2007:124 CIV

MASTER'S THESIS

# Persistent Homology in the Cubical Setting:

## *Theory, implementations and applications*

DANIEL STRÖMBOM

**MASTER OF SCIENCE PROGRAMME**
**Engineering Physics**

Luleå University of Technology
Department of Mathematics

LULEÅ UNIVERSITY OF TECHNOLOGY

2007:124 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 07/124 - - SE

Abstract. The theory of persistent homology is based on simplicial homology. In this thesis we explore the possibility of basing persistent homology on cubical homology. We managed to achive this to some extent and have created a working set of prototype procedures able to calculate the persistent homology of a filtered cubical complex in 2D, and in part 3D, with $\mathbb{Z}_2$ coefficients. We also propose a path that will transform our embryo to a set of procedures capable of handling real applications, in e.g. digital image processing, involving large amounts of data. Extensions to arbitrary finite dimension, orientation, spaces with torsion, PID coefficients and more are also included in the plan for the future.

Sammanfattning. Teorin för persistent homologi är baserad på simplicial homologi. I denna uppsats undersöker vi möjligheten att basera persistent homologi på kubisk homologi. Vi lyckades med detta till viss grad och har skapad en fungerande mängd prototypprocedurer som kan beräkna persistent homologi hos ett filtrerat kubisk komplex i 2D och delvis 3D, med $\mathbb{Z}_2$ koefficienter. Det vi skapat hittills är att betrakta som en prototyp och vi beskriver en serie förändringar och utvecklingar som kommer att ge en mängd procedurer redo för att hantera verkliga tillämpningar, inom exempelvis digital bildanalys, där stora datamängder är involverade. Utökning till godtycklig ändlig dimension, orientering, rum med torsion, PID koefficienter mm finns också i de framtida planerna.

# Contents

# Introduction

Homology in particular and its mother subject algebraic topology in general are versatile tools invented by Poincaré and others to handle complicated global problems. The basic idea of algebraic topology is to use algebraic methods to study topological spaces. Algebraic topology/homological algebra rests on a complicated machinery involving category theory and the reason this approach works is that there are nice functors from the category of topological spaces to the category of abelian groups, and homology theory may be defined as the study of these (homology) functors. However, the main focus of this thesis is rather computational and we refer the reader mainly interested in the theory of algebraic topology or homological algebra to [**Hat01**], [**Spa66**] or [**Lan02**] for introductions and [**CE73**] for the bible of homological algebra. The only material of this kind we provide here are brief overviews of the homology theories of interest to us for computation, e.g. cubical homology and persistent homology based on simplicial homology. The theoretically important singular theory, exact sequences, proof of the invariance of homology is left out and may be found in the mentioned references. It is assumed that the reader have some familiarity with homology but if this is not the case large parts of the thesis should still be of use since the computational parts of the material are fairly intuitive.

The reason homology is well suited for computer calculations, as opposed to e.g. homotopy, is that the former is discrete in nature whereas the latter continuous. The main problem with using computers for homology is that in practice huge amounts of data needs to be processed and calculating homology using the standard reduction algorithm as outlined in textbooks on algebraic topology usually carried out by hand on easy examples in exercises quickly become cumbersome and very slow when applied to slightly more sophisticated situations. However, nowadays more efficient means of calculating homology using computers are available, but still there are huge amounts of data to be processed in typical applications and therefore each improvement is of great value. The hope is of course that the idea we chose to investigate in this thesis will turn out to be such an improvement after some more work. We know that using cubes impose severe restrictions on the type of spaces that can be handled by this method, but some very important ones fit the cubical setting like hand in glove, e.g. image processing, and potential improvements in such areas are more than worth while investigating the matter for.

Now, in recent years a new type of homology, called persistent homology, have been invented and developed by G. Carlsson at Stanford and others. This type of homology is different from classical (static) homology, e.g. singular, simplicial etc., in that it describes a nested sequence of subcomplexes and in this sense a dynamic complex. Efficient algorithms for persistent homology has been developed and applied successfully in, for example, structural biology, surface recovery from point cloud data, pattern recognition, structure in large data sets etc. These algorithms has also been implemented and a collection of procedures for persistent homology calculations in MATLAB called PLEX is freely available. PLEX may be found at http://math.stanford.edu/comptop/programs/plex/ and chapter 2 in this thesis contains an outline of the theory of persistent homology.

The theory of persistent homology is based on simplicial homology. However, there is another static theory that has been shown to have advantages over the simplicial in some important applications and this is cubical homology. Cubical homology has been applied successfully in digital image processing/analysis, study of dynamical systems, structural medicine etc. A free software package called CHOMP for calculating cubical homology has also been developed and it may be downloaded from http://www.math.gatech.edu/ $\sim$ chomp/download/.

In the early stages of this diploma work we came across these theories and to our surprise we could not find anything about the use of cubical complexes in persistent homology, despite the fact that this ought to be more efficient in some relevant situations and also make persistent homology suitable for applications in new areas. Thus we decided to investigate the possibility of basing persistent homology on cubical complexes in this master thesis project. The theory was not much of a hurdle since cubical complexes are constructed as chain complexes so all the homological algebra is still working and in particular each cubical complex can be expressed as a simplicial complex and it is easy to see that every cubical complex satisfies the requirements in the definition of an abstract simplicial complex, as any set of identical geometric figures that can only be fitted together in a "nice" way would. Hence in this sense the simplicial theory contains the cubical. Some small elements of the theory still need special care but the main thing was creating actual implementations able to calculate persistent homology of a filtered cubical complex.

We wanted things in this first attempt to be absolutely transparent and for this reason we chose a geometrically intuitive representation of the cubes despite the fact that far more efficient representations are known and have been used by others. This issue is discussed in detail in chapter 4 and 6.

Now, to calculate persistent homology we need to have a filtered cubical complex, i.e. a nested sequence of subcomplexes of a cubical complex. Thus we first need to have some way of obtaining such a nested sequence or we will not be able to do anything. For some practical applications of persistent homology in the simplicial setting a nice way to generate filtrations is by using so called $\alpha$-shapes [**Ede95**] and [**EM94**]. The idea of $\alpha$-shapes is that we grow (or shrink) a finite collection of balls $S$ in the plane or 3-space by varying the radius $r$ of the balls and then find the weighted Voronoï diagram with respect to the centers of the balls. The Vononoï diagram is not a simplicial complex but its dual $K$, which for each fixed $r$ is a subcomplex of the corresponding Deluaney triangulation, is. Also, if $r_0 \leq r_1$ then $K_0$ is a subcomplex of $K_1$ and hence when we increase the radius $r$ of the balls the collection of dual complexes $\{K_r\}_{r=r_0..r_k}$ is a filtered simplicial complex. The reason to care about this is that $K$ is a deformation retract of the union of balls $\cup S$ as shown in [**Ede95**] and hence $K$ share its basic topology with the union of balls and hence we may study $K$ to obtain information about $\cup S$ and the point is of course that the former is a simplicial complex whereas the latter is not. See the mentioned references for the details. Now, to begin with in the cubical setting we constructed a very simple way of obtaining a filtered cubical complex by successively generate random points in $\mathbb{R}^2$ or $\mathbb{R}^3$ and then set up the cube containing the particular point (and if necessary that cubes' boundaries in reverse order). This provided us with all the toy-filtrations we needed to develop and test our procedures for cubical and persistent homology but for most real applications we are likely to need a filtration that grows like that obtained by $\alpha$-shapes in the simplicial setting. For this reason we created a prototype of a special filter generator called *cgfilter* that is described in chapter 4.

Once we had a means of obtaining filtrations to use as input to persistent homology procedures we began to consider algorithms for cubical and persistent homology. We chose to implement a reduced version of the classical reduction algorithm and the

Edelsbrunner incremental algorithm to find the Betti numbers of a cubical complex in 2D. For persistent homology in 2D with $\mathbb{Z}_2$ coefficients we took the algorithms described by Carlsson and Zomorodian in [**CZ04**] and modified them to take and handle cubical instead of simplicial complexes. Furthermore, to present the persistence data we have constructed procedures that plot barcodes and we also have procedures that calculate both the usual $p$-persistent $k$:th Betti numbers and the square Betti numbers of a filtered complex. The implementations of key algorithms with comments may be found in chapter 4 and the most important support procedures may be found in appendix A.

Luleå University of Technology
March 2007

D. Strömbom
danstr-2@student.ltu.se

CHAPTER 1

# Static Homology

We adopt the term static homology as a label for homology theories which do not involve filtered complexes. The reason for this is to distinguish these theories from persistent homology which, in contrast, may be considered dynamic as we will see in the next chapter. In this thesis we are interested in two particular static homology theories, namely simplicial and cubical. Simplicial homology is more general and as mentioned in the introduction the simplicial theory includes the cubical. However, the cubical setting has some very attractive qualities which in some cases make it more suitable for computations. Also for some applications cubes are very natural to use, in particular in image processing and the like since the pixel and its higher-dimensional analogues are cubes in this sense. The following sections on simplicial and cubical homology are rough summaries. For more complete accounts see e.g. [**Hat01**] for simplicial homology and [**KMM04**] for a full introduction to cubical homology.

## 1. Simplicial Homology

There are several ways to think about a simplicial complex. One way is to define simplices and simpical complexes in a geometric way by saying that a **geometric $k$-simplex** $\sigma$ is the convex hull of $k+1$ affinely independent points $S = \{v_0, v_1, \ldots, v_k\}$ and then go on to define the notion of a of a $k$-simplex $\sigma$ defined by $S$ as follows.
A simplex $\tau$ defined by $T \subset S$ is a **face** of $\sigma$ and we denote this by $\tau \leq \sigma$. Using these notions we may now define a **geometrical simplicial complex** $K$ as a finite set of simplices satisfying

If $\sigma \in K$ and $\tau \leq \sigma$ then $\tau \in K$, and
if $\sigma, \tau \in K$ then $\sigma \cap \tau$ is either empty or a face of both $\sigma$ and $\tau$.

Also, a $k$-simplex $\sigma$ is of dimension $k$ and the dimension of a geometrical simplicial complex $K$ is defined by

$$\dim K = max\{\dim \sigma : \sigma \in K\}.$$

This geometric definition is nice since it allows one to visualize low-dimensional simplices with $k \leq 3$ as points, line segments, triangles and tetrahedrons and also complexes composed of such simplexes. However, what is more important for us is the notion of an *abstract simplicial complex*, since this is a combinatorial object rather than a geometrical one and hence better suited for computation.

Definition 1.1. *A (finite)* ***abstract simplicial complex*** *consists of a finite set $K$ and a collection of subsets $\boldsymbol{S}$ of $K$ called* ***simplices*** *such that the following holds.*

*If $v \in K$ then $\{v\} \in S$.*
*If $\sigma \in S$ and $\tau \subset \sigma$ then $\tau \in S$.*

A simplex $\sigma$ is a $k$-simplex, $\dim \sigma = k$, if $\sigma$ contains $k+1$ elements. In particular this makes $\dim \emptyset = -1$, that is, the empty set is a $(-1)$-simplex. Also, a **subcomplex** of a simplicial complex $K$ is a subset $L \subseteq K$ that is itself a simplicial complex. We also need the notion of an **oriented $k$-simplex** $[\sigma] = (v_0, v_1, \ldots, v_k)$ and we define orientations as

equivalence classes of orderings of the vertices defining the simplex by saying that two orientations are equivalent if they differ by an even permutation. That is, we define an equivalence relation $\sim$ by

$$(v_0, v_1, \ldots, v_k) \sim (v_{\pi(0)}, v_{\pi(1)}, \ldots, v_{\pi(k)})$$

iff $\pi$ is an even permutation. For a complex $K$ of dimension $n$ we may define a free abelian group $\mathcal{C}_k$ on the oriented $k$-simplices of $K$. That is, the oriented $k$-simplices of $K$ constitute a basis for $\mathcal{C}_k$ and we call this group the **$k$:th chain group**, note that for $k > n$ the chain groups will be trivial. The elements of $\mathcal{C}_k$ are called $k$-chains and are of the form

$$\sum_i a_i[\sigma_i],$$

with $[\sigma_i] \in K$ and the $a_i$ are coefficients from some ring $R$, for example $\mathbb{Z}$. The structure of the homology groups is ultimately dependent on $R$ and the additional structure it might carry as we will see later. In addition to the chain groups we need a collection of homomorphisms, called **boundary operators** in order to form a chain complex. The action of the boundary operator

$$\partial_k : \mathcal{C}_k \to \mathcal{C}_{k-1}$$

on a single oriented $k$-simplex $[\sigma] = (v_0, v_1, \ldots, v_k)$ is defined by

$$\partial_k[\sigma] = \sum_{i=0}^{k} (-1)^i [v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k],$$

and this is extended linearly to $k$-chains. That is, for

$$c = \sum_i a_i[\sigma_i] \in \mathcal{C}_k$$

we have that

$$\partial_k c = \sum_i a_i \partial_k[\sigma_i].$$

Now we define a **simplicial chain complex $\mathfrak{C}$** to be a collection of chain groups $\{\mathcal{C}_k\}_{k\geq 0}$ together with a collection of connecting boundary operators $\{\partial_k\}$, that is $\mathfrak{C} = \{\mathcal{C}_k, \partial_k\}_{k\geq 0}$ or in sequence form

$$\ldots \xrightarrow{\partial_{k+2}} \mathcal{C}_{k+1} \xrightarrow{\partial_{k+1}} \mathcal{C}_k \xrightarrow{\partial_k} \mathcal{C}_{k-1} \xrightarrow{\partial_{k-1}} \ldots.$$

