

To see if there are marked cells that turn unmarked or if there are unmarked cells that turn marked, we need to check whether there are kernels that turn (un)filled. If x_i was the last vertex in c_1, we need to check the reverse kernel $\bar{K_{c_1}}$ of c_1 for cells that turn unmarked. If x_i is the first vertex in c_2, we need to check the reverse kernel $\bar{K_{c_2}}$ of c_2 for cells that should be marked. There are four situations we need to consider separately. If c_1 still contains vertices after x_i leaves and c_2 already contains vertices when x_i enters, no kernels turn (non)filled so no cells need to be (un)marked. If c_1 contains no more vertices after x_i leaves and c_2 already contains vertices when x_i enters, we need to check all cells in the reverse kernel of c_1. If a cell c' in the reverse kernel of c_1 was marked, we remove the mark, remove the dummy vertex $x_{c'}$ of c' from T and add all the vertices $\chi_{c'}$ in c' to T. How to do the insertions and deletion exactly, will be explained later. During these deletions and additions, we maintain the Delaunay complex of T. Because of the nature of a kernel, all newly created d-simplices in the Delaunay complex have to be short so all newly created d-simplices also exist in the alpha complex. We add all appropriate Delaunay certificates and all simplex certificates. Also, the Delaunay certificates that contain $x_{c'}$ are removed with the deletion of $x_{c'}$.

If c_1 still contains vertices after x_i leaves and c_2 was empty before x_i entered, we need to check whether some cells in the reverse kernel of c_2 should be marked. The naive way to do this is to check the kernel of each cell in the reverse kernel of c_2. If we find a cell c with K_c filled, we mark cell c. We then add a dummy vertex x_c to T and delete all the vertices in c from T along with

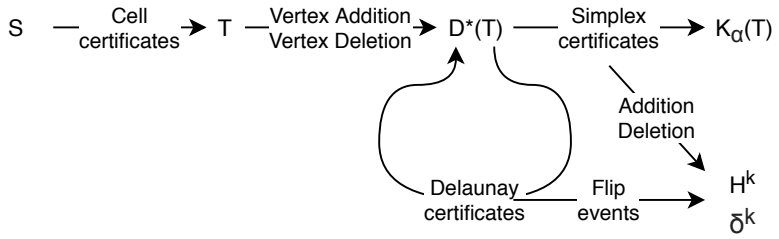


Figure 6.3: The control flow for maintaining the point set T, the α-complex of T and the homology of the α-complex of T.

the Delaunay certificates and simplex certificates that involve these vertices. We maintain the Delaunay complex while doing this and in the end, we add the Delaunay certificates of the newly created $(d-1)$-simplices. Note that we do not need to create any simplex certificates since the d-simplices that contain the dummy vertex in c are necessarily short. We add each newly created d-simplex σ to the *alpha*-complex.

A more efficient way to find the cells that should be marked is to check all cells in the union $\cup_{c' \in \bar{K_{c_2}}} K_{c'}$. We initialize $\Theta = \cup_{c' \in \bar{K_{c_2}}} K_{c'}$ and $\Phi = \bar{K_{c_2}}$. Now, whenever $c.\text{NUMBER} = 0$ for $c \in \Theta$ we set $\Phi \leftarrow \Phi \setminus \bar{K_c}$. After we checked all cells in Θ, we mark all remaining cells in Φ and execute the corresponding steps.

If c_1 contains no more vertices after x_i leaves and c_2 was empty before x_i entered, we check the reverse kernel of c_1 without the reverse kernel of c_2 for marked cells that turn unmarked and we check the reverse kernel of c_2 without the reverse kernel of c_1 for cells that should become marked. Naturally, the handling of the certificates and the vertices is similar to the handling in the two previous cases.

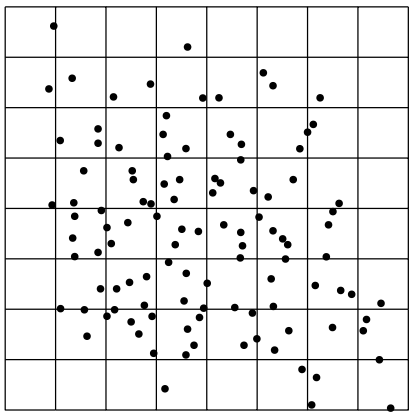
Lastly, we might need to add or remove x_i from the alpha complex. If c_1 was marked before x_i left and c_2 is marked after x_i enters, we do not need to add x_i to T. If c_1 was marked before x_i left and c_2 is not marked after x_i enters, x_i should be added to T and we maintain the Delaunay complex and the Delaunay certificates and simplex certificates appropriately. If c_1 was not marked before x_i left and c_2 is marked after x_i enters, x_i should be removed from T and we maintain the Delaunay complex and the Delaunay certificates and simplex certificates appropriately. If c_1 was not marked before x_i left and c_2 is not marked after x_i enters, we do not need to remove x_i from T.

