State-of-the-art methods in the computation of persistent homology

Ben Napier
Supervised by: Jeffrey Giansiracusa

April 29, 2022

Plagarism declaration

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Contents

Chapter 1

Introduction

Topological data analysis (TDA) is an emerging field of data analysis, with its mathematical foundations within algebraic topology. It is an approach to the analysis of datasets which exploits the underlying topological structure of data, and provides a general framework for extracting information from high-dimensional and noisy datasets. TDA has provided significant insight into the study of data in multiple applications, such as: the spread of infectious disease, cancer, proteins, and information networks [1].

Persistent homology is the main tool within topological data analysis, which combines rich theory from algebraic topology with the analysis of point-cloud data (and other forms of data). Persistent homology was first introduced by Edelsbrunner, Letscher, and Zomorodian [2], although many precursors exist.

To find the persistent homology of some *space* (in the case of point-cloud data, we have a metric space), the space must first be represented as a *filtered simplicial complex*. The persistent homology of such a complex is typically given as *persistent barcodes*.

Within this work, we primarily aim to explore various algorithms for the two main steps in the persistent homology pipeline:

(i) construction of a filtered simplicial complex (Section 4.1); and

(ii) computation of the corresponding persistent barcodes (Section 4.2).

Within Chapter 4, we introduce algorithms for the two steps above, analyse their theoretical running times, and conduct experiments to compare their real-world running times. For (ii), we focus mainly on the methods implemented by the current prevalent persistent homology package, *Ripser* [3]. These experiments required significant programming, building up data structures for many of the algebraic structures we will introduce. The source code for this project can be found in Appendix C.

To facilitate the exposition of these algorithms, Chapter 2 builds foundational knowledge in algebraic topology and combinatorics, formalising notions such as *filtered simplicial complexes* and *persistent homology*. Chapter 3 aims to familiarise the reader with the theory of computation, formalising notions such as *problems* and *algorithms*.

For readers unfamiliar with topology, Appendix A provides a more elementary introduction. Furthermore, for interested readers, Appendix B provides some supplementary material on the theory of computation, delving into more rigorous definitions of *Turing machines* and *computational complexity*.

In Chapter 5, we also touch on some applications and how persistent barcodes may be inter-

preted, as well as introducing some novel applications. But note, this is not the main focus of our work.

Chapter 2

Background in topology and combinatorics

In this chapter, we provide some needed background in topology and combinatorics in order to understand fast constructions of the Vietoris-Rips complex, and the persistent homology algorithm and the speed-ups utilised in Ripser [3].

For readers unfamiliar with topology, with a key interest in having some context before reading, see Appendix A.

2.1 Persistence modules

This section serves to build a single algebraic structure called a *persistence module*, for which we can describe the concept of *persistence*, but to do so we require tools from *category theory* and *module theory*. We also touch on cohomology, as it is needed for a speed-up in the persistent homology algorithm.

We move to form a bijection between the isomorphism classes of *persistence modules* of finite type over a field and the finite sets of $\mathcal{P}$-intervals.

2.1.1 A brief touch on category theory

Category theory has the ambitious goal to generalize all of mathematics in terms of *categories*, irrespective of the underlying structure of such objects.

Definition 2.1.1 (Category)**.** A *category* C consists of

(i) a class $\mathrm{ob}(C)$ of *objects*;

(ii) a class $\hom(C)$ of *morphisms* between objects;

(iii) a *domain* class function $\mathrm{dom} : \hom(C) \to \mathrm{ob}(C)$;

(iv) a *codomain* class function $\mathrm{cod} : \hom(C) \to \mathrm{ob}(C)$; and

(v) for every $f, g \in \hom(C)$ such that $\mathrm{cod}(f) = \mathrm{dom}(g)$, a morphism denoted $g \circ f$ (or gf) called their *composite*

such that for all $f, g, h \in \hom(C)$ and $x, y \in \mathrm{ob}(C)$,

(i) if $\mathrm{cod}(f) = \mathrm{dom}(g)$, then $\mathrm{dom}(g \circ f) = \mathrm{dom}(f)$ and $\mathrm{cod}(g \circ f) = \mathrm{cod}(g)$;

(ii) there exists a morphism $\mathrm{id}_x \in \hom(C)$ (called the *identity morphism*) such that $\mathrm{dom}(\mathrm{id}_x) = x$ and $\mathrm{cod}(\mathrm{id}_x) = x$;

(iii) if $\mathrm{cod}(f) = \mathrm{dom}(g)$ and $\mathrm{cod}(g) = \mathrm{dom}(h)$, then $(h \circ g) \circ f = h \circ (g \circ f)$; and

(iv) if $\mathrm{dom}(f) = x$ and $\mathrm{cod}(f) = y$, then $\mathrm{id}_y \circ f = f \circ \mathrm{id}_x$.