A very important property of the boundary operators, that is easily proved from the definition, is that for all $k$ we have that

$$\partial_k \partial_{k+1} = 0.$$

By the previous paragraph we know that given a complex $K$ we can get an associated chain complex $\mathfrak{C} = \{\mathcal{C}_k, \partial_k\}_{k\geq 0}$. Now, using the boundary operators $\partial_k$ we can define two interesting subgroups of $\mathcal{C}_k$. First we define the **cycle group** $\mathcal{Z}_k$ by

$$\mathcal{Z}_k = \ker \partial_k,$$

and the **boundary group** $\mathcal{B}_k$ by

$$\mathcal{B}_k = Im\ \partial_{k+1}.$$

Also recall that $\partial_k \partial_{k+1} = 0$, i.e. the boundary of a boundary is 0 which implies that every boundary (an element of $\mathcal{B}_k$) is also a cycle (a member of $\mathcal{Z}_k$), and hence we have that

$$\mathcal{B}_k \subset \mathcal{Z}_k \subset \mathcal{C}_k.$$

Furthermore $\mathcal{B}_k$ is *normal* in $\mathcal{Z}_k$, since subgroups of abelian groups are normal, and we may thus define the **$k$:th homology group** $H_k$ as the quotient group

$$\mathcal{H}_k = \mathcal{Z}_k/\mathcal{B}_k,$$

for each $k$. As mentioned above the type of coefficients we choose to use determines the structure of the homology groups. If we choose the coefficient ring to be $\mathbb{Z}$ the structure of the homology groups is described by *the structure theorem for finitely generated abelian groups*. Each homology group thus decomposes into a *free part* and a *torsion part* and hence the structure is described fully by the Betti number and the torsion coefficients. If the coefficient ring is a field the torsion part in the decomposition vanishes and the homology groups are vector spaces fully described by their dimension.

## 2. Cubical Homology

Whereas the geometric building blocks of simplicial homology are triangles and their higher dimensional analogues the building blocks of cubical homology are squares, cubes and so on. Furthermore, in simplical homology nothing is said about the (relative) sizes of the simplices but in the cubical theory we decide that each cube should have unit size, e.g. all 1-cubes, i.e. intervals, have length one and all 2-cubes are squares of side length one, all 3-cubes are cubes with side length one and so on for higher dimensional analogues. As is expressed in [**KMM04**] this may at first seem to restrictive, but in applications it is simply a matter of units and scaling. We now define the building blocks of cubical homology as follows. The 1-cubes are called **elementary intervals** and are defined as closed subsets $I \subset \mathbb{R}$ of the form

$$I = [m, m+1], \quad m \in \mathbb{Z}.$$

The 0-cubes, or points, are considered degenerate elementary intervals

$$[m] = [m, m], \quad m \in \mathbb{Z}.$$

Based on these two we now define an $n$-cube, or **elementary cube**, $Q$ to be a finite product of elementary intervals $I_i$ (degenerate or non-degenerate) as

$$Q = I_1 \times I_2 \times \cdots \times I_n \subset \mathbb{R}^n.$$

Furthermore, we denote the set of all elementary cubes in $R^n$ by $\mathcal{K}^n$ and the set of all elementary cubes by

$$\mathcal{K} = \bigcup_{n=0}^{\infty} \mathcal{K}^n.$$

For an elementary cube

$$Q = I_1 \times I_2 \times \cdots \times I_n \subset \mathbb{R}^n$$

we define the **embedding number** of $Q$, *emb* $Q$, to be the dimension of the space of which $Q$ is a subset of, i.e. *emb* $Q = n$ here. Also, the dimension of $Q$, $\dim Q$ is the number of non-degenerate components $I_i$ of $Q$ and the set of all elementary cubes of dimension $k$ is denoted by $\mathcal{K}_k$. Moreover, the set of elementary cubes in $R^n$ of dimension $k$ is

$$\mathcal{K}_k^n = \mathcal{K}^n \cap \mathcal{K}_k.$$

Due to this setup it is clear that the class of topological spaces we can examine using this cubical theory is very limited and in fact we are only interested in so called **cubical sets**, which are subsets of $\mathbb{R}^n$ that can be written as a finite union of elementary cubes. If $X \subset \mathbb{R}^n$ then we denote the set of all elementary cubes in $X$ by $\mathcal{K}(X)$ and all elementary cubes of $X$ of dimension $k$ by $\mathcal{K}_k(X)$ and the elements of the latter are called **the $k$-cubes of $X$**. Elementary cubes are geometric objects in the same sense as that of a geometrical simplex and just as in case with simplices we associate to each elementary

cube $Q \subset \mathcal{K}_k^n$ an abstract object $\hat{Q}$ which we call an **elementary $k$ -chain** of $\mathbb{R}^n$. The two objects are usually identified with each other, but as is mentioned in [**KMM04**] distinguishing between them is useful for determining data structures. Furthermore, the set of all elementary $k$-chains of $\mathbb{R}^n$ is denoted by

$$\hat{\mathcal{K}}_k^n = \{\hat{Q} \colon Q \in \mathcal{K}_k^n\},$$

and the set of all elementary chains of $\mathbb{R}^n$ by

$$\hat{\mathcal{K}}^n = \bigcup_{k=0}^{\infty} \mathcal{K}_k^n.$$

Now, for any finite collection of elementary $k$-chains $\mathcal{B} = \{\hat{Q}_i\}_{1 \leq i \leq d} \subset \mathbb{R}^n$ we may form so called cubical $k$-chains of the form

$$c = \sum_{i=1}^{d} a_i \hat{Q}_i$$

where the $a_i$ are coefficients from the algebraic structure over which the homology is to be calculated, e.g. $\mathbb{Z}$ or $\mathbb{Z}_p$ for $p$ prime. In addition, if all the $a_i$ are 0 then we define $c = 0$. The set of all $k$-chains that may be formed by the elements of $\mathcal{B}$ and coefficients from a ring $R$ is denoted by $\mathcal{C}_k^n(R)$. For two elements $c_1, c_2 \in C_k^n(R)$ their sum is defined by

$$c_1 + c_2 = \sum_{i=1}^{d} a_i \hat{Q}_i + \sum_{i=1}^{d} b_i \hat{Q}_i = \sum_{i=1}^{d} (a_i + b_i) \hat{Q}_i.$$

From now on we will not write out the $R$ explicitly. Thus, under $+$ the set $C_k^n$ is a free abelian group with basis $\mathcal{B}$. Also, since the basis contains a finite number of elementary $k$-chains the group is finitely generated.

Definition 1.2. *Let $X \subset \mathbb{R}^n$ be a cubical set and let*

$$\hat{\mathcal{K}}_k(X) = \{\hat{Q} \colon Q \in \mathcal{K}_k(X)\}.$$

*Then **the set of** $k$ **-chains of** $X$ is the free abelian group $\mathcal{C}_k(X)$ generated by the (finite number of) elements of $\hat{\mathcal{K}}_k(X)$.*

For introducing the cubical boundary operator it is useful to define the following **cubical product** $\star$ for two elementary cubes $Q_1 \in \mathcal{K}_k^n$ and $Q_2 \in \mathcal{K}_k^n$ by

$$\hat{Q}_1 \star \hat{Q}_2 = \widehat{Q_1 \times Q_2}.$$

Also, if we define an inner product of chains in $\mathcal{C}_k^n$, $c_1 = \sum_{i=1}^m a_i Q_i, c_2 = \sum_{i=1}^m b_i Q_i$ by

$$(c_1, c_2) = \sum_{i=1}^{m} a_i b_i,$$

then we may extend the definition of the cubical product $\star$ to chains $c_1 \in \mathcal{C}_k^n$ and $c_2 \in \mathcal{C}_{k'}^{n'}$ by

$$c_1 \star c_2 = \sum_{Q_1 \in \mathcal{K}_k^n, Q_2 \in \mathcal{K}_{k'}^{n'}} (c_1, \hat{Q}_1)(c_2, \hat{Q}_2) \widehat{Q_1 \times Q_2} \in \mathcal{K}_{k+k'}^{n+n'}.$$

It is straight forward to check that the cubical product is *associative*, *distributive over addition of $k$-chains* and *if the product of two chains is* 0 *then at least one of the factors are* 0. In [**KMM04**] p. 52 they state and prove the following useful fact.

THEOREM 1.3. *If $\hat{Q}$ is an elementary cubical chain of $\mathbb{R}^n$ with $n > 1$, then there exists unique elementary cubical chains $\hat{I}$ and $\hat{P}$ with emb $I = 1$ and emb $P = n - 1$ such that*

$$\hat{Q} = \hat{I} \star \hat{P}.$$

DEFINITION 1.4. *For each $k \in \mathbb{Z}$ the cubical boundary operator*

$$\partial_k \colon \mathcal{C}_k^n \to \mathcal{C}_{k-1}^n$$

*is a homorphism of free abelian groups, defined for an elementary chain $\hat{Q} \in \hat{\mathcal{K}}_k^n$ by induction on the embedding number $n$ as follows.*

For $n = 1$ $Q$ is an elementary interval, i.e. $Q = [m]$ or $Q = [m, m+1]$ for some $m \in \mathbb{Z}$, and we define

$$\partial_k \hat{Q} = \begin{cases} 0 & \text{if } Q = [m] \\ \widehat{[m+1]} - \hat{[m]} & \text{if } Q = [m, m+1] \end{cases}.$$

For $n > 1$ let $I = I_1(Q)$ and $P = I_2(Q) \times \cdots I_n(Q)$ so that (by the proposition above)

$$\hat{Q} = \hat{I} \star \hat{P}.$$

Then define

$$\partial_k \hat{Q} = \partial_{\dim I} \hat{I} \star \hat{P} + (-1)^{\dim I} \hat{I} \star \partial_{\dim P} \hat{P}.$$

By linearity this is then extended to chains, i.e. if $c = \sum_{i=1}^p a_i \hat{Q}_i$, then

$$\partial_k c = \sum_{i=1}^{p} a_i \partial_k \hat{Q}_i.$$

As is customary, from now on subscripts in the boundary operator notation is most often left out since this is usually clear from the context. In [**KMM04**] p.56-57 the following theorem and its generalization is proved.

THEOREM 1.5. *If $c_1$ and $c_2$ are cubical chains, then*

$$\partial(c_1 \star c_2) = \partial c_1 \star c_2 + (-1)^{\dim c_1} c_1 \partial c_2.$$

Of course this statement holds for elementary cubical chains as well since they are special cases of cubical chains. Furthermore, this theorem may, by induction, be generalized to any finite number $p$ of cubical chains, elementary or otherwise (since the boundary operator is linear), as

$$\partial(\hat{Q}_1 \star \cdots \star \hat{Q}_p) = \sum_{j=1}^{p} (-1)^{\sum_{i=1}^{j-1} \dim Q_i} \hat{Q}_1 \star \cdots \star \hat{Q}_{j-1} \star \hat{Q}_j \star \hat{Q}_{j+1} \star \cdots \star \hat{Q}_m.$$

As should be expected the following result hold for the cubical boundary operator.

THEOREM 1.6. *The cubical boundary operator satisfies*

$$\partial\partial = 0.$$

The proof is by induction on the embedding number and makes use of theorem 2 and may be found in [**KMM04**] p. 58. Now it can be proved that if $X \subset \mathbb{R}^n$ is a cubical set, then

$$\partial_k(\mathcal{C}_k(X)) \subseteq \mathcal{C}_{k-1}(X),$$

and hence it makes sense to define the boundary operator for the cubical set X, $\partial_k^X$, to be the restriction of $\partial_k \colon \mathcal{C}_k \to \mathcal{C}_{k-1}$ to $\mathcal{C}_k(X)$. Thus we now have chain groups $\mathcal{C}_k(X)$ and boundary operators $\partial_k^X$ so we can define the chain complex of a cubical set as follows.

DEFINITION 1.7. *If $X \subset \mathbb{R}^n$ is a cubical set, then the* ***cubical chain complex*** *for $X$ is*

$$\mathfrak{C}(X) = \{\mathcal{C}_k(X), \partial_k^X\}_{k\in\mathbb{Z}},$$

*where $\mathcal{C}_k(X)$ are the groups of $k$-chains generated by $\mathcal{K}_k(X)$ and $\partial_k^X$ is the cubical boundary operator restricted to $X$.*

For a comparison of simplicial and cubical complexes, especially as objects for computing, see [**KMM04**] p.385-87.