6.1.1 Vertex Insertion and Deletion

The insertion and deletion of vertices from the Delaunay complex requires some extra attention because it is a nontrivial task. To delete a vertex from the Delaunay complex we make use of an existing algorithm. The idea behind this algorithm is to maintain a priority queue of 'ears' based on a property called the 'power' of that ear [14]. The ear with lowest power is certain to be a simplex in the Delaunay complex so it is added to the Delaunay complex. Afterwards, the existing ears are updated and their position in the priority queue is recomputed. This is repeated until all ears are destroyed and a single simplex remains, which is also added to the Delaunay complex. For more details, we refer the reader to [14].

The insertion of a vertex is slightly more complicated than the deletion of a vertex. When a vertex is deleted, we already have information about its neighborhood because we know which vertices are incident to it. When a vertex is inserted, we generally do not know what its neighborhood is like and which other vertices are closest to it.

To handle the addition of a vertex x_i to T, we use an algorithm described by Bowyer et al. [9]. This algorithm consists of two important steps. The first step is point location: we need to determine which d-simplex in the Delaunay complex has the new point x_i in its interior. This is implemented by performing a search walk through the simplices of the Delaunay complex. If no



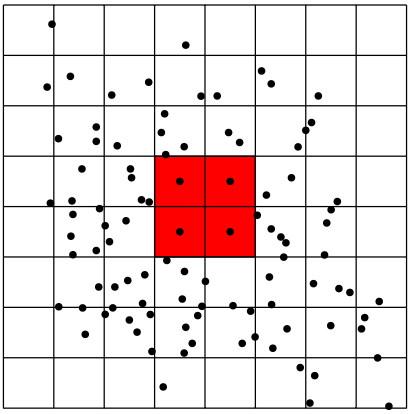
(a) The vertices in a partitioned ambient space

1	0	0	1	0	0	0	0
1	1	2	1	1	2	1	0
0	3	1	4	2	1	2	0
1	2	4	3	5	2	1	0
0	5	3	4	4	5	3	0
0	2	4	4	2	2	3	1
0	2	6	4	3	2	1	2
0	0	0	1	0	1	2	2

(b) The number of vertices in each cell

1	0	0	1	0	0	0	0
1	1	2	1	1	2	1	0
0	3	1	4	2	1	2	0
1	2	4	3	5	2	1	0
0	5	3	4	4	5	3	0
0	2	4	4	2	2	3	1
0	2	6	4	3	2	1	2
0	0	0	1	0	1	2	2

(c) The cells which have a filled kernel are marked red.



(d) The vertices in the marked cells are replaced by dummy vertices.

Figure 6.4: The different steps from the normal union of balls to the adjusted union of balls.

further information or preprocessing is used, this step may take $\Omega\left(n^{1/d}\right)$ expected time. However, in our case, we do have information about the neighborhood of x_i, namely the cell into which it is inserted. We use this information by starting the search walk at another vertex in the same cell. In most cases, this significantly reduces the number of search steps needed.

The second step is updating the topological structure of the Delaunay complex. This takes time proportional to the degree of the newly inserted vertex. If the point set is distributed uniformly at random, this takes expected constant time [9]. However, even if our original point set S is distributed uniformly at random, the Delaunay complex that is currently known is about point set T which is not distributed uniformly at random. This may have a negative effect on the topological structure update time.

We conclude that we can insert and delete vertices into the augmented Delaunay complex. The deletion of vertices is relatively easy, since we only have to perform a structural update and adequate algorithms exist to do that. The insertion of vertices is more complicated because we don't directly have any information about the surroundings of the new vertex. However, if we use information about the cell into which the new vertex is inserted, the search walk to the surroundings of the new vertex is reduced to a nearly trivial task. Existing algorithms can then be used to do the structural update.