Note here that there is little in the way of description of what is meant by an *object* and a *morphisms*, which is the purpose at this level of abstraction.

Example 2.1.2.

(i) The category of *sets and functions*, denoted as Set, is a valid category.

(ii) The category of *abelian groups and group homomorphisms*, denoted Ab, is another valid category. It is a *subcategory* of the category of *groups and group homomorphisms*, denoted Grp.

Definition 2.1.3 (Isomorphism)**.** Let C be a category and $x, y \in \mathrm{ob}(C)$. A morphism $f : \hom(x, y)$ is an *isomorphism* if there exists a morphism $f^{-1} \in \hom(y, x)$ such that $f \circ f^{-1} = \mathrm{id}_y$ and $f^{-1} \circ f = \mathrm{id}_x$. We call f^{-1} the *inverse* of f.

This gives us a generalisation of isomorphisms as we know them for groups, graphs, vector spaces, topology, etc.

We now define a mapping *between* categories. For a category C and $x, y \in \mathrm{ob}(C)$, $\hom(x, y)$ denotes a subclass of of $\hom(C)$ such that, if $f \in \hom(x, y)$, then $\mathrm{dom}(f) = x$ and $\mathrm{cod}(f) = y$. Denote such a morphism by $f : x \to y$.

Definition 2.1.4 (Functor)**.** Let C and D be categories. A *functor* $F : C \to D$ from C to D is a mapping that

(i) associates each $x \in \mathrm{ob}(C)$ to a $F(x) \in \mathrm{ob}(D)$;

(ii) associates each $f \in \hom(C)$ to a $F(f) \in \hom(D)$ such that

  (a) $\mathrm{dom}(F(f)) = F(\mathrm{dom}(f))$ and $\mathrm{cod}(F(f)) = F(\mathrm{cod}(f))$;

  (b) for every $x \in \mathrm{ob}(C)$, we have $F(\mathrm{id}_x) = \mathrm{id}_{F(x)}$; and

  (c) for all morphism $f, g \in \hom(C)$ such that $\mathrm{cod}(f) = \mathrm{dom}(g)$, we have $F(g \circ f) = F(g) \circ F(f)$.

Functors between categories must preserve identity morphisms and composition of morphisms.

Example 2.1.5 (Examples of functors)**.**

(i) For any category C, we may define the *identity endofunctor* (an endofunctor is functor that maps from a category to itself) that maps all objects to itself and all morphisms to itself, we denote this functor id_C.

(ii) We may define *homology* as a *contravariant* functor from the category of chain complexes to the category of abelian groups (or modules, which we will later see). A contravariant functor is defined in the same way as a regular functor, except for the fact that they *reverse composition* and switches the domain and codomain; that is $F(g \circ f) = F(f) \circ F(g)$ (in (ii)(b) above) and if $f : x \to y$, then $F(f) : F(y) \to F(x)$.

(iii) Another interesting example (which we do not go into details here) is the abelianization functor mapping from the category of groups to the category of abelian groups. The abelianization of some group G is the quotient of G by the relation $xy = yx$ for all

$x, y \in G$, and the construction of this functor is as one would expect (using the canonical quotient map).

Proposition 2.1.6. *Functors preserve isomorphisms.*

Proof. Let C and D be categories and $F : C \to D$ a functor. Let $f \in \hom(C)$ be an isomorphism with inverse f^{-1}. Then

$$F(f) \circ F(f^{-1}) = F(f \circ f^{-1}) = F(\mathrm{id}_y) = \mathrm{id}_{F(y)}$$

and

$$F(f^{-1}) \circ F(f) = F(f^{-1} \circ f) = F(\mathrm{id}_x) = \mathrm{id}_{F(x)},$$

thus $F(f) \in \hom(D)$ is an isomorphism. □

Definition 2.1.7 (Natural isomorphism)**.** Let F and G be functors between categories C and D. A *natural transformation* $\eta = \{\eta_x\}_{x \in C}$ from F to G is a family of morphisms such that $\eta_x : F(x) \to G(x)$ and the following diagram commutes for all $f : x \to y$ in C.

$$\begin{array}{ccc} F(x) & \xrightarrow{F(f)} & G(y) \\ {\scriptstyle \eta_x}\downarrow & & \downarrow{\scriptstyle \eta_y} \\ G(x) & \xrightarrow[G(f)]{} & G(y) \end{array}$$

If η_x is an isomorphism for every $x \in \mathrm{ob}(C)$, then η is said to be a *natural isomorphism*. If such an η exists, F and G are said to be naturally isomorphic, denoted $F \simeq G$.