CHAPTER 2

# Persistent Homology

Persistence is a fairly new notion invented and developed by G. Carlsson, H. Edelsbrunner, A. Zomorodian and others during the past few years. Persistence may be described as a measure of the lifetimes of topological attributes within a filtration and the basic premise is that attributes that persists for a long time are significant, whereas attributes with short lifetimes are considered topological 'static' or noise. Mathematically persistence is described through the following. In the classical setting it is assumed that we have a complex, simplicial or cubical, corresponding to the space we wish to find the homology of, whereas for persistent homology we assume that we have a filtered complex . That is, a complex $K$ together with a nested sequence of subcomplexes $\{K^i\}_{0\leq i\leq n}$ such that for each $i \geq 1$ we have that $K^{i-1}$ is a subcomplex of $K^i$ and $K^n = K$, i.e.

$$\emptyset = K^0 \subseteq K^1 \subseteq \cdots \subseteq K^n = K.$$

Assuming that we have a filtered complex $K$ each of the subcomplexes in $\{K^i\}_{0\leq i\leq n}$ will have an associated chain group $\mathcal{C}_k^i$, boundary operator $\partial_k^i$, cycle group $\mathcal{Z}_k^i$, boundary group $\mathcal{B}_k^i$ and homology group $\mathcal{H}_k^i$ for all $i, k \geq 0$ as defined in chapter 1. Now we wish to define homology groups that will capture the topological attributes of the filtered complex that persists for at least $p$ steps in the filtration. Generalizing the classical definition

$$\mathcal{H}_k = \mathcal{Z}_k/\mathcal{B}_k,$$

and factoring out $\mathcal{B}_k^{i+p} \cap \mathcal{Z}_k^i$ (the boundary group of $K^{i+p}$) instead of $\mathcal{B}_k^i$ gives us what we want and this is well defined since both $\mathcal{B}_k^{i+p}$ and $\mathcal{Z}_k^i$ are subgroups of $\mathcal{C}_k^{i+p}$ and thus so is their intersection, which of course also is a subgroup of $\mathcal{Z}_k^i$. Thus we have the following definition.

DEFINITION 2.1. *The $p$ **-persistent $k$:th homology group of** $K^i$ is*

$$\mathcal{H}_k^{i,p} = \mathcal{Z}_k^i/\mathcal{B}_k^{i+p} \cap \mathcal{Z}_k^i.$$

Analogously to the classical version the $p$-persistent k:th Betti number of $K^i$, denoted by $\beta_k^{i,p}$, is defined to be the rank of the free part of $\mathcal{H}_k^{i,p}$.
In this thesis we will use the almost trivial filtered complex in shown in fig.1 as the main example in 2 dimensions. It contains the topological attributes we are interested in and is simple enough to be understood by anyone. We can represent the persistence data for a filtered complex $K^i$ in different ways. One way which is especially useful and that has a crucial role in the algorithms presented later is as a collection of half-open intervals. To understand what they are and how they are constructed consider the following. Assume that the $i$:th simplex/cube, $\kappa_i$, in the filtration gives rise to a new non-bounding $k$-cycle as it is added to the complex. The homology class of this cycle is a member of $\mathcal{H}_k^i$ and we denote it by $[z]$. Now, later in the filtration, say that the $j$:th simplex/cube, $\kappa_j$, as it is added to the complex makes a cycle $\hat{z} \in [z]$ bounding so that $\hat{z} \in \mathcal{B}_k^j$. When this happens, at filtration index $j$, the homology class $[z]$ ceases to exist on its own and is merged with older cycle classes and thereby the rank of the homology group is lowered. Thus the homology class $[z]$ existed on its own from filtration index $i$ up until filtration

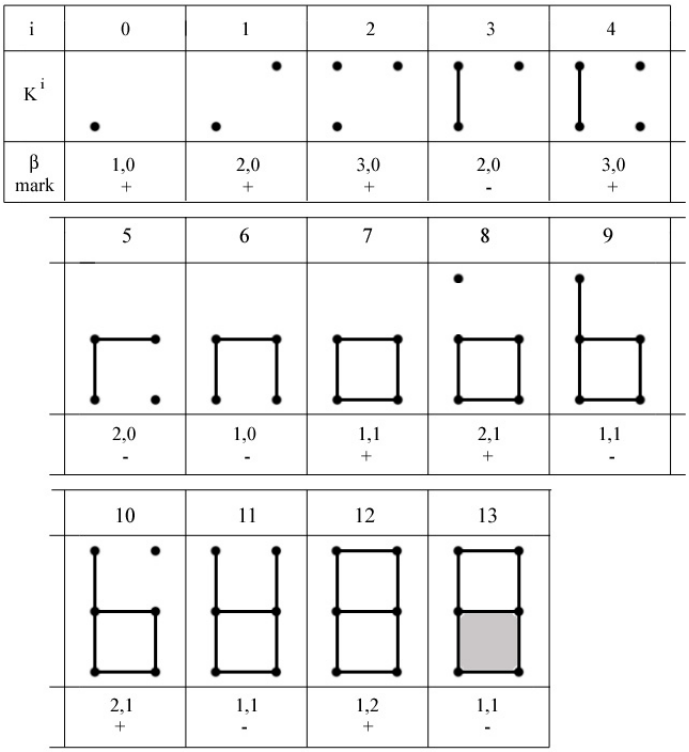



FIGURE 1. Example of a filtered cubical complex in 2D with Betti numbers and information about whether the $i$:th cube is positive/creator or negative/destroyer.

index $j$ and we represent this as a so called $k$-interval $[i, j)$, and the persistence of $z$, and $[z]$, is $i - j - 1$. If a cycle/homology class is created by the $i$:th simplex/cube but is not destroyed by any later simplex/cube in the filtration then the corresponding interval is $[i, \infty)$ and the cycle/homology class is said to have infinite persistence. Furthermore, we call $\kappa_i$ the creator and $\kappa_j$ the destroyer of the homology class $[z]$. Later, in the persistence algorithms and implementations we will mark cubes that are creators as positive and cubes that are destroyers as negative. It is also common to represent the persistence data, for torsion free spaces, visually in the index-persistence plane, where each $k$-interval $[i, j)$ gives rise to a $k$-triangle , see figure 2. If we represent all $k$-intervals corresponding to a filtered complex $K^i$ as $k$-triangles in the index-persistence plane, then as shown in [**Zom05**] p. 104 the Betti number $\beta_k^{i,p}$ is the number of $k$-triangles that contain the point $(i, k)$. There is also another number called the $p$-persistent $k$:th square Betti number, denoted by $\gamma_k^{i,p}$, that is sometimes useful. It is defined as the number of $k$-squares that contain the point (i,p) in the index-persistence plane, see figure 3. Later in chapter 4 we show implementations that calculates both of these Betti numbers.

Having the homology of each complex $K^i$ in a filtration as separate entities is sufficient for some purposes but for many important things, such as classification of the persistent homology groups, it is useful to collect the homology of all the complexes in the filtration into one single structure. This is achieved through the following.

DEFINITION 2.2. *A **persistence complex** $\wp$ is a collection of chain complexes $\{K_*^i\}_{i\geq 0}$ over a ring $R$, together with chain maps $f^i : K_*^i \to K_*^{i+1}$, so that we have*

$$K_*^0 \xrightarrow{f^0} K_*^1 \xrightarrow{f^1} K_*^2 \xrightarrow{f^2} \cdots .$$

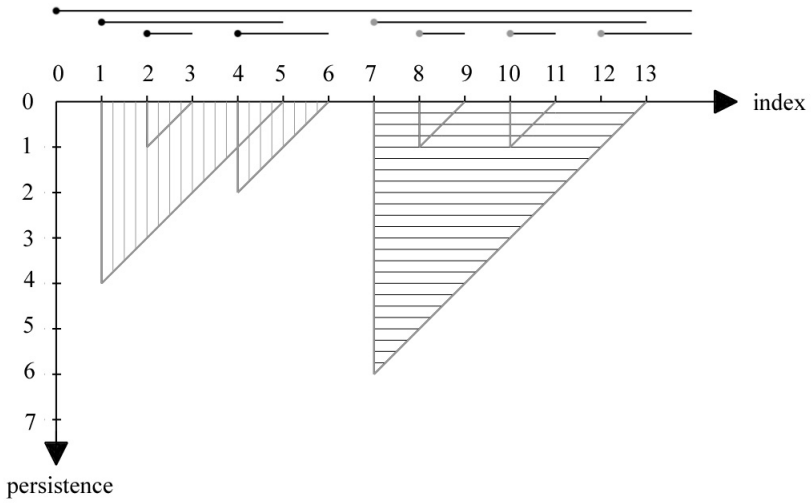



FIGURE 2. The $k$-triangles in the index-persistence plane for the example shown in figure 1. 0-triangles are filled with black vertical lines and 1-triangles with grey horizontal lines and above them the corresponding $k$-intervals are drawn. The infinite triangles that the 0:th and the 12:th cubes gives rise to are not included in the image.

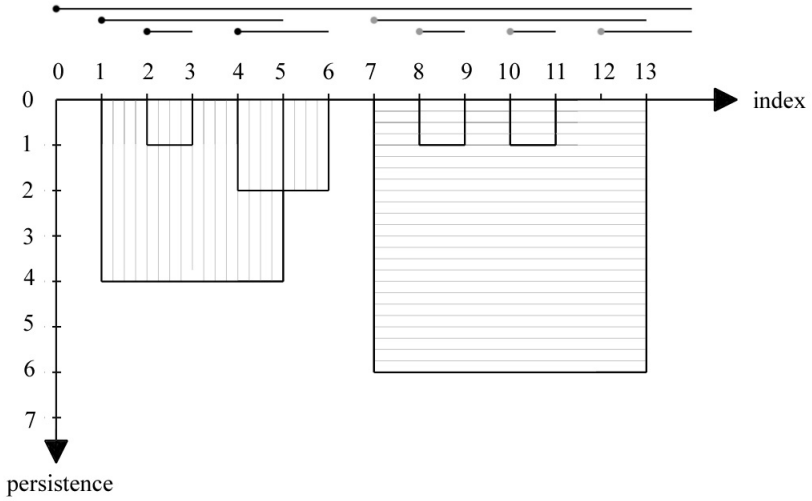



FIGURE 3. The $k$-squares in the index-persistence plane for the example shown in figure 1. 0-squares are filled with black vertical lines and 1-squares with grey horizontal lines and above them the corresponding $k$-intervals are drawn. The infinite squares that the 0:th and the 12:th cubes gives rise to are not included in the image.

A filtered complex $K$ with inclusion maps for the simplices/cubes becomes a persistence complex. This far we have managed to get all the complexes of a filtration into one structure, the persistence complex. Next we define the notion of a persistence module, which is the type of structure the homology of a persistence complex will be, where the maps $\phi^i$ take a homology class to the one that contains it. Thus we have managed to get the homology of a whole filtration into one entity.

DEFINITION 2.3. *A **persistence module** $\mathcal{M} = \{M^i, \phi^i\}_{i\geq 0}$ over a ring $R$ is a collection of $R$-modules $M^i$ together with homomorphisms $\phi^i : M^i \to M^{i+1}$.*

All complexes, both cubical and simplicial, in this thesis are finite and will therefore generate persistence complexes/modules of finite type, in the sense that each component complex/module is a finitely generated $R$-module and the maps $f^i/\phi^i$ are isomorphisms for $i \geq m$ for some $m \in \mathbb{Z}$. Now assume that we have a persistence module $\mathcal{M} = \{M^i, \phi^i\}_{i\geq 0}$ over $R$, and equip $R[t]$ with the standard grading and define a graded $R[t]$-module by

$$\alpha(\mathcal{M}) = \bigoplus_{i=0}^{\infty} M^i,$$

where the $R$-module structure is the sum of the structures on the individual components and where the action of $t$ is given by

$$t \cdot (m^0, m^1, \ldots) = (0, \phi^0(m^0), \phi^1(m^1), \ldots).$$

This correspondence $\alpha$ is the key to understanding the structure of persistence, and the following theorems are stated and proved in [**Zom05**] p. 103.

THEOREM 2.4 (structure of persistence). *The correspondence $\alpha$ defines an equivalence of categories between the category of persistence modules of finite type over $R$ and the category of finitely generated non-negatively graded modules over $R[t]$.*

As is also mentioned in [**CZ04**] this theorem shows that there is no simple classification of persistence modules over rings that are not fields, such as $\mathbb{Z}$. This because it is well known in commutative algebra that the classification of modules over $\mathbb{Z}[t]$ is extremely complicated. However, when the ring is a field, so that $F[t]$ is a $PID$, using the structure theorem for finitely generated $D$-modules gives us the following decomposition of the persistence module.

$$\left(\bigoplus_{i=1}^{n} \Sigma^{\alpha_i} F[t]\right) \oplus \left(\bigoplus_{i=1}^{m} \Sigma^{\gamma_i} F[t]/(t^{n_j})\right).$$

Now we tie this together somewhat to the discussion on $k$-intervals we had before. This by parameterizing the isomorphism classes of $F[t]$-modules by so called $\mathcal{P}$-intervals, which we define to be an ordered pair $(i, j)$ with $0 \leq i < j$, and $i \in \mathbb{Z}$ and $j \in \mathbb{Z} \cup \{\infty\}$. Next a graded $F[t]$-module is associated to a set $\mathcal{S}$ of $\mathcal{P}$-intervals via a bijection $\mathcal{Q}$. For $\mathcal{P}$-interval $(i, j)$ define

$$\mathcal{Q}(i, j) = \Sigma^i F[t]/(t^{j-1})$$

and for $(i, \infty)$

$$\mathcal{Q}(i, \infty) = \Sigma^i F[t].$$

For a set of $\mathcal{P}$-intervals $\mathcal{S} = \{(i_1, j_1), (i_2, j_2), \cdots, (i_n, j_n)\}$ define

$$\mathcal{Q}(\mathcal{S}) = \bigoplus_{l=1}^{n} Q(i_l, j_l).$$

The correspondence from above may now be restated as follows.