6.2 Kernel Optimization Analysis

To know if this kernel optimization actually saves time and effort or rather costs us, we perform an analysis on how much time it saves us and how much time it costs us. We first analyze the quality of the kernel optimization using the four different quality aspects of a kinetic data structure. We then try to capture how much computing time is saved using this technique. To analyze the quality aspects of the kernel optimization technique, we assume there is a bounding box B, surrounding S at all times. Let the side length of B in the i^{th} dimension be denoted by l_i, so that $B = \prod_{i=1}^{d}[0, l_i]$. We find that $|C| = \prod_{i=1}^{d} l_i/l$.

6.2.1 Efficiency

We note that all cell events are internal events. This means that if the kernel technique does not save us any other events, the efficiency will always get worse. A cell event happens each time a vertex crosses the boundary of a cell. There are l_i/l boundaries to cross in the i^{th} dimension. A vertex can cross each boundary at most δ times, where δ is the maximum degree of the coordinate functions of the vertices. So there are at most $\delta l_i/l$ cell events for the i^{th} dimension for a single vertex. We assume δ to be constant so there can be $\mathcal{O}\left(n\sum_{i=1}^{d} l_i/l\right)$ cell events. Though this bound is linear in n, we note that the constant $\sum_{i=1}^{d} l_i/l$ can be arbitrarily bad and is therefore worth taking into account.

6.2.2 Responsiveness

We again look at the case where a vertex x_i moves from a cell c_1 to another cell c_2 where c_1 and c_2 only differ in their j^{th} coordinate. Updating c_1.NUMBER and c_2.NUMBER takes constant time. To compute the failure time of the new cell certificate we need to solve $f_i^j(t) = a_1$ and $f_i^j(t) = a_2$ for some numbers a_1 and a_2. These computations take constant time and adding the certificates to the event queue takes $\mathcal{O}(\log|Q|)$ time.

Next, there are four cases. If c_1 still contains vertices after x_i leaves and c_2 already contains vertices when x_i enters, we do not check if any cells turn (un-)marked and we are done. This clearly takes constant time. In the other three cases, we check the reverse kernels of c_1 or c_2 for cells that become marked or unmarked. Since we consider the situation as in Theorem 6, the reverse kernel of c_1 is exactly the same as the kernel of c_1. The kernel of c_1 contains $(1+2\lceil\sqrt{d}\rceil)^d$ cells, so that is how many cells we check. We assume d to be constant so finding the cells that

become marked or unmarked can also be done in constant time. Marking or unmarking a cell takes time proportional to the time it takes to delete and add vertices to the Delaunay complex. To know the running time of (un-)marking we use the following lemma.

Lemma 9. *If a kernel K_c becomes (non-)filled and the points are distributed uniformly at random, the expected number of vertices in any of the cells is $\mathbb{E}[c.\text{NUMBER}] = N_{|K_c|} = \mathcal{O}(1)$, where N_i is the i^{th} harmonic number.*

Proof. We first prove the statement for the case that a kernel becomes filled. Because the points are distributed uniformly at random, each point that is in the kernel has equal probability to lie in each of the cells. We consider a process that adds vertex to K_c uniformly at random until l cells are covered by a vertex. Let Y_l be the number of vertices needed to cover $l \leq |K_c|$ of the cells. We are interested in $\mathbb{E}[c.\text{NUMBER}] = \mathbb{E}[Y_{|K_c|}]/|K_c|$. Let X_i be the number of vertices added before the number of covered cells increases from i to $i+1$. We find that $Y_l = \sum_{i=0}^{l-1} X_i$ and because the X_i are independent, we get that $\mathbb{E}[Y_l] = \sum_{i=0}^{l-1} \mathbb{E}[X_i]$. Whenever a vertex gets added to K_c while i cells already contain a vertex, the number of cells that contain a vertex only increases when the new vertex gets added to an empty cell. Since each cell has equal probability, the probability that the new vertex gets added to an empty cell is $\frac{|K_c|-i}{|K_c|}$. This means that the X_i are distributed geometrically with success probability $\frac{|K_c|-i}{|K_c|}$. We find that