Definition 2.1.8 (Equivalence of categories)**.** Let C and D be categories. An *equivalence* of two categories is a pair of functors $F : C \to D$ and $G : D \to C$ alongside two natural isomorphisms $F \circ G \simeq \mathrm{id}_D$ and $G \circ F \simeq \mathrm{id}_C$.

If such an equivalence of categories exists between categories C and D, then C and D are said to be *equivalent*, denoted $C \simeq D$, and in the definition above $F : C \to D$ (or $G : D \to C$) are said to define an equivalence of categories. Equivalence of categories is more analogous to *homotopy equivalence* from homotopy theory then it is to homeomorphic; however, this is not a perfect analogy.

Equivalence of categories do not present particularly strong correspondence between objects within each category; however, it does provide a bijection between the isomorphisms within each category.

Corollary 2.1.9. *Let F be a functor which defines an equivalence between categories C and D. Then $f \in \hom(C)$ is an isomorphism if and only if $F(f) \in \hom(D)$ is an isomorphism.*

2.1.2 On modules and their structure

Here we briefly touch on *module theory* and a fundamental result from abstract algebra on the structure of finitely generated modules over principal ideal domain.

A module is a generalization of a vector spaces, but we replace the field of scalars with a ring.

Definition 2.1.10 (Module)**.** Let R be a commutative ring with multiplicative identity 1. A *R-module* M consists of an abelian group $(M, +)$ and an operation $\cdot : R \times M \to M$ such that for all $r, s \in R$ and $m, n \in M$, we have

(i) $(r + s) \cdot m = r \cdot m + s \cdot m$;

(ii) $(rs) \cdot m = r \cdot (s \cdot m)$;

(iii) $r \cdot (m + n) = r \cdot m + r \cdot n$; and

(iv) $1 \cdot m = m$.

The use of studying modules is vast; we may study some object by making it into a module over some ring with *well-behaved* properties, giving us insight into the structure the original object. We note that we can define modules for a non-commutative ring R, but in this case we define *left R-modules* and *right R-modules* (when R is commutative, these coincide). We will be dealing with commutative rings only.

Example 2.1.11 (Examples of modules)**.**

(i) Every abelian group is a $\mathbb{Z}$-module (and in fact this correspondence is bijective). Thus every ring is a $\mathbb{Z}$-module.

(ii) Every ideal of a commutative ring R is an R-module.

(iii) Every quotient ring of a commutative ring R is an R-module.

(iv) Every polynomial ring over a commutative ring R is a R-module.

We define the morphisms in the category of modules as one would expect.

Definition 2.1.12 (Module homomorphism)**.** Let R be a commutative ring and M and N be R-modules. A function $f : M \to N$ is a *R-module homomorphism* (or *R-linear map*) if for all $m, n \in M$ and $r \in R$, $f(m + n) = f(m) + f(n)$ and $f(rx) = rf(x)$.

The category of *modules and module homomorphisms* indeed forms a category, which is easy to check.

A module homomorphism is called a *module isomorphism* if it is a bijection, and if there is such a homomorphism between two modules then they are said to be *isomorphic* (denoted with the typical $\cong$).

Example 2.1.13 (Examples of module homomorphisms)**.**

(i) For a commutative ring R and R-modules M and N, the zero map $M \to N$ is a module homomorphism.

(ii) Let R be a commutative ring, then $R[x]$ is a ring (and thus a R-module). We define $f : R[x] \to R[x]$ where $f(p(x)) \mapsto xp(x)$. This is indeed a module homomorphism; however, it is not a ring homomorphism as

$$f((x)(x)) = x^3 \neq x^4 = f(x)f(x).$$

We now introduce *exact sequences*, which are useful tools in algebraic topology. They also motivate our definition of *homology*.

Definition 2.1.14 (Exact sequence)**.** A sequence of modules and homomorphisms

$$M_0 \xrightarrow{f_1} M_1 \xrightarrow{f_2} M_2 \xrightarrow{f_3} \dots \xrightarrow{f_n} M_n$$

is said to be *exact* if $\operatorname{im} f_i = \ker f_{i+1}$ for all $i \in \{1, \dots, n-1\}$.