THEOREM 2.5. *The correspondence $\mathcal{S} \to \mathcal{Q}(\mathcal{S})$ defines a bijection between the finite set of $\mathcal{P}$-intervals and the finitely generated graded modules over the graded ring $F[t]$. Thus the isomorphism classes of persistence modules of finite type over $F$ are in bijective correspondence with the finite sets of $\mathcal{P}$-intervals.*

Now, using this theorem one can prove that the assertion we made earlier, that $\beta_k^{i,p}$ is equal to the number of $k$-triangles containing the point $(i, p)$ in the index-persistence plane. This and the details of the theorem above may be found in [**Zom05**] p. 104 where the following theorem is also explicitly stated.

THEOREM 2.6. *Let $\mathcal{T}$ be the set of triangles defined by $\mathcal{P}$-intervals for the $k$-dimensional persistence module. Then the rank $\beta_k^{i,p}$ of $\mathcal{H}_k^{i,p}$ is the number of triangles in $\mathcal{T}$ that contain the point $(i, p)$.*

This also shows that for computing persistent homology over a field it is sufficient to find the corresponding set of $\mathcal{P}$-intervals.

CHAPTER 3

# Algorithms

In this chapter we will take a look at some algorithms for computing static homology and persistent homology. The main algorithm for static homology is the reduction algorithm and this represents the standard way to compute the homology of a simplicial complex. It is also the basis for computing the homology of cubical complexes as is carried out in [**KMM04**]. Several persistence algorithms are presented. First we consider the Edelsbrunner incremental algorithm which enables us to calculate the homology of torsion free filtered subcomplexes of $\mathcal{S}^3$. Slightly modified this is also the first step in persistence calculations and a basic component in the extended persistence algorithm which works in arbitrary dimension with coefficients from a field and even PID:s. It should be noted that these latter persistence algorithms use a version of the classical reduction algorithm.Unless otherwise specified assume that we have cubical complexes and coefficients in $\mathbb{Z}_2$, but remember that *the Universal Coefficient Theorem for Homology*, see e.g. [**Hat01**], tells us that our calculations will give us the same Betti numbers as calculating over $\mathbb{Z}$ would.

## 1. The Reduction Algorithm

The reduction algorithm (RA) is the standard way to calculate the homology of a simplicial (and cubical) complex. The idea is to systematically reduce the given complex to a minimal one without changing the homology. In this thesis we simply take the classical reduction algorithm for simplicial complexes, take away things not necessary for mod 2 calculations, and modify it to work with cubical complexes. Furthermore, we only consider the cases where the dimension of the complex is 2 or 3 (without torsion) but in theory the reduction algorithm works for any finite dimensional complex (with or without torsion). An implementation for simplicial complexes of any dimension and with, or without, torsion constructed by R. Andrew Hicks is called MOISE and may be found at http://www.maplesoft.com/Members/. The classical reduction algorithm works like this.

Input: simplicial complex $K$
Output: Betti numbers and torsion coefficients of a simplicial complex $K$.

1. Construct ordered bases for the chains in each dimension. I.e. collect all simplices of the same order from $K$ and store them in an ordered list.
2. Find the boundary operators in matrix form relative to the ordered bases created previously.
3. Reduce the boundary operators to Smith Normal Form.
4. Read off the torsion coefficients from the boundary operators in Smith Normal Form.
5. Calculate the Betti numbers by first finding the column dimensions and the ranks of the boundary operators. (The Smith Normal Form of a matrix and the matrix itself will have the same rank and dimensions.)

If we are only interested in the Betti numbers we may skip the calculation of the Smith normal form since this is only used in finding torsion coefficients. This reduces

the calculation time since a very time consuming part of the algorithm is the reduction to Smith normal form, hence if one knows that there is no torsion this part should not be included. What we do explicitly in this thesis is to implement such a reduced reduction algorithm (RRA) in 2 and 3 dimensions with cubical complexes instead of simplicial. It should be noted though that it is possible to carry out the full reduction algorithm on cubical complexes as is shown in [**KMM04**]. Thus our algorithm can be described as follows.

Input: cubical complex $K$.
Output: Betti numbers of $K$.

1. Construct ordered bases for the chains in each dimension. That is, collect all elementary cubes of the same order from $K$ and store them in an ordered list.
2. Find the boundary operators in matrix form relative to the ordered bases created previously.
3. Calculate the Betti numbers by first finding the column dimensions and the ranks of the boundary operators.

However, even this reduced version is slow compared to the incremental algorithm for computing Betti numbers that will be introduced next.

## 2. Edelsbrunner's Incremental Algorithm

Edelsbrunner's Incremental Algorithm (EIA), first presented in [**DE95**], can be used to find the Betti numbers of torsion free filtered subcomplexes of $S^3$. The idea is that we add simplices one by one from the filtration, say that we add a $k$-simplex at time $i$, and then we check whether the newly added $k$-simplex belongs to $k$-cycle in $K^i$. If it does then we add 1 to the $k$:th Betti number $\beta_k$, else we subtract 1 from the $(k-1)$:th Betti number $\beta_{k-1}$. The EIA can be summarized by the pseudo-code in figure 1.The main problem is to decide whether or not a newly added $k$-simplex belongs to a $k$-cycle or not. For 0-simplices this is trivial since each of them are 0-cycles also. 1-simplices require a little more work. The idea is the following. Whenever a 0-simplex is added and thereby creating the next complex in the filtration the set of it is added to a special list $S$. Now, when a 1-simplex is added in the filtration we search the list $S$ and determine if the endpoints of the 1-simplex belongs to the same set in $S$ or not. If they do, then the 1-simplex belongs to a 1-cycle, if they belong to different sets, say $A$ and $B$, in $S$ then we add $A \cup B$ to $S$ and remove $A$ and $B$. This method to detect cycles is usually called *Union-Find* and we use it in this thesis. There is, however, a way to achieve the same thing without Union-Find as is explained in [**Zom05**]. Now 2-cycles are more difficult. Actually, the reason that EIA only works up to $S^3$ is that we (at present) only have means to detect 1-cycles and $(d-1)$-cycles in subcomplexes of $S^d$. Hence the highest complete case is $d = 3$. For example, for $d = 4$ we would be able to detect 1- and 3-cycles but not 2-cycles. The reason for this will be clear when we now explain how we detect 2-cycles in subcomplexes of $S^3$.

## 3. Persistence Algorithms

**3.1. 2D and 3D.** The goal of the persistence algorithms is to take a filtered complex and from this find pairs, consisting of the creator (a $k$-cube) and, if one exists, the destroyer (a $(k+1)$-cube) of each $k$-cycle. Having such a pair and knowing its components place in the filtration enables us to calculate the "lifetime" of the topological feature which corresponds to the non-bounding $k$-cycle in terms of the filtration index. If the creator's filtration index is $i$ and the destroyer's filtration index is $j$ then we represent the creator-destroyer pair as $[i, j)$ and think of this a representing an interval, which we call a *$k$-interval*. If a $k$-cycle created by a cube with index $i$ does not become

```
for i from 0 to 2 do
    β_i = 0;
end for
for j from 0 to m do
    k = dim σ_j;
    if σ_j belongs to a k-cycle in K^j then
        β_j = β_j + 1;
    else
        β_{j-1} = β_{j-1} - 1;
    end if
end for
```

FIGURE 1. Pseudo-code for the EIA. The input $K$ is a filtered complex containing $m$ simplicies and the output is the Betti numbers $\beta_i$. $\sigma_j$ is the $j$:th simplex in the filtration and its addition results in the complex $K^j$.

```
for j from 0 to m-1 do
    k = dim σ_j;
    if σ_j belongs to a k-cycle in K^j then
        mark σ_j as positive
    else
        mark σ_j as negative
    end if
end for
```

FIGURE 2. Pseudo-code for *mark* The input $K$ is a filtered complex containing $m$ simplicies and the output is a list of the simplices in F marked as either positive or negative. $\sigma_j$ is the $j$:th simplex in the filtration and its addition results in the complex $K^j$.

```
for i from 0 to dim K do
    P_i = ∅;
end for
for j from 0 to m-1 do
    k = dim σ_j;
    if σ_j is negative then
        b = ∂_k σ_j;
        n = max{l : σ_l in b};
        P_{k-1} = P_{k-1} ∪ {(σ_j, σ_n)};
    end if
end for
```

FIGURE 3. Pseudo-code for *pairing*. The input $K$ is a filtered complex containing $m$ simplicies and the output is a list of pairs consisting of the creator and the destroyer of a particular homology class. $\sigma_j$ is the $j$:th simplex in the filtration and its addition results in the complex $K^j$.

bounding within the filtration, i.e. a destroyer of it does not exist, then we say that $j = \infty$ and the corresponding interval becomes $[i, \infty)$. We may show this information for a filtered complex visually by plotting so called barcodes. Finally we can use the pairings to calculate the $p$-persistent $k$:th Betti numbers and also the square Betti Numbers. The pairing algorithm is of course the key algorithm here, but it has a crucial support algorithm that marks the every cube/simplex in the filtration as either positive or negative. This marking algorithm is actually the EIA described previously with the difference that instead of raising or lowering Betti numbers when a cube/simplex is entered it is marked as either positive or negative. In pseudo-code these two algorithms may be presented as in figure 2 and 3.

CHAPTER 4

# Implementations

## 1. Filtering Procedures

For many of the things we wish to do we need a filtered cubical complex. We have deviced two related ways to obtain a totally ordered filtered cubical complex from a collection of $r$ pairs of points in $\mathbb{R}^2$ or triples in $\mathbb{R}^3$. We represent this collection as a $r \times 2$ or $r \times 3$ matrix, respectively, and such a matrix is the input to the filtering procedures. All of the filtering procedures are highly dependent on the following procedure cubsim that sets up an elementary cube in $\mathbb{R}^2$ or $\mathbb{R}^3$ for each pair or triple in the input matrix. This is especially import to study since it reveals how we have chosen to represent elementary cubes in this project. Of course there are more clever ways to represent cubes, e.g. as hashes, but we have chosen this one so that the structure of everything will be absolutely clear. Later when we have a working set of procedures this will be the first to be modified. Here is a detailed description of it.

**1.1. *cubsim.*** The *cubsim* procedure takes two real numbers $x$ and $y$, in our case representing coordinates of a point in the plane. It then constructs an elementary cube (and if necessary its boundaries, and all its boundaries boundaries) in the plane containing the point $(x, y)$. Of course in each relevant case the main cube, boundaries and boundaries of boundaries are created in reverse order so that the addition of this collection of elementary cubes do not destroy the filter property of the filtration. Now, depending on whether $x$ and/or $y$ are integers we get four different cases. If $x$ and $y$ are both integers then the single point $[x, y]$ is returned. If $x$ is an integer and $y$ not an integer then the line segment $[x, [\lfloor y \rfloor, \lceil y \rceil]]$ and its endpoints $[x, \lfloor y \rfloor]$ and $[x, \lceil y \rceil]]$ are added and similarly if $y$ is an integer and $x$ is not. If neither x or y are integers then the square $[[\lfloor x \rfloor, \lceil x \rceil], [\lfloor y \rfloor, \lceil y \rceil]]$ and all its boundary line segments and all of their end points. The actual MAPLE code is.

```
cubsim:=proc(x,y)
  local s;
  if x::integer then
    if y::integer then
      s:=[x,y];
    else
      s:=[x,floor(y)],[x,ceil(y)],[x,[floor(y),ceil(y)]];
    end if:
  else
    if y::integer then
      s:=[floor(x),y],[ceil(x),y],[[floor(x),ceil(x)],y];
    else
      s:=[floor(x),floor(y)],[floor(x),ceil(y)],
      [ceil(x),floor(y)], [ceil(x),ceil(y)],
      [[floor(x),ceil(x)],floor(y)],
      [[floor(x),ceil(x)],ceil(y)],
      [floor(x),[floor(y),ceil(y)]],
```

```
      [ceil(x),[floor(y),ceil(y)]],
      [[floor(x),ceil(x)],[floor(y),ceil(y)]];
    end if:
  end if:
  s:
end proc:
```