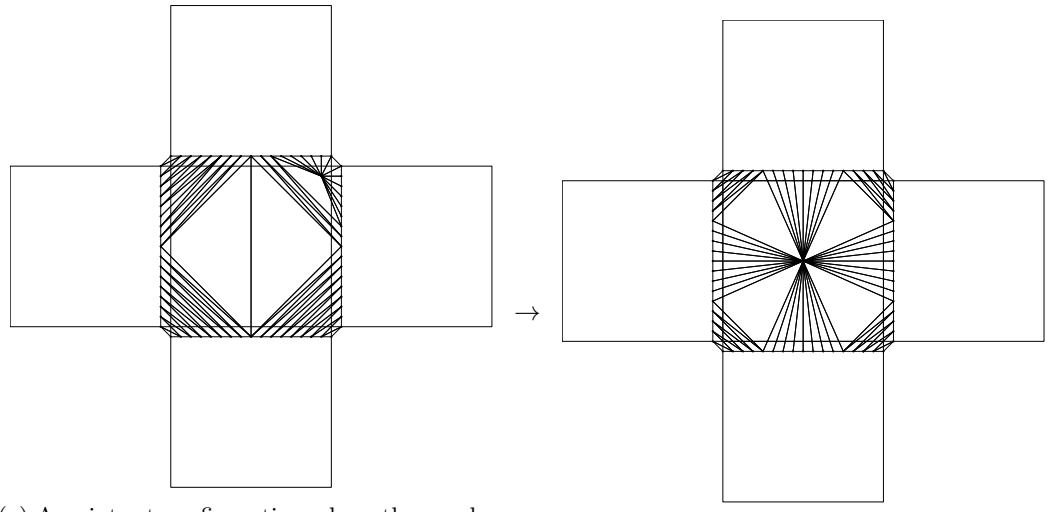
$$\begin{aligned}\mathbb{E}[Y_l] &= \sum_{i=0}^{l-1} \mathbb{E}[X_i] \\ &= \sum_{i=0}^{l-1} \frac{|K_c|}{|K_c|-i}\end{aligned}$$

So we find that

$$\begin{aligned}\mathbb{E}[Y_{|K_c|}] &= \sum_{i=0}^{|K_c|-1} \mathbb{E}[X_i] \\ &= \sum_{i=0}^{|K_c|-1} \frac{|K_c|}{|K_c|-i} \\ &= |K_c| \sum_{i=1}^{|K_c|} \frac{1}{i} \\ &= |K_c| N_{|K_c|}\end{aligned}$$

This means that we expect there to be $|K_c|N_{|K_c|}$ vertices in K_c, so we expect there to be $N_{|K_c|}$ vertices in each cell of K_c. Since $N_{|K_c|} = \mathcal{O}(\log |K_c|)$ [24] and $|K_c| = \left(1 + 2\lceil\sqrt{d}\rceil\right)^d$, we can conclude that $N_{|K_c|} = \mathcal{O}(1)$. The situation where a kernel turns non-filled is analogue. □

This means that (un)-marking a cell takes an expected constant number of deletions and insertions if the points are distributed uniformly at random. We now investigate the running time of adding and deleting vertices to the Delaunay complex. Deleting a vertex takes $\mathcal{O}(f \log f)$ time, where f is the number of d-simplices created [14]. In each cell of the kernel there are an expected constant number of vertices. Therefore, one would intuitively expect that the number of simplices created is also constant, since the number of simplices on n uniformly distributed points is expected to be $\mathcal{O}(n)$. This would mean that deletion of a vertex takes $\mathcal{O}(1)$ time. In the worst case however, the running time may be far worse. For example, if the point distribution is similar to the one depicted in Figure 6.5b, then the number of d-simplices associated with a vertex may be $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$ and the running time of deleting that vertex may be $\mathcal{O}\left(n^{\lceil d/2 \rceil} \log n\right)$.



(a) A point set configuration where the number of search steps from the last remaining vertex to the simplex that has the center in its interior may be large.

(b) A point set configuration where the dummy vertex in the center of a cell may have arbitrarily high degree.

Figure 6.5: The center cell of a kernel before and after the deletion of the last vertex and the insertion of the dummy vertex.