Example 2.1.15 (Examples of exact sequences)**.**

(i) Consider the sequence

$$0 \to \mathbb{Z} \xrightarrow{\cdot p} \mathbb{Z} \to \mathbb{Z}/p \to 0$$

where the map $\mathbb{Z} \to \mathbb{Z}/p$ is the quotient map. This sequence is exact. In fact, this can be generalised to the sequence

$$0 \to A \xrightarrow{f} B \to B/f(A) \to 0$$

where f is injective.

(ii) Consider the sequence

$$0 \to A \to B.$$

This sequence is exact if and only if the map $A \to B$ is injective. Now consider the sequence

$$A \to B \to 0.$$

This sequence is exact if and only if the map $A \to B$ is surjective. Thus if the sequence

$$0 \to A \to B \to 0$$

is exact if and only if the map $A \to B$ is a bijection.

We borrow the notion of *finitely generated* from linear algebra and apply this to modules in the way you would expect, but we benefit from the following algebraic definition.

Definition 2.1.16 (Finitely generated module)**.** Let R be a commutative ring and M be a R-module. M is *finitely generated* if there is an exact sequence $R^p \to M \to 0$ for some $p \in \mathbb{N}_0$. Additionally, M is *finitely presented* if there exists an exact sequence $R^q \to R^p \to M \to 0$ for some $p, q \in \mathbb{N}_0$.

We now introduce the notion of *grading*, which is a decomposition of a ring or module.

Definition 2.1.17 (Graded ring)**.** A *graded ring* is a ring R equipped with a direct sum decomposition of abelian groups $R \cong \bigoplus_{i \in \mathbb{N}_0} R_i$ such that $R_m R_n \subset R_{m+n}$ for all $m, n \in \mathbb{N}_0$.

We similarly define the grading of a module.

Definition 2.1.18 (Graded module)**.** Let $R \cong \bigoplus_{i \in \mathbb{N}_0} R_i$ be a graded commutative ring and M a R-module. M is a *graded module* if there is a direct sum of abelian groups $M \cong \bigoplus_{i \in \mathbb{N}_0} M_i$ such that $R_m M_n \subset M_{m+n}$ for all $m, n \in \mathbb{N}_0$.

We note here that our gradation is a decomposition over $\mathbb{N}_0$, which may be referred to as a $\mathbb{N}_0$-*grading*. Other texts may choose $\mathbb{Z}$-grading as the standard, for which we would call the above a *non-negatively graded ring* or *module*.

Example 2.1.19 (Examples of graded rings)**.**

(i) For any ring R, we have the trivial grading

$$R = R_0 \oplus \bigoplus_{i=1}^{\infty} 0.$$

(ii) Let R be a commutative ring. Then the polynomial ring is graded by its degree:

$$R[x] = \bigoplus_{i \in \mathbb{N}_0} \{rx^i : r \in R\}.$$

We now present the standard structure theorem for finitely generated modules over a principal ideal domain.

Theorem 2.1.20. *Let R be a principal ideal domain and M a finitely generated R-module. Then there is a unique decomposition*

$$M \cong R^{\beta} \oplus \bigoplus_{i=1}^{m} R/(d_i) \tag{2.1}$$

where $d_i \in R$ such that $d_1 \mid d_2 \mid \ldots \mid d_m$ and $\beta \in \mathbb{Z}$.

Sketch of proof. As R is a principal ideal domain, it is a Noetherian ring in which the concepts of finitely generated and finitely presented coincide. Thus, M is finitely presented. That is, there exists an exact sequence $R^q \to R^p \to M \to 0$ where $p, q \in \mathbb{N}$. We take A as a presentation matrix isomorphic to the morphism $R^q \to R^p$. Then SNF A yields the required decomposition, where diagonal entries correspond the to d_i's. □

SNF A above denotes the Smith normal form of the matrix A. A similar result holds for the graded case.

Theorem 2.1.21. *Let R be a graded principal ideal domain and M a finitely generated graded R-module. Then there is a unique decomposition*

$$M \cong \left(\bigoplus_{i=1}^{n} \Sigma^{\alpha_i} R \right) \oplus \left(\bigoplus_{i=1}^{m} \Sigma^{\gamma_i} R/(d_i) \right) \tag{2.2}$$

where $d_i \in R$ such that $d_1 \mid d_2 \mid \ldots \mid d_m$, $\alpha_i, \gamma_i \in \mathbb{Z}$, and Σ^k denotes a k-shift upward in grading.

A proof for this mimics that for Theorem 2.1.20, for which there exists a variant of the Smith normal form algorithm for graded principal ideal domains [4].

We have thus established a resemblance of the structure of finitely generated modules and finitely generated graded modules to that of vector spaces, but with some finite size *torsional* portion.