The equivalent procedure for 3D looks like

```
cubsim:=proc(x,y,z)
  local s,X,Y,Z;
  X:=[floor(x),ceil(x)];
  Y:=[floor(y),ceil(y)];
  Z:=[floor(z),ceil(z)];
  if x::integer then
    if y::integer then
      if z::integer then
        s:=[x,y,z];
      else
        s:=[x,y,Z[1]],[x,y,Z[2]],[x,y,[Z[1],Z[2]]];
      end if:
    else
      if z::integer then
        s:=[x,Y[1],z],[x,Y[2],z],[x,[Y[1],Y[2]],z];
      else
        s:=[x,Y[1],Z[1]],[x,Y[1],Z[2]],[x,Y[2],Z[1]],
        [x,Y[2],Z[2]],[x,Y[1],[Z[1],Z[2]]],
        [x,Y[2],[Z[1],Z[2]]],[x,[Y[1],Y[2]],Z[1]],
        [x,[Y[1],Y[2]],Z[2]],
        [x,[Y[1],Y[2]],[Z[1],Z[2]]];
      end if:
    end if:
  else
    if y::integer then
      if z::integer then
        s:=[X[1],y,z],[X[2],y,z],[[X[1],X[2]],y,z];
      else
        s:=[X[1],y,Z[1]],[X[1],y,Z[2]],[X[2],y,Z[1]],
        [X[2],y,Z[2]],[X[1],y,[Z[1],Z[2]]],
        [X[2],y,[Z[1],Z[2]]],[[X[1],X[2]],y,Z[1]],
        [[X[1],X[2]],y,Z[2]],
        [[X[1],X[2]],y,[Z[1],Z[2]]];
      end if:
    else
      if z::integer then
        s:=[X[1],Y[1],z],[X[1],Y[2],z],[X[2],Y[1],z],
        [X[2],Y[2],z],[X[1],[Y[1],Y[2]],z],
        [X[2],[Y[1],Y[2]],z],[[X[1],X[2]],Y[1],z],
        [[X[1],X[2]],Y[2],z],[[X[1],X[2]],[Y[1],Y[2]],z];
      else
        s:=[X[1],Y[1],Z[1]],[X[1],Y[1],Z[2]],
```

```
          [X[1],Y[2],Z[1]],[X[1],Y[2],Z[2]],
          [X[2],Y[1],Z[1]],[X[2],Y[1],Z[2]],
          [X[2],Y[2],Z[1]],[X[2],Y[2],Z[2]],
          [X[1],Y[1],[Z[1],Z[2]]],[X[1],Y[2],[Z[1],Z[2]]],
          [X[1],[Y[1],Y[2]],Z[1]],[X[1],[Y[1],Y[2]],Z[2]],
          [X[2],Y[1],[Z[1],Z[2]]],[X[2],Y[2],[Z[1],Z[2]]],
          [X[2],[Y[1],Y[2]],Z[1]],[X[2],[Y[1],Y[2]],Z[2]],
          [X[1],Y[1],[Z[1],Z[2]]],[X[2],Y[1],[Z[1],Z[2]]],
          [[X[1],X[2]],Y[1],Z[1]],[[X[1],X[2]],Y[1],Z[2]],
          [X[1],Y[2],[Z[1],Z[2]]],[X[2],Y[2],[Z[1],Z[2]]],
          [[X[1],X[2]],Y[2],Z[1]],[[X[1],X[2]],Y[2],Z[2]],
          [X[1],[Y[1],Y[2]],Z[1]],[X[2],[Y[1],Y[2]],Z[1]],
          [[X[1],X[2]],Y[1],Z[1]],[[X[1],X[2]],Y[2],Z[1]],
          [X[1],[Y[1],Y[2]],Z[2]],[X[2],[Y[1],Y[2]],Z[2]],
          [[X[1],X[2]],Y[1],Z[2]],[[X[1],X[2]],Y[2],Z[2]],
          [X[1],[Y[1],Y[2]],[Z[1],Z[2]]],
          [X[2],[Y[1],Y[2]],[Z[1],Z[2]]],
          [[X[1],X[2]],Y[1],[Z[1],Z[2]]],
          [[X[1],X[2]],Y[2],[Z[1],Z[2]]],
          [[X[1],X[2]],[Y[1],Y[2]],Z[1]],
          [[X[1],X[2]],[Y[1],Y[2]],Z[2]],
          [[X[1],X[2]],[Y[1],Y[2]],[Z[1],Z[2]]];
      end if:
    end if:
  end if:
  s;
end proc:
```

As can be seen this code is quite large and the representation of the cubes is obviously not very good. As a matter of fact, before we can do any real applied work in 3D we must reconstruct this because at the moment we cannot use it for large datasets since it occupies to much memory and is to slow. We do however have several ideas for a new more efficient representation that we will discuss later.

**1.2.** ***bdop.*** Another small procedure used frequently in the following procedures is *bdop* which is used for calculating the (mod 2) boundary of a cube.

```
bdop:=proc(s)
  local bd,i,k,x,y;
  x:=s[1]:  y:=s[2];
  k:=nops(x)+nops(y);
  if k=2 then
    bd:=[];
  end if:
  if k=3 then
    if nops(x)=1 then
      bd:=[[x,y[1]],[x,y[2]]];
    else
      bd:=[[x[1],y],[x[2],y]];
    end if;
  end if;
  if k=4 then
```

```
    bd:=[[[x[1],x[2]],y[1]],[x[1],[y[1],y[2]]],
    [[x[1],x[2]],y[2]],[x[2],[y[1],y[2]]]];
  end if:
  bd;
end proc:
```

The 3D eqivalent is coded as follows.

```
bdop:=proc(s)
  local bd,i,k,x,y,z;
  x:=s[1]:    y:=s[2];   z:=s[3];
  k:=nops(x)+nops(y)+nops(z);
  if k=3 then
    bd:=[];
  end if:
  if k=4 then
    if nops(x)=1 then
      if nops(y)=1 then
        bd:=[[x,y,z[1]],[x,y,z[2]]];
      else
        bd:=[[x,y[1],z],[x,y[2],z]];
      end if;
    else
      bd:=[[x[1],y,z],[x[2],y,z]];
    end if;
  end if:
  if k=5 then
    if nops(x)=1 then
      bd:=[[x,y[1],[z[1],z[2]]],[x,y[2],[z[1],z[2]]],
      [x,[y[1],y[2]],z[1]],[x,[y[1],y[2]],z[2]]];
    else
      if nops(y)=1 then
        bd:=[[x[1],y,[z[1],z[2]]],[x[2],y,[z[1],z[2]]],
        [[x[1],x[2]],y,z[1]],[[x[1],x[2]],y,z[2]]];
      else
        bd:=[[x[1],[y[1],y[2]],z],[x[2],[y[1],y[2]],z],
        [[x[1],x[2]],y[1],z],[[x[1],x[2]],y[2],z]];
      end if:
    end if:
  end if:
  if k=6 then
    bd:=[[x[1],[y[1],y[2]],[z[1],z[2]]],
    [x[2],[y[1],y[2]],[z[1],z[2]]],
    [[x[1],x[2]],y[1],[z[1],z[2]]],
    [[x[1],x[2]],y[2],[z[1],z[2]]],
    [[x[1],x[2]],[y[1],y[2]],z[1]],
    [[x[1],x[2]],[y[1],y[2]],z[2]]];
  end if:
  bd;
end proc:
```

**1.3. *filter*.** Now that we have the ability to set up a cube containing a point in $\mathbb{R}^2$ or $\mathbb{R}^3$ the next step is to do the same for a collection of points and also making sure that the resulting cubical complex will be properly filtered. We achieve this using the following procedure. We show the 2D code, the extension to 3D is obvious.

```
filter:=proc(M::Matrix)
  local i,n,F,L,MM:
  MM:=mkint(M):
  n:=LinearAlgebra[RowDimension](MM);
  F:=vector(n);
  for i from 1 to n do
    F[i]:=cubsim(MM[i,1],MM[i,2]);
  end do:
  L:=convert(eval(F),list):
  ListTools[MakeUnique](L):
end proc:
```

That this code gives a totally ordered filter follows from the construction of *cubsim* and the fact that the *ListTools[MakeUnique]* procedure keeps the first (i.e. the leftmost) entry it comes across and deletes the following ones. If it were to proceed differently we could not have used it since it would then spoil the filtration. The usefulness of this filtration in persistence calculations and applications may be debated but it is a nice and easy way to obtain filtrations for testing procedures that require a filtration as input. Also, this procedure should be optimized as soon as possible. First of all, some mechanism that prevents *filter* from calling *cubsim* for the same elementary cube more than once should be deviced. This would save us the time of calling cubsim a number of times for nothing, and for an input matrix with many non-integer entries from a small domain this could save a substantial amount of time, and memory. Also, if we managed to create such a mechanism we would not need the *ListTools[MakeUnique]* procedure. Several ideas to solve this are known but they are likely to add more complexity than is saved from not having to call *cubsim* a few times extra and make the list unique.
Another interesting way to obtain a filtration which is based on *filter* is given by the following set of procedures.

**1.4. *cgfilter*.** *cgfilter* and its support procedures is used for create a filtration of filtrations of a growing cubical complex. This particular version is such that it takes as input a matrix of coordinates, like our previous filtration techniques, and in addition a positive integer $N$. The first step does exactly what the *filter* procedure does, it sets up elementary cubes containing each point represented in the matrix. From this *cgfilter* then creates a filter of all cubes which are within or at a distance of 1 (in the max-metric) from a cube created in the first step. Each cube from the first step is also included in this filtration. Next it creates a filter of all cubes within or at a distance of 2 of a cube created in the first step. It continues in this manner until it has reached a distance of $N$. The actual code for this consists of three procedures, *cgfilter* and the two support procedures *ep* and *cubgrow*.

*ep* takes as input two real numbers, say $x$ and $y$, and a positive integer $N$ and from this it generates a list of points at a distance of 1 from $(x, y)$. If $x$ and $y$ are both not integers then exactly one point will be in each of the nine lattice squares adjacent to the lattice square containing $(x, y)$. If either of $x$ or $y$ is an integer some of the points

will be on adjacent lattice square boundaries and if both $x$ and $y$ are integers we get the nine adjacent lattice points.

```
ep:=proc(x,y,N)
  local j,k,P;
  P:=;
  for j from 0 to N do
    for k from 0 to N do
      P:=P union {[x+j,y+k], [x+j,y-k], [x-j,y+k], [x-j,y-k]};
    end do:
  end do:
  P;
end proc:
```

*cubgrow* takes as input a matrix $M$ of the same type as our other filter procedures and also a positive integer $N$. From this it creates a list of points for each $N$ by adding all the points from $M$ and all of their the adjacent points as created by *ep*.

```
cubgrow:=proc(M::Matrix,N)
  local i,j,k,F,G,RD;
  RD:=LinearAlgebra[RowDimension](M);
  F:=array(1..N);
  G:=Matrix(1..RD,1..N);
  for i from 1 to RD do
    for j from 1 to N do
      G[i,j]:=ep(M[i,1],M[i,2],j);
    end do:
  end do:
  for k from 1 to N do
    F[k]:=[seq(G[l,k],l=1..RD)];
  end do;
  convert(F,list);
end proc:
```

Now the final procedure *cgfilter* takes a coordinate matrix and creates a filtration of filtrations by using *ep* and *cubgrow*.

```
cgfilter:=proc(M::Matrix,N)
  local i,F,FP;
  F:=array(1..N);
  FP:=cubgrow(M,N);
  for i from 1 to N do
    F[i]:=filter(convert(FP[i],Matrix));
  end do:
end proc:
```

## 2. EIA with filtered cubical complex in 2D

The EIA with filtered cubical complex in 2D is implemented through the procedures *bettic* and *mark* which are almost identical code-wise. The difference is that *bettic* produces the Betti numbers whereas *mark* returns each cube in the filtration and information about whether it is positive or negative, i.e. if it gives rise to a new non bounding cycle or if it destroys one. The calculations for the two are thus the same,

but in *mark* we mark cubes as positive or negative instead of rising or lowering Betti numbers. For both procedures the input is a filtered cubical complex. *bettic* is not used as a support procedure for anything later whereas *mark* is used in the procedures for calculating persistence. The algorithm was described in chapter 3 and the actual MAPLE code for *bettic* is the following.

```
bettic:=proc(F)
  local b,i,j,k,m,s,x,y,Q,S;
  b:=[0,0]; S:=[];
  for i from 1 to nops(F) do
    x:=F[i][1]; y:=F[i][2];
    k:=nops(x)+nops(y);
    if k=2 then
      b[1]:=b[1]+1;
      S:=[op(S),{F[i]}];
    end if:
    if k=3 then
      Q:={};
      if nops(x)=1 then
        s:=[[x,y[1]],[x,y[2]]];
      else
        s:=[[x[1],y],[x[2],y]];
      end if:
      for j from 1 to nops(S) do
        for m from 1 to nops(s) do
          if member(s[m],S[j])=true then
            Q:=Q union {j};
          end if:
        end do:
      end do:
      if nops(Q)=1 then
        b[2]:=b[2]+1;
      else
        b[1]:=b[1]-1;
        Q:=convert(Q,list);
        S:=[op(S[1..Q[1]-1]),op(S[Q[1]+1..Q[2]-1]),
        op(S[Q[2]+1..nops(S)]),S[Q[1]] union S[Q[2]]];
      end if:
    end if:
    if k=4 then
      b[2]:=b[2]-1;
    end if;
  end do;
  b;
end proc:
```