For the running time of inserting a vertex, we recall that inserting a vertex uses two important steps: a search walk to find where the augmented Delaunay complex should be updated and the update of the augmented Delaunay complex itself. We analyze both steps separately. Since there are only a constant number of vertices in the center cell, we intuitively expect the search walk from one vertex in that cell to another vertex in that cell to take only a constant number of steps. In that case, the search walk takes only constant time. Again, in the worst case, the running time may be far worse. If part of the vertices are distributed as in Figure 6.5a and the last vertex that is deleted is in the corner of the cell, we may need a linear number of search steps before we reach the simplex that has the center in its interior. The running time of the update itself is roughly proportional to the number of d-simplices created with the addition [9]. Again, in each cell of the kernel there are an expected constant number of vertices. So as we have seen before, one would intuitively expect that the number of simplices created is also constant, since the number of simplices on n uniformly distributed points is expected to be $\mathcal{O}(n)$. This would mean that the update of the structure of the Delaunay complex takes $\mathcal{O}(1)$ time. However, in the worst case, running times may be far worse, again. Adding the dummy vertex as in Figure 6.5b may take time proportional to the number of d-simplices that contain the dummy vertex, which may be $\mathcal{O}\left(n^{\lceil d/2 \rceil}\right)$. So in total, we strongly expect a cell event to take $\mathcal{O}(\log n)$ time. Only in extreme situations can a cell event take $\mathcal{O}\left(n^{\lceil d/2 \rceil} \log n\right)$ time.

6.2.3 Locality

Each vertex has a constant number of cell certificates, so the kernel optimization data structure is definitely local. However, because vertices are deleted and new vertices are inserted, the known expected bounds do not necessarily hold anymore. This will be discussed in Section 6.2.5.

6.2.4 Compactness

We store a set of cells and pointers from these cells to the vertices and back. This means we need $\mathcal{O}(n)$ space to store the pointers from the vertices and $\mathcal{O}(|C|)$ space to store the cells and their pointers.

We summarize the results in the following theorem.

Theorem 7. *Let B be a bounding box around a simulation of a point set S. We can maintain which cells are marked and which are not using $\mathcal{O}(|C|)$ storage space, $\mathcal{O}\left(n\sum_{i=1}^{d} l_i/l\right)$ events, $\mathcal{O}(\log n)$ expected update time and $\mathcal{O}(1)$ certificates per vertex.*

We find that there may be many events but each event can be handled in little time. Also, the storage of this data structure may be large if B is very large.

6.2.5 Benefits of Kernel Optimization

The benefit of kernel optimization is twofold. First of all, some vertices from the vertex set are not considered, saving us the events these vertices would be involved in. Second of all, the dummy vertices are static, so they cause fewer events. In fact, if there are n points and only $k \leq n$ of these points move, we know that the number of flip events $\xi^f = \mathcal{O}\left(kn^{d-1}\lambda_\delta(n) + (n-k)^d\lambda_\delta(k)\right)$ [3]. The second benefit only yields strong results if the number of dummy vertices is large compared to the number of normal vertices.

For each cell that is marked, we expect a number of vertices to be disregarded. The expected number of vertices in a cell is $n/|C|$ so for each marked cell, we expect to be able to disregard $n/|C|$ vertices. If we assume the points of S are distributed uniformly at random and several cells are marked, then we expect to disregard $|C^m|n/|C|$ vertices. If our simulation is such that n points cause $\Theta\left(n^{\lceil d/2\rceil+1}\right)$ topological events, then after disregarding $|C^m|n/|C|$ vertices, there will be $\Theta\left((n-|C^m|n/|C|)^{\lceil d/2\rceil+1}\right)$ events, thus saving us $\Theta\left((|C^m|/|C|)^{\lceil d/2\rceil+1}\, n^{\lceil d/2\rceil+1}\right)$ events. When we decrease the number of vertices, the number of simplices is also expected to decrease. This means that the handling of the addition of a simplex to the α-complex or the deletion of a simplex from the α-complex can be done faster. Also the storage required to store the homology matrices and the boundary matrices will be smaller. Unfortunately, the maximum degree of a vertex does not necessarily decrease. In fact, many vertices may appear near the boundary of a marked cell and the dummy vertex in that marked cell will be incident to all of them. This may cause very high vertex degrees, resulting in a data structure that is less local.

Chapter 7

Conclusions

The goal of this thesis is to present a kinetic data structure that maintains the homology of the union of balls of radius $\alpha > 0$ around a point set S of n points $x_1, \ldots, x_n \in \mathbb{R}^d$. We have seen that we can maintain the α-complex, which has the same homology as the union of balls, only has much lower complexity. To maintain the α-complex, we need $\mathcal{O}(\rho \log n)$ update time per event, where ρ denotes the maximum degree of a vertex in S. We do not know what exactly is the worst case number of events that may occur. However, we do know that the worst case number of internal events is of the same order of magnitude as the worst case number of external events, so this kinetic data structure is certainly efficient. In the worst case, these bounds are quite poor, since n_d may be in $\Theta\left(n^{\lceil d/2 \rceil}\right)$ and ρ may be linear in n. However, if we assume the trajectories of the points are such that the point set distribution is uniformly at random throughout some convex space at all times, then the expected number of simplices is $\mathcal{O}(n)$ and the expected maximum degree is $\mathcal{O}\left(\log^{2+\varepsilon} n\right)$ for some arbitrarily small $\varepsilon > 0$, thus tremendously improving the bounds.

If we combine the data structure for maintaining the α-complex with a dynamic data structure that maintains the homology of a simplicial complex, then we obtain a kinetic data structure that maintains the homology of a set of moving balls with radius $\alpha > 0$. We know that the total number of events is the same as for the data structure that maintains the α-complex, only we do not know the ratio between the number of events that change the homology of the complex and the number of events that do not change the homology. Therefore, we do not know whether the data structure is efficient or not. The update time increases from $\mathcal{O}(\rho \log n)$ to $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ where a is a small integer. Though this update time implies that the combined kinetic data structure is far from responsive, an improvement to an update time of $o\left(n_d^2\right)$ would imply we can use this data structure to incrementally compute the homology of a simplicial complex in $o\left(n_d^3\right)$ time. Such a running time would improve the best known running time which is $\mathcal{O}\left(n_d^3\right)$ in the worst case.

The bound on the update running time for the combined kinetic data structure comes from the addition of simplices to the α-complex. These additions take $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time, so handling an event takes $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time. Better results can be obtained if we add a d-simplex, $(d-1)$-simplex or edge. If an event causes a d-simplex to be added or removed, this adding and deleting can be done in $\mathcal{O}\left(\log^2 n\right)$ amortized time, instead of in $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time or $\mathcal{O}(n_d \beta)$ time if we use a so-called dual graph. The total time for handling an event is then $\mathcal{O}\left(\rho \log n + \log^2 n\right)$ amortized time, instead of $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time. If an event causes a $(d-1)$-simplex or edge to be added or removed, we may handle this addition or deletion in $\mathcal{O}\left(\log^2 n\right)$ amortized time in some cases and in $\mathcal{O}\left(n_d^2 \beta \log^a n\right)$ time in the other cases. This improvement is most useful if many of the additions and deletions are edges, $(d-1)$-simplices or d-simplices. We expect this is the case if the points live in a low-dimensional space.

The running time of the handling of an event is defined in terms of the number of simplices in the α-complex. We have seen that it is unlikely we find a dynamic algorithm faster than roughly quadratic in the number of simplices. However, it may be possible to reduce the number of simplices and achieve faster results that way. A first attempt to reduce the number of simplices

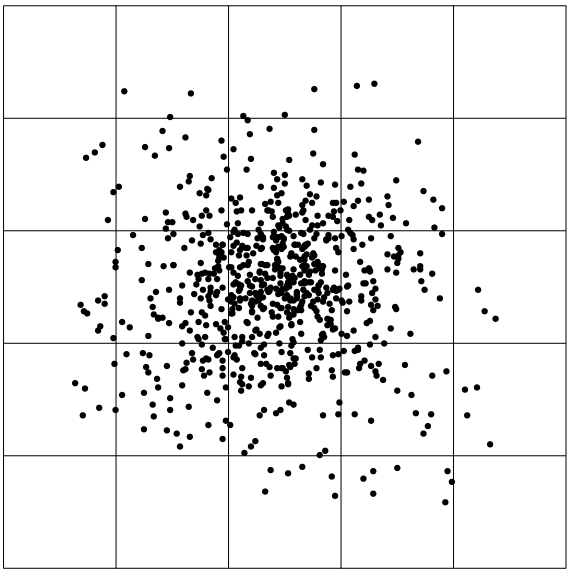


Figure 7.1: A point cloud in a kernel in 2 dimensions where there are many points in the center cell that can be disregarded but the kernel is not filled.