## 3. RRA

Here we present an implementation of the reduced reduction algorithm, called *hom*. It can calculate the Betti numbers of any cubical complex in 2D. It takes a cubical complex in filtered form as input. Then it uses a support procedure called *simp* to create a list of lists each of which contains all cubes of a particular dimension. Here

of course only -1, 0, 1 and 2. Each of these lists are thought of as an ordered basis for the chains in that dimension. Using these ordered bases another support procedure *bd* constructs the (transposed) boundary operators in each dimension. A final support procedure is called *entry* and given a particular cube it finds its place in basis. Using these procedures the main procedure hom is able to calculate the Betti numbers of the given cubical complex. The MAPLE code for *hom* and its three support procedures is given next.

```
simp:=proc(F)
  local i,j,k,x,y,K,S;
  S:=[[[NULL,NULL]]];
  K:={};
  for i from 1 to nops(F) do
    x:=F[i][1]:  y:=F[i][2];
    k:=nops(x)+nops(y):
    if k<>0 then
      if {k} subset K=false then
        S:=[op(S),[]];
      end if:
      S[k]:=[op(S[k]),F[i]]:
      K:=K union {k}:
    end if:
  end do:
  S:
end proc:

entry:=proc(F,s)
  local i; i:=1:
  while i<=nops(F) and F[i]<>s do
    i:=i+1;
  end do:
  if i<nops(F)+1 then
    i:
  end if:
end proc:

bd:=proc(F)
  local b,i,j,m,n,s,v,S,M;
  S:=simp(F);
  for n from 1 to nops(S)-1 do
    M[n]:=Array(1..nops(S[n+1]),1..nops(S[n]),fill=0):
    for m from 1 to nops(S[n+1]) do
      s:=S[n+1][m];
      b:=bdop(s);
      for i from 1 to nops(b) do
        j:=entry(S[n],b[i]):
        M[n][m,j]:=1;
      end do:
    end do:
  end do:
  convert(M,list):
```

```
end proc:

hom:=proc(F)
  local i,j,k,m,B,BS,T,M,Y,Betti,Betti1,Betti2;
  BS:=bd(F):
  B:=array(1..nops(BS));
  for k from 1 to nops(BS) do
    B[k]:=LinearAlgebra[Transpose](convert(BS[k],Matrix));
  end do:
  for i from 1 to nops(BS)-1 do
    Betti1[i]:=LinearAlgebra[ColumnDimension](B[i])-
    LinearAlgebra[Rank](B[i])-LinearAlgebra[Rank](B[i+1]);
  end do:
  Betti2:=convert(Betti1,list):
  Betti:=[op(Betti2)];
end proc:
```

## 4. Persistence in 2D

The first step in calculating persistence is to mark each $k$-simplex/cube as positive, if it gives rise to a new $k$-cycle/homology class, or negative if it destroys a $(k-1)$-cycle/homology class. To mark the cubes in a filtration we use a procedure called *mark*, which is closely related to the procedure *bettic* above. The difference is that *mark* marks simplices as positive or negative instead of raising and lowering Betti numbers. Next we want to create pair of creators/positive and destroyers/negative and for this we use the procedure *pairing* . It takes a filtered cubical complex as input and produces a list of pairs of cubes. Each pair consists of a $k$-cube which is responsible for creating a new non-bounding $k$-cycle and a $(k+1)$-cube which makes that k-cycle bounding. Hence these pairs contain the creator and the destroyer of a particular homology class. Its major support procedure is *mark*, which is a modified version of *bettic* presented above, In addition to just giving the cubes their positions in the original filtration is also given explicitly. This is of course somewhat redundant and should be fixed but at present it simplifies the work.

### 4.1. *pairing*.

```
pairing:=proc(F)
  local d,i,j,k,m,n,C,CC,COL,COLP,G,P,PP,T;
  P:=[[],[]]:    COL:=[]:    COLP:={}:    T:={}:
  G:=mark(F);
  for i from 1 to nops(G) do
    C:=[]:
    if G[i][2]=-1 then
      n:=nops(G[i][1][1])+nops(G[i][1][2]);
      d:=bdop(G[i][1]);
      for m from 1 to nops(d) do
        for j from 1 to i-1 do
          if d[m]=G[j][1] then
            C:=[op(C),j];
          end if:
        end do:
      end do:
      C:=sort(C);
```

```
      k:=nops(C):
      CC:=C[k];
      while member(CC,COL)=true and k>1 do
        k:=k-1:  CC:=C[k]:
      end do:
      if member(CC,COL)=true and k=1 then
        CC:=CC[1];
      end if:
      COL:=[op(COL),CC]:
      P[n-2]:= [op(P[n-2]),[[G[CC][1],G[i][1]],[CC,i]]]:
    else
      COLP:={op(COLP)} union {[G[i][1],i]}:
    end if:
  end do:
  for i from 1 to nops(COL) do
    T:={op(T)} union {[G[COL[i]][1],COL[i]]};
  end do;
  PP:=convert((COLP minus T),list);
  for j from 1 to nops(PP) do
    n:=nops(PP[j][1][1])+nops(PP[j][1][2]);
    if n=2 then
      P[1]:=[op(P[1]),[[PP[j][1],[]],[PP[j][2],nops(F)+1]]];
    end if:
    if n=3 then
      P[2]:=[op(P[2]),[[PP[j][1],[]],[PP[j][2],nops(F)+1]]];
    end if:
  end do:
  convert(P,list);
end proc:
```

One way of presenting persistence data in graphical form is as barcodes. Such a barcode plot shows the collection of $p$-intervals and hence one can see at what time (i.e. step of the filtration index) a $k$-cycle is born, when it vanishes and thus for how long it was present. This requires nothing special, just plot the result from the index-pairs from the procedure *pairing*. The MAPLE code for generating barcodes and examples of what they look like may be found in Appendix A. We can also calculate the $p$-persistent Betti numbers $\beta_k^{i,p}$ for a given $p$. The algorithm/implementation is based on the fact that $\beta_k^{i,p}$ is equal to the number of $k$-triangles that contain the point $(i,p)$ in the index-persistence plane. The procedure *pbettic* takes as input a filtration $F$ and a non negative integer $p$ and the output is a list of lists, one sublist for each $k$, and in each of these lists contain pairs of an index $i$ and its corresponding $p$-persistent Betti number $\beta_k^{i,p}$, i.e. $[i, \beta_k^{i,p}]$. The MAPLE code for *pbettic* is as follows.

### 4.2. *pbettic*.

```
pbettic:=proc(F,p)
  local b,i,j,k,m,x,y,I1,I2,P,S,S2;
  k:=nops(F); b:=[[],[]]; S:=[]; S2:=[]; I1:=[]; I2:=[];
  P:=pairing(F);
  for i from 1 to nops(P[1]) do
    if P[1][i][2][2]-P[1][i][2][1]>=p then
      I1:=[op(I1),P[1][i][2]];
```

```
    end if:
  end do:
  for j from 1 to nops(P[2]) do
    if P[2][j][2][2]-P[2][j][2][1]>=p then
      I2:=[op(I2),P[2][j][2]];
    end if:
  end do:
  for i from 1 to nops(I1) do
    S:=[op(S),seq(j,j=I1[i][1]..I1[i][2]-1-p)];
  end do:
  for j from 1 to nops(S) do
    m:=ListTools[Occurrences](S[j],S);
    b[1]:=[op(b[1]),[S[j],m]];
  end do;
  b[1]:=ListTools[MakeUnique](b[1]);
  for i from 1 to nops(I2) do
    S2:=[op(S2),seq(j,j=I2[i][1]..I2[i][2]-1-p)];
  end do:
  for j from 1 to nops(S2) do
    m:=ListTools[Occurrences](S2[j],S2);
    b[2]:=[op(b[2]),[S2[j],m]];
  end do:
  b[2]:=ListTools[MakeUnique](b[2]);
  b;
end proc:
```

Furthermore, as explained in the section on persistent homology it is sometimes useful to calculate the so called $p$-persistent $k$:th square Betti numbers, $\gamma_k^{i,p}$. A procedure called *psqbettic* for calculating the square Betti numbers has been contructed but is not included here since it is an exact copy of the procedure *pbettic* above with the difference that the term `-p` is dropped from the sequence endpoints in the `S` and `S2` lists in *pbettic*, i.e. `I1[i][2]-1-p` and `I2[i][2]-1-p` is replaced by `I1[i][2]-1` and `I2[i][2]-1`. Why it works like this is easily understood if one consider the geometric situation in the index-persistence plane, where the $\beta_k^{i,p}$ is equal to the number of $k$-triangles containing the point $(i,p)$ and the $\gamma_k^{i,p}$ equals the number of $k$-squares containing the point $(i,p)$.

## 5. Experimental tests

Most of these tests are conducted in the following way. An $r \times 2$-matrix with randomly chosen numbers from some interval $I$ as elements is constructed. This can be thought of as choosing points $(x,y)$ at random from some subsquare $I \times I$ of $\mathbb{R}^2$. This matrix is then used as input to the procedure *filter* which creates the filtered cubical complex $K$ that corresponds to it. This filter is then used as input to one run of the procedure to be tested and the time to completion is measured and plotted against the size of the filter, i.e. the number of processed elementary cubes. Usually either the number of points r or the length of I is changed incrementally in a sequence of runs and the result of the whole sequence is plotted together. The MAPLE code for a typical test procedure may be found in Appendix A. Other types of tests are also conducted but they are explained where they appear.

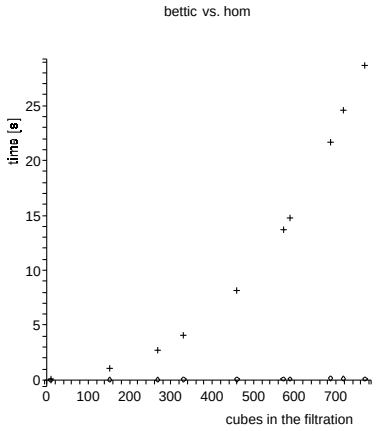



FIGURE 1. *bettic* vs. *hom* for up to about 800 elementary cubes and constant interval size $I = 20$.

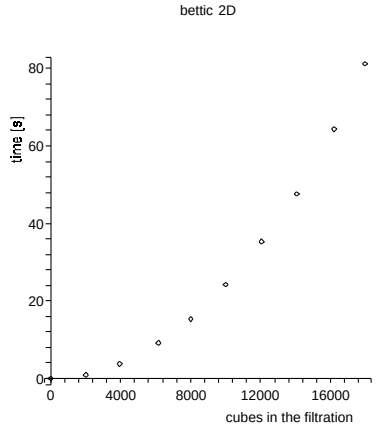



FIGURE 2. *bettic* for up to about 16000 elementary cubes and constant interval size $I = 500$.

**5.1. Comparison of *bettic* and *hom*.** Remember that *bettic* is a procedure based on the Edelsbrunner incremental algorithm (EIA) and *hom* is an implementation of the reduced reduction algorithm (RRA).

In fig.1 a comparison of *hom* (crosses) and *bettic* (circles) is shown and it is obvious *hom* is a lot slower than *bettic*. There are several reasons for this. First of all *hom* creates actual matrices for the boundary operators and these can be huge for large complexes. Furthermore, it uses Gaussian elimination ($O(n^3)$) on each of the (possibly huge) matrices to compute their rank. Hence it is no great surprise that *bettic* outperforms *hom* in this situation.

**5.2. *bettic*.** As we can see in fig.2 bettic grows fairly fast and if we want to use it for some real applications it needs to be faster. We do however believe that once we have choosen and implemented a more efficient representation of the elementary cubes it will be considerably faster. One reason for thinking that is that with the more efficient representation of the cubes we have in mind bettic will not have to process each and every elementary cube in the filtration. This is what bettic does at the moment and it is not really necessary.

**5.3. *pairing*.** In fig.3 we can see the behavior of *pairing* and at present it cannot handle the large data sets we want to use it for. But remember that it uses *mark* and

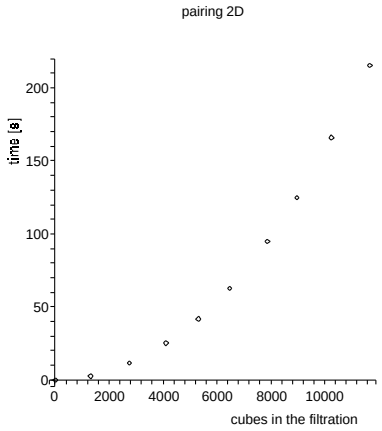



FIGURE 3. *pairing* for up to about 11000 elementary cubes and constant interval size $I = 200$.

*mark* is as we described earlier virtually a copy of *bettic* and thus the discussion above on *bettic* applies to *pairing* also. Hence improving *bettic* mean improving *mark* and thereby *pairing*. It is also to be expected that the non-*mark* part of *pairing* will also benefit greatly from a new cube representation.

CHAPTER 5

# Applications

Computational persistent and cubical homology each has a wealth of applications and as it seems even more proposed future applications. Here we present a brief account of a few main applications of the two and refer the reader to [**Zom05**] for a more detailed account of the applications of persistent homology and [**KMM04**] for applications of cubical homology. This brief survey gives us an idea of what kind of applications our method of cubical persistent homology might have.

## 1. Applications of Cubical Homology

The main areas in which computational cubical homology seems to have applications at present is in digital image processing/analysis, dynamical systems and medicine/structural biology. Its use in digital imagery is easy to understand since the main building blocks (in 2D) are squares and the theory concerns finite collections of such squares and in essence a digital image is a collection of squares (pixels). For an account of the cubical approach used in image analysis, in particular topological classification of 2D and 3D imagery see [**AMT01**]. Of course there are a few differences between a cubical set and a digital image. For example in a cubical set we can have isolated points and line segments and each line segment and square have a boundary whereas in a digital image the pixel is the smallest unit. However, as is described in [**KMM04**] this does not prevent us from using this to analyze/process digital images. The main applications in medicine/structural biology also concern digital imagery in a sense. For example, cubical homology has been used to extract topological information, e.g. cavities, directly from raw MRI/MRA-data. That is, without creating a visual representation/image of the scanned specimen/organ. The usefulness of this is that creating an image is costly in terms of time and memory used by the computer and thus if we are only interested in topological information we only need to use cubical homology which is much cheaper and thus allows for more detailed calcultaions under the same conditions. For examples of this approach being used to examine blood vessels and cavities in the brain see [**Mea02**]. It is reasonable to believe that a similar approach could be used to detect topological features in anything in three dimensions that we can somehow measure the local density inside of, e.g. engineering structures, rocks, the ground etc. Specific applications could be finding interior defects in bridges or concrete structures, gas pockets in rock or ground. Cubical homology has also been applied extensively in the study of dynamical systems. For example, phase separation processes in compound materials [**GMW05**] and topological characterization of certain types of chaos [**KGMS04**] [**GKM04**]. Several other projects involving cubical homology applied to dynamical systems, for example automated chaos proving, invariant sets in discrete systems etc. visit the CHOMP website http://www.math.gatech.edu/ chomp/.

## 2. Applications of Persistent Homology

Although persistent homology is a young invention there is a wealth of current and proposed future applications. As mentioned before a key strength of persistent homology

is that it provides a means to distinguish between topological noise and actual features. Looking at the history of persist homology and $\alpha$-shapes it is evident that a driving force in the development of these techniques has been problems in computational structural biology. In this subject persistent homology and $\alpha$-shapes has been successfully used for feature detection, knot detection and structure determination etc. In [**Zom05**] p. 223-226 there is account of how these methods were used to detect topological features in the protein gramicidin A and zeolite BOG. The same reference also describes the use of a linking number algorithm for knot detection in proteins and such and proposes the use of e.g. Alexander polynomials for this in the future. A method involving X-ray crystallography on crystals of protein and the persistence algorithm for structure determination is also described. For more information on the use of these methods in structural biology see e.g. [**EFL96**] and [**EFFL95**]. Another use of persistent homology is to discover/examine hierarchial clustering, e.g. the distribution of galaxies in the universe as initiated by Dyksterhouse in 1992 and explicitly proposed by Edelsbrunner and Mucke in 1994. This approach may also be used to classify proteins according to their hydrophobic surfaces [**Zom05**] p. 228. Applications outside structural biology include surface reconstruction from noisy samples [**DG04**]and the shape of point sets in the plane [**EKS83**].

CHAPTER 6

# Conclusions and Discussion

We managed to base persistent homology on cubical complexes and have created a working set of prototype procedures able to calculate the persistent homology of a filtered cubical complex in 2D, and in part 3D, with $\mathbb{Z}_2$ coefficients. The reason that the 3D version was not completed and included is that until a new representation of the cubes is implemented it would be hopelessly ineffective and basically useless in practice so we decided to halt the 3D project and implement a new cube representation first.

The procedures presented in this thesis are not yet ready to be applied to real problems and should be considered as first rough drafts. However, after some modifications this set of procedures should be a powerful computational tool, we just have to make them more efficient and we have several ideas to achieve this. The first thing we must do is to implement a new representation of the cubes. In particular there is no need to represent cubes as cumbersome lists of lists. As a matter of fact we could represent them collected by dimension in hashes listing only one vertex similar to the approach in [**KMM04**]. Also it is not necessary to explicitly include the boundary cubes of every cube in the representation. To get a feeling for what this can do for the effectiveness remember that each square has four sides and each side has two endpoints and in the present representation all of these components are explicitly represented as a list of lists and each of them is taken into account in calculations. In the new representation a square would be represented by only one cube and it boundaries will appear explicitly only in the precise calculation steps where it is needed. Hence instead of nine cubes we will only have one represented explicitly. Of course, even more is saved in higher dimensions, a real cube in 3-space has six square sides, each square have four sides and each side two endpoints so in the old representation a cube in 3-space would require 27 objects to be represented whereas in the new only one explicitly and the rest made to appear when/if they are needed in calculations.

When the new representation has been implemented in 2D the set of procedures should be fast and efficient enough to handle real 2D applications in image processing, pattern recognition and similar things. At present the key procedure *pairing* for persistent homology takes about 3.5 minutes to process a filtered cubical complex consisting of about $11,000$ randomly generated elementary cubes on a $200 \times 200$ pixel square. A reasonable requirement for this to be useful in practical situations is at least $100,000$ elementary cubes on an area of $1000 \times 1000$ in 2D and we see no reason to doubt that the representation change and general optimisation of the procedures for 2D will allow for more than $1,000,000$ cubes on a $2000 \times 2000$ square in reasonable time. This belief is based on the results achieved in experiments using cubical homology by Mischaikow et al. and Zomorodian et al. cf. [**Zom05**] p. 206 for persistent homology. In particular, the latter using the same approach as us but with simplicial complexes completes the pairing process on a complex consisting of more than $1,000,000$ simplices in about 10 seconds on a moderately powerful computer.

As described in chapter 4 there are also a few other general things to be done to increase the efficiency of the procedures and these will be taken care of after the representation issue. Further things that should to be done in the future are

(1) Make the 3D version complete.
(2) Try to figure out how to detect other than 0-, 1- and $(d-1)$-cycles in subcomplexes of $\mathcal{S}^d$ so that the EIA can be extended to higher dimensions. In particular finding a way to detect $(d-2)$-cycles so that we could find the Betti numbers of torsion free subcomplexes of $\mathcal{S}^4$ would be very useful.
(3) Extend to arbitrary dimension and coefficients by continuing to mimick Zomorodians' approach as presented in [**Zom05**] with suitable modifications.
(4) Free the procedures from the MAPLE environment (Ex. C, C++).
(5) Include statistical methods and develop new filtration techniques.
(6) Make it possible to vary cube size and in particular carry out a sequence of filter runs with increasing/decreasing cube size. This would give a "resolution change" filtration technique.
(7) Write a procedure that uses information about the test points and probability and distance calculations to make the filtration techniques adaptive.
(8) Implement the Morse-Smale complex algorithm [**Zom05**] p.161-170.
(9) Extend everything to maps, i.e. cubical persistent homology of maps, by modifying the static approach presented in [**KMM04**].

There are also several other approaches to problems similar to the ones that motivate persistent homology, e.g. surface reconstruction, image processing, Morse-Smale complexes etc. A few examples are

(1) Combinatorial Laplacians [**Fri96**].
(2) Statistical approach to persistence [**BK06**].
(3) Witness complexes [**CZ04**].

Of course some elements of these theories are likely to be useful to include in our venture but deciding this requires further study.

APPENDIX A

# Visualization, Test and Support Procedures

In addition to the core procedures presented in chapter 4 some support, test and plotting procedures have been created. These include

(1) Two procedures, *cubplot2D* and *cubplot3D* for plotting the cubical complex that arises from a coordinate matrix in 2 and 3 dimensions.
(2) A procedure called *cgplot* that creates an animation of the growth of the complex created by the *cgfilter* procedure.
(3) Two procedures that plot barcodes for filtered complexes in 2 dimensions. One of them, *barcodes*, plots the $\mathcal{P}$-intervals from creation index to destruction index and in the other, *barcodep*, the indices in each case have been translated so that the creation index is at 0.
(4) A small procedure, *intmk*, that converts floating point entries in a coordinate matrix that have decimal expansions consisting of only 0:s to integer type. For example the float number 9.0000 would be converted to 9. This saves a lot of time for the filtering procedures and is actually necessary to get the intended cubes, due to the construction of the *cubsim* procedure, when the coefficient matrix contains decimal entries.
(5) A test program that is used for generating plots of time vs. number of cubes for any procedure or a set of procedures that are to be compared. We use this for testing all of the essential procedures in chapter4 and in particular for comparing *bettic* and *hom*.

All procedures described in this appendix, except *intmk*, needs the MAPLE packages `plots` and `plottools` so these have to be loaded in advance. In the following the MAPLE code for these procedures and examples of how they behave is shown.

### 1. *cubplot2D*

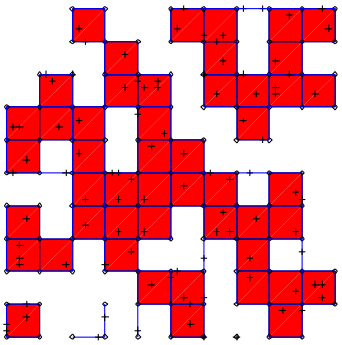
The input to *cubplot2D* is the coefficient matrix $M$ and the output is a plot of the cubical complex corresponding to $M$ and also the actual points contained in $M$ which are marked as crosses in the plot.

```
cubplot2D:=proc(M)
local i,j,l1,l2,x,y,p,q,F:
F:=filter(intmk(M));
l1:=nops(F):   l2:=LinearAlgebra[RowDimension](M):
p:=vector(l1);   q:=vector(l2);
for i from 1 to l1 do
  x:=F[i][1];   y:=F[i][2];
  if nops(x)=1 then
    if nops(y)=1 then
      p[i]:=point([x,y],color=green);
    else
      p[i]:=line([x,y[1]],[x,y[2]],color=blue);
```

```
    end if:
  else
    if nops(y)=1 then
      p[i]:=line([x[1],y],[x[2],y],color=blue);
    else
      p[i]:=polygon([[x[1],y[1]],[x[2],y[1]],
      [x[2],y[2]],[x[1],y[2]]],color=red);
    end if:
  end if:
end do:
for j from 1 to l2 do
  q[j]:=point([M[j,1],M[j,2]],symbol=CROSS,color=black);
end do:
p:=convert(p,list); q:=convert(q,list);
display(seq(p[i],i=1..nops(p)),
seq(q[j],j=1..nops(q)),axes=none);
end proc:
```

**Example cubplot2D**

```
M:=Matrix(100,2,0.200*rand(0..50)):
cubplot2D(M);
```



### 2. *cubplot3D*

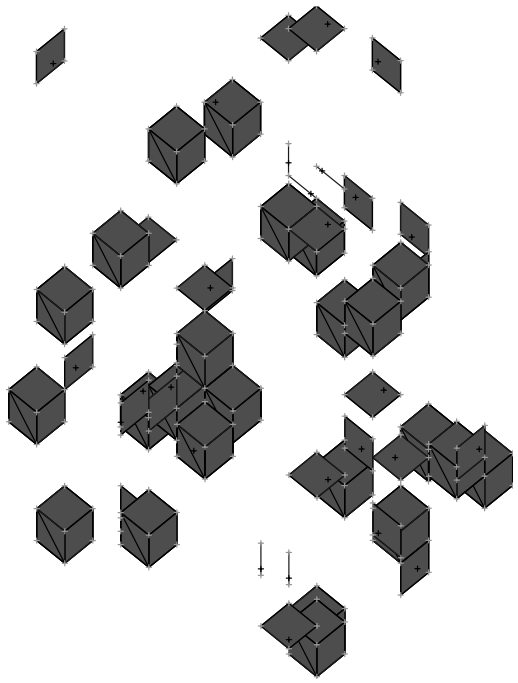
The code for *cubplot3D* is much larger but the structure is the same as in cubplot2D with the obvious extensions.

```
cubplot3D:=proc(M)
  local i,j,l1,l2,x,y,z,p,q,F:
  F:=filter(intmk(M));
  l1:=nops(F):   l2:=LinearAlgebra[RowDimension](M):
  p:=vector(l1);   q:=vector(l2);
  for i from 1 to l1 do
    x:=F[i][1];   y:=F[i][2];   z:=F[i][3];
    if nops(x)=1 then
      if nops(y)=1 then
        if nops(z)=1 then
          p[i]:=point([x,y,z],color=green);
        else
```

```
          p[i]:=line([x,y,z[1]],[x,y,z[2]],color=blue,thickness=2);
        end if:
      else
        if nops(z)=1 then
          p[i]:=line([x,y[1],z],[x,y[2],z],color=blue,thickness=2);
        else
          p[i]:=polygon([[x,y[1],z[1]],[x,y[2],z[1]],
          [x,y[2],z[2]],[x,y[1],z[2]]],color=red, transparency=0.9);
        end if:
      end if:
    else
      if nops(y)=1 then
        if nops(z)=1 then
          p[i]:=line([x[1],y,z],[x[2],y,z],color=blue,thickness=2);
        else
          p[i]:=polygon([[x[1],y,z[1]],[x[2],y,z[1]],
          [x[2],y,z[2]],[x[1],y,z[2]]],color=red, transparency=0.9);
        end if:
      else
        if nops(z)=1 then
          p[i]:=polygon([[x[1],y[1],z],[x[2],y[1],z],[x[2],y[2],z],
          [x[1],y[2],z]],color=red, transparency=0.9);
        else
          p[i]:=polygon([[x[1],y[1],z[1]],[x[1],y[1],z[2]],
          [x[1],y[2],z[1]],[x[1],y[2],z[2]],[x[2],y[1],z[1]],
          [x[2],y[1],z[2]],[x[2],y[2],z[1]],[x[2],y[2],z[2]]],
          color=yellow, transparency=0.9);
        end if:
      end if:
    end if:
  end do:
  for j from 1 to l2 do
    q[j]:=point([M[j,1],M[j,2],M[j,3]],symbol=CROSS,color=black);
  end do:
  p:=convert(p,list); q:=convert(q,list);
  display(seq(p[i],i=1..nops(p)),seq(q[j],j=1..nops(q)));
end proc:
```