is the kernel optimization technique. The concept of the kernel optimization technique is to keep track of the number of points in cells in order to find dense clusters of points. We know that there are no radius events in such a cluster because all the simplices are necessarily short and therefore, we know that the homology does not change with any of the events in this cluster. We use this by not keeping track of the points in the center cell of such a cluster, thus saving us the handling of internal events. This technique costs us $\mathcal{O}\left(n\sum_{i=1}^{d} l_i/l\right)$ events, which each can be handled in constant time. However, the running time of handling a cell event is $\mathcal{O}\left(\left(1+2\lceil\sqrt{d}\rceil\right)^d\right)$, which grows very rapidly in d and is quickly infeasible in practice. Each vertex in S gets a constant number of extra certificates and we need $\mathcal{O}\left(|C|+n\right)$ space to store the structure, where $|C| = \prod_{i=1}^{d} l_i/l$ is the number of cells. How many events are saved using this technique depends on the number of cells that are marked and for how long these cells are marked. Whether a cell is marked strongly depends on the distribution of the points which is generally not known. To properly determine the number of events saved and time saved, we either need a better theoretical analysis or an empirical analysis, running the simulation with and without the kernel optimization technique and comparing the results. This analysis may be done in future research.

One way to improve the kernel optimization technique is by coming up with better kernels. A kernel is usually better when it contains fewer cells, because then, it is filled more quickly and we may disregard more events. Another way to improve this technique is by scaling the cell size l. The kernel presented in Theorem 6 is proven to work for any $0 < l \leq \alpha/\sqrt{d}$ so we may choose any such l. If we choose l to be too large, then a situation as in Figure 7.1 may occur where there clearly is a cluster with flip events we wish to ignore, but the kernel is not filled because l is too large. We would like to use the fact that the kernel technique also works for smaller l, to handle cases such as the one depicted in Figure 7.1. The smaller the value of l is, the more cell events take place so this will cost us more computation time. However, we may also be able to identify more clusters, saving us more events. Setting l too small will result in no filled kernels at all. Some analysis is still needed to find the optimal value of l. Again, this analysis can be done in a theoretical setting or empirically, using an implementation.

The idea behind the kernel optimization technique is to find clusters and disregard the events in the center of such a cluster. Another way to disregard the events in a cluster would be to use different cluster finding algorithms. However, such a clustering technique will need to be very

simple in order to be profitable. Other ways the homology of moving points may be found more efficiently is by trying to convert another homology algorithm to a dynamic homology algorithm. We do not expect to obtain an algorithm that has much better theoretical running time bounds, but something that is faster in practice may very well exist. Further research could include extending the proposed data structure such that it may also compute persistent homology. In such a case we would be interested in the development of so-called *barcode diagrams* over time. However, this would imply we need to consider two continuous variables, namely time and α. This will most probably cause a large number of events in the event queue. Another extension could be to also store the lifespan of certain homology classes. This is related to persistent homology in the sense that it also concerns the stability of a homology class.

Bibliography

[1] P. K. Agarwal, D. Eppstein, L. J. Guibas, and M. R. Henzinger. Parametric and Kinetic Minimum Spanning Trees. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*, pages 596–605, 1998. 3

[2] P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach. Maintaining the Extent of a Moving Point Set. In F. Dehne, A. Rau-Chaplin, J. Sack, and R. Tamassia, editors, *Algorithms and Data Structures*, pages 31–44, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. 3

[3] G. Albers, L. Guibas, J. Mitchell, and T. Roos. Voronoi Diagrams of Moving Points. *International Journal of Computational Geometry & Applications*, 08:365–379, 1998. 13, 18, 26, 43

[4] P Alexandroff. Über den Allgemeinen Dimensionsbegriff und seine Beziehungen zur Elementaren Geometrischen Anschauung. *Mathematische Annalen*, 98(1):617–635, 1928. 6

[5] J. Basch, L. J. Guibas, and J. Hershberger. Data Structures for Mobile Data. *J. Algorithms*, 31(1):1–28, 1999. 2, 3, 9

[6] J. Basch, L. J. Guibas, and L. Zhang. Proximity Problems on Moving Points. In *proc. 13th Annu. ACM Sympos. Comput. Geom*, pages 344–351, 1997. 3

[7] M. Bern, D. Eppstein, and F Yao. The Expected Extremes in a Delaunay Triangulation. In Albert, Javier Leach and Monien, Burkhard and Artalejo, Mario Rodríguez, editor, *Automata, Languages and Programming*, pages 674–685, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. 20

[8] J. D. Boissonnat, F Chazal, and M Yvinec. *Geometric and Topological Inference*. Cambridge University Press, 2018. Cambridge Texts in Applied Mathematics. 19