**Example cubplot3D**

```
M:=Matrix(60,3,0.200*rand(0..60)):
cubplot3D(M);
```



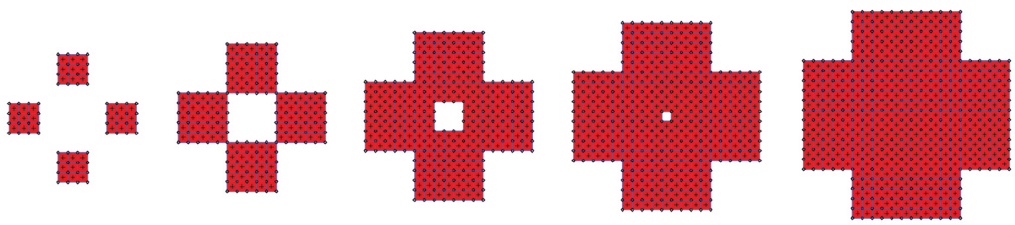
### 3. *cgplot*

The *cgplot* procedure takes as input an $r \times 2$ matrix $M$ and a positive integer $N$ which is the number of growth steps we want to use. The output is an animation of the growing cubical complex from step 1 to step $N$. The 0:th step would be the result that *cubplot2D* gives for the same matrix $M$ but this step is not included in *cgplot* for practical reasons. In example below a picture of the five frames in the corresponding animation is shown. The crosses in the plot represents the actual matrix and growth coordinates. The MAPLE code is the following.

```
cgplot:=proc(M::Matrix,N)
  local i,p,CGR;
  for i from 1 to N do
    p[i]:=cubplot(convert(cubgrow(M,N)[i],Matrix));
  end do:
  display([seq(p[j],j=1..N)],insequence=true);
end proc:
```

**Example cgplot**

```
M:=Matrix([[1.5,1.5],[6.5,6.5],[1.5,11.5],[-3.5,6.5]]);
cgplot(M,5);
```



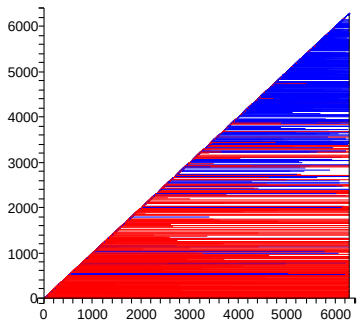
## 4. *barcodes* and *barcodep*

These two procedures takes as input a filtration $F$ of a cubical complex in 2-dimensions and plots the $\mathcal{P}$-intervals corresponding to it. The actual code is as follows.

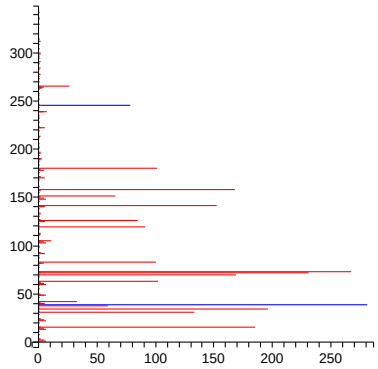
```
barcodes:=proc(F)
  local e,i,j,p,q,x,y,max,PF;
  PF:=pairing(F);
  p:=array(1..nops(PF[1])):
  q:=array(1..nops(PF[2])):
  for i from 1 to nops(PF[1]) do
    y:=PF[1][i][2][1];   x:=PF[1][i][2][2];
    p[i]:=line([y,y],[x,y],color=red);
  end do:
  for j from 1 to nops(PF[2]) do
    y:=PF[2][j][2][1];   x:=PF[2][j][2][2];
    q[j]:=line([y,y],[x,y],color=blue);
  end do:
  max:=nops(F):
  e:=line([max+1,0],[max+1,max],color=black);
  display(seq(p[i],i=1..nops(PF[1])),
  seq(q[j],j=1..nops(PF[2])),e);
end proc:
```

To get *barcodep* instead all we have to do is replace the line `q[j]:=line([y,y], [x,y],color=blue);` in *barcodes* with `q[j]:=line([0,y],[x-y,y],color=blue);`.

### Example barcodes




### Example barcodep



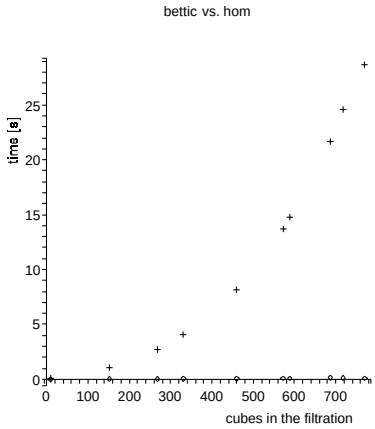

### 5. *test*

The *test* procedure can be slightly modified to produce plots of time to completion vs. number of cubes for any of our procedure. The particular one we show here is the plot compairing *bettic* and *hom*. If we wish to test other procedures we simply replace *bettic* and *hom* by their names or if we just want to test one procedure erase one of *bettic* and *hom* (and its corresponding tags and title) and replace the remaining one with the procedure to be tested. The input is $u0$ which is the number of point we want to start with, *incr* is the number of points that is added to the next test run, *steps* is the number of tests we want to do and finally $v$ is the factor in endpoint of the interval $[0, 0.200v]$ which defines the domain for the points to be chosen for each of the tests.

```
test:=proc(u0,incr,steps,v)
  local i,j,n,p,q,u,st,tb,th,F,M;
  tb:=array(1..steps);   th:=array(1..steps);
  p:=array(1..steps);   q:=array(1..steps);
  u:=u0;
  for i from 1 to steps do
    M:=Matrix(u,2,0.200*rand(0..v));
    F:=filter(intmk(M));
    n[i]:=nops(F);
    st:=time();
      bettic(F);
    tb[i]:=time()-st;
    st:=time():
      hom(F):
    th[i]:=time()-st;
    p[i]:=point([n[i],tb[i]]);
    q[i]:=point([n[i],th[i]],symbol=cross);
    u:=u+incr;
  end do:
  display(seq(p[i],i=1..steps),seq(q[j],j=1..steps),
  title="bettic vs.  hom",labels=["cubes in the filtration",
  "time[s]"],labeldirections=[horizontal,vertical]);
end proc:
```

**Example test on bettic vs. hom**

```
testbh(1,20,10,100);
```




## 6. *intmk*

The intmk procedures takes an $m \times r$ matrix and converts the float elements that have a decimal expansion consisting of only 0:s to integer type. The input is a matrix with float entries and the output is a matrix with entries of either float or integer type. The code for is as follows.

```
intmk:=proc(M::Matrix)
  local i,j,n,m;
  n:=LinearAlgebra[RowDimension](M);
  m:=LinearAlgebra[ColumnDimension](M);
  for i from 1 to n do
    for j from 1 to m do
      if frac(M[i,j])=0 then
        M[i,j]:=round(M[i,j]);
      end if:
    end do:
  end do:
  M;
  end proc:
```

APPENDIX B

# Biography of the Author

The author was born in Gävle Sweden in October 1980 and received his elementary education there culminating in the graduation from the 'gymnasium' in May 1999. Then in the autumn of 1999 he took a variety of courses at Högskolan i Gävle, in particular, elementary courses in calculus and linear algebra. This represents a personal milestone for the author since this the first time he actually saw some real mathematics. Up until this point the high regards of mathematics was mainly based on hearsay and the subject unfortunately disregarded. Before this, attention was focused on other sciences, in particular medicine, physics and psychology as well as intense social activity. Including a seven years long pseudo-career as a disc jockey. In the spring of 2002 the author and his lovely girl friend moved to Luleå and he enrolled in a general engineering program while she chose to attend a newly created so called Arena. In the autumn of 2002 the author, as planned, chose to continue his studies in the Engineering physics program. The reason for choosing engineering, and in particular engineering physics, was uncertainty as to whether a career in mathematics, theoretical physics or possibly something else was what he wanted to pursue. The main reason for attending Ltu, instead of some other university, was the fact that at the time the Engineering physics program at Ltu was the only one that, at least explicitly, had a specialization towards mathematics. The years at Ltu has been dominated by mathematics, both formally in taking university courses and personal investigations, since early on the author realized that simply taking the available courses is not sufficient to acquire enough knowledge of mathematics. At present the author has realized that simply taking every available course and using all spare time to explore mathematics is not sufficient to attain the goal of enough knowledge. To compensate for this miscalculation the author has decided to allocate more time on his long time non-scientific interests of oil-painting and drawing. Besides formal courses at Ltu the author has taken some mathematics courses at Uppsala University, in particular, a course on Fuchsian groups and Automorphic functions which represents the author's second mathematical milestone. During the summer and autumn of 2005 the author wrote a Bachelor's thesis entitled *Mathematics and Mathematical Arguments -an overview of the logical structure, heuristics and applications of mathematical reasoning* and in May of 2006 the author, using this thesis, obtained a Bachelor's degree in mathematics. After this came a period of interest in computational mathematics and it culminated with the present Master thesis where this interest was combined with the interest in algebraic topology.

# References


[AMT01] M. Allili, K. Mischaikow, and A. Tennenbaum. Cubical homology and the topological classification of 2d and 3d imagery. *Proc. IEEE International Conference on Image Processing*, 2:173–176, 2001.

[BK06] P. Bubenik and P.T. Kim. A statistical approach to persistent homology (manuscript). *http://www.ii.uj.edu.pl/ mrozek/papers/homology-algorithm-based-on-acyclic-subspace6.pdf*, 2006.

[C+01] T.H. Cormen et al. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[C+04] G. Carlsson et al. Persistence barcodes for shapes. *Proc. of the Symposium on Geom. Processing*, pages 127–138, 2004.

[CCdS06] E. Carlsson, G. Carlsson, and V. de Silva. An algebraic topological method for feature identification. *International Journal of Computational Geometry and Applications*, 16(4):291Ű314, 2006.

[CE73] H. Cartan and S. Eilenberg. *Homological Algebra*. Princeton University Press, 1973.

[CGPZ05] F. Cazalas, J. Geisen, M. Pauly, and A. Zomorodian. Conformal alpha shapes. *Proc. Sympos. Point-Based Graphics*, 2005.

[CIdSZ06] G. Carlsson, T. Ishkhanov, V. de Silva, and A. Zomorodian. On the local behavior of spaces of natural images (preprint). *http://math.stanford.edu/comptop/preprints/mumford.pdf*, 2006.

[CZ04] G. Carlsson and A. Zomorodian. Computing persistent homology. *Proc. 20th Ann. ACM Sympos. Comput. Geom.*, pages 247–356, 2004.

[CZCG06] A. Collins, A. Zomorodian, G. Carlsson, and L. Guibas. A barcode shape descriptor for curve point cloud data. *Computers and Graphics, to appear*, 2006.

[DE95] C.J.A. Delfinado and H. Edelsbrunner. An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere. *Computer Aided Geom. Design*, 12:771–784, 1995.

[DG04] T.K. Dey and S. Goswami. Provable surface reconstruction from noisy samples. *Proc. 20th Ann. Symposium Comput. Geom.*, pages 330–339, 2004.

[Ede95] H. Edelsbrunner. The union of balls and its dual shape. *Discrete Comput. Geom.*, 13:415–440, 1995.

[EFFL95] H. Edelsbrunner, M. Facello, P. Fu, and J. Liang. Measuring proteins and voids in proteins. *Proc. 28th Ann. Hawaii Internat. Conf. System Sci.*, pages 256–264, 1995.

[EFL96] H. Edelsbrunner, M. A. Facello, and J. Liang. On the definition and the construction of pockets in macromolecules. *Proc. Pacific Sympos. Biocomputing*, 1996.

[EKS83] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory*, 29(4):551–559, 1983.

[ELZ02] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete Comput. Geom.*, 28:511–533, 2002.

[EM94] H. Edelsbrunner and E.P. Mucke. Three-dimensional alpha shapes. *ACM Trans. Graphics*, 13:43–72, 1994.

[Fri96] J. Friedman. Computing betti numbers via combinatorial laplacians. *Proc. 28th ann. ACM Symp. Theory of Comput.*, pages 386–391, 1996.

[GKM04] M. Gameiro, W.D. Kalies, and K. Mischaikow. Topological characterization of spatial-temporal chaos. *Physical Review E*, 70, 2004.

[GMW05] M. Gameiro, K. Mischaikow, and T. Wanner. Evolution of pattern complexity in the cahn-hilliard theory of phase separation. *Acta Materialia*, 53(3):693–704, 2005.

[Hat01] A. Hatcher. *Algebraic Topology*. Cambridge University Press, Cambridge, UK, 2001.

[KGMS04] K. Krishan, M. Gameiro, K. Mischaikow, and M. F. Schatz. Homological characterizations of spiral defect chaos in rayleigh-benard convection (preprint), 2004.

[KMM04] T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*. Springer-Verlag, New York, USA, 2004.

[Lan02] S. Lang. *Algebra*. Springer-Verlag, New York, USA, 3rd revised edition, 2002.

[Mea02] K. Mischaikow and M. Niethammer et al. Analysis of blood vessel topology by cubical homology. *Proc. 2002 IEEE International Conference on Image Processing*, 2:969–972, 2002.

[MPZ06] M. Mrozek, P. Pilarczyk, and N. Zalazna. Homology algorithm based on acyclic subspace (manuscript). *http://www.ii.uj.edu.pl/ mrozek/papers/homology-algorithm-based-on-acyclic-subspace6.pdf*, 2006.

[Spa66] E.H. Spanier. *Algebraic Topology*. Springer-Verlag, New York, USA, 1966.

[Vic94] J.W. Vick. *Homology Theory*. Springer-Verlag, New York, USA, 2nd edition, 1994.

[Zom05] A.J. Zomorodian. *Topology for Computing*. Cambridge University Press, Cambridge, UK, 2005.

# Index