[9] A. Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, 24(2):162–166, 1981. 38, 40, 42

[10] N. Broutin, O. Devillers, and R. Hemsley. The Maximum Degree of a Random Delaunay Triangulation in a Smooth Convex. *AofA 2014 - 25th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms*, 2014. 18, 20

[11] K. Buchin, M. Buchin, M. van Kreveld, B. Speckmann, and F. Staals. Trajectory Grouping Structure. In F. Dehne, R. Solis-Oba, and J. Sack, editor, *Algorithms and Data Structures*, pages 219–230, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 3

[12] H. S. M. Coxeter. The Circumradius of the General Simplex. *The Mathematical Gazette*, 15(210):229–231, 1930. 15

[13] C. J. A. Delfinado and H. Edelsbrunner. An Incremental Algorithm for Betti Numbers of Simplicial Complexes on the 3-Sphere. *Computer Aided Geometric Design*, 12:771–784, 1995. 23

[14] O. Devillers. On Deletion in Delaunay Triangulations. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, SCG '99, pages 181–188, New York, NY, USA, 1999. ACM. 38, 41

[15] P. Dłotko, T. Kaczynski, M. Mrozek, and T. Wanner. Coreduction Homology Algorithm for Regular CW-Complexes. *Discrete & Computational Geometry*, 46(2):361–388, 2011. 2

[16] J. G. Dumas, F. Heckenbach, D. Saunders, and V. Welker. Computing Simplicial Homology Based on Efficient Smith Normal Form Algorithms. In M. Joswig and N. Takayama, editors, *Algebra, Geometry and Software Systems*, pages 177–206, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 2

[17] R. A. Dwyer. Higher-Dimensional Voronoi Diagrams in Linear Expected Time. *Discrete & Computational Geometry*, 6(3):343–367, 1991. 18

[18] H. Edelsbrunner. *Geometry and Topology for Mesh Generation.* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2001. 5, 6

[19] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction.* American Mathematical Soc., 2010. 2, 7, 8, 27

[20] J. Friedman. Computing Betti Numbers via Combinatorial Laplacians. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 386–391, New York, NY, USA, 1996. ACM. 2

[21] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, 2004. 3

[22] J. E. Goodman, J. O'Rourke, and C. D. Tóth. *Handbook of Discrete and Computational Geometry.* CRC Press, 2017. 1, 3

[23] L. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic Connectivity for Unit Disks. *Discrete & Computational Geometry*, 25, 2000. 3

[24] B. Guo and F. Qi. Sharp Bounds for Harmonic Numbers. *Applied Mathematics and Computation*, 218:991–995, 2011. 41

[25] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic Deterministic Fully-dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity. *J. ACM*, 48(4):723–760, 2001. 29, 30, 31

[26] T. Kaczyński, M. Mrozek, and M. Ślusarek. Homology Computation by Reduction of Chain Complexes. *Computers & Mathematics with Applications*, 35(4):59 – 70, 1998. 2

[27] M. Kerber and H. Edelsbrunner. 3D Kinetic Alpha Complexes and their Implementation. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 70–77. Society for Industrial and Applied Mathematics, 2012. 3, 7, 14

[28] V. Klee. On the Complexity of d-Dimensional Voronoi Diagrams. *Archiv der Mathematik*, 34(1):75–80, 1980. 26

[29] C. Maria. *Algorithms and Data Structures in Computational Topology.* PhD thesis, Université Nice Sophia Antipolis, 2014. 2

[30] P. McMullen. The Maximum Numbers of Faces of a Convex Polytope. *Mathematika*, 17(2):179–184, 1970. 18

[31] D. P. Mehta and S. Sahni, editors. *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2004. 3

[32] D. P. Mehta and S. Sahni, editors. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2018. 3

[33] J. R. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984. 1

[34] N. Rubin. On Kinetic Delaunay Triangulations: A Near-Quadratic Bound for Unit Speed Motions. *J. ACM*, 62(3):25:1–25:85, 2015. 26

[35] R. Seidel. The Upper Bound Theorem for Polytopes: An Easy Proof of its Asymptotic Version. *Computational Geometry*, 5(2):115 – 116, 1995. 18

[36] D. Wiedemann. Solving Sparse Linear Equations over Finite Fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986. 27

[37] C. K. Yap. Towards exact geometric computation. *Computational Geometry*, 7(1):3 – 23, 1997. 13