We will use this structure theorem to form our bijection between the isomorphism classes of persistent modules of finite type and $\mathcal{P}$-intervals.

2.1.3 Persistence modules

We now describe *persistent modules*. Here we will define notions such as *chain complexes*, *persistence complexes*, and *persistence modules* as non-negative (over $\mathbb{N}_0$), but similar constructions exist over $\mathbb{Z}$.

Definition 2.1.22 (Chain complex)**.** Let R be a commutative ring. A *chain complex* $C = (C_*, \partial_*)$ is a sequence of R-modules $\{C_i\}_{i \in \mathbb{N}_0}$ (called *chain groups*) connected by module homomorphism $\{\partial_i\}_{i \in \mathbb{Z}}$ with $\partial_i : C_i \to C_{i-1}$ such that $\partial_i \circ \partial_{i+1} = 0$ for all $i \in \mathbb{Z}$. We may write a complex as follows.

$$\ldots \xleftarrow{\partial_0} C_0 \xleftarrow{\partial_1} C_1 \xleftarrow{\partial_2} C_2 \xleftarrow{\partial_3} \ldots$$

We can, in fact, consider a category of chain complexes. To do this, we must define what a morphism of chain complexes is.

Definition 2.1.23 (Chain map)**.** Let $C = (C_*, \partial_*^C)$ and $D = (D_*, \partial_*^D)$ be chain complexes. A *chain map* is a sequence of homomorphisms $f = \{f_i\}_{i \in \mathbb{N}_0}$ such that $f_i : C_i \to D_i$ and the following diagram commutes for all $i \in \mathbb{N}_0$.

$$\begin{array}{ccc} C_i & \xleftarrow{\partial^C_{i+1}} & C_{i+1} \\ {\scriptstyle f_i}\downarrow & & \downarrow{\scriptstyle f_{i+1}} \\ D_i & \xleftarrow[\partial^D_{i+1}]{} & D_{i+1} \end{array}$$

Lemma 2.1.24. *Let R be a commutative ring. Then chain complexes of R-modules with chain maps form a category (denoted by Ch_R).*

Definition 2.1.25 (Homology)**.** Let $C = (C_*, \partial_*)$ be a chain complex. An element $c \in C_n$ is called an *n-chain*. An n-chain that is in the kernel of ∂_n is called an *n-cycle*, and an n-chain that is in the image of ∂_{n+1} is called an *n-boundary*. The *nth homology group* of C, denoted as $H_n(C)$, is the group of n-cycles modulo the group of n-boundaries; that is,

$$H_n(C) = \frac{\ker \partial_n}{\operatorname{im} \partial_{n+1}}.$$

This is well-defined, as $\operatorname{im} \partial_{i+1} \subset \ker \partial_i$. Homology can be thought of as a measure of how far a chain complex is from being exact; a chain complex is exact (or *acyclic*) if and only if all homology groups are zero.

Example 2.1.26. We take the time to work through an example of computing homology. Consider the chain complex C:

$$0 \xrightarrow{0_1} \mathbb{Z} \xrightarrow{\partial_2} \mathbb{Z} \xrightarrow{\partial_1} \mathbb{Z}^2 \xrightarrow{0_2} 0$$

where $\partial_2(x) = px$ and $\partial_1(x) = (x, -2x)$ for some $p \in \mathbb{N}$. We note that $C_n = 0$ for all $n \geq 3$, thus $H_n(C) = 0$ for all $n \geq 3$. We first look at the matrix representations of the boundary maps, picking canonical basis:

$$M_{\partial_1} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \qquad M_{\partial_2} = \begin{pmatrix} p \end{pmatrix}.$$

From this, we perform suitable (invertible) column and row operations to get the image and kernels:

$$\begin{aligned} \operatorname{im} \partial_1 &= \mathbb{Z} & \operatorname{im} \partial_2 &= p\mathbb{Z} \\ \ker \partial_1 &= \mathbb{Z} & \ker \partial_2 &= 0. \end{aligned}$$

Thus we get the following homology groups:

$$\begin{aligned} H_0(C) &= \frac{\ker 0_2}{\operatorname{im} \partial_1} = \mathbb{Z}, \\ H_1(C) &= \frac{\ker \partial_1}{\operatorname{im} \partial_2} = \mathbb{Z}/p, \\ H_2(C) &= \frac{\ker \partial_2}{\operatorname{im} 0_1} = 0. \end{aligned}$$

We denote the sequence of homology groups by $H_*(C)$.

Lemma 2.1.27. *Homology is a functor from the category of chain complexes of R-modules with chain maps to the category of R-modules with module homomorphisms.*

Proof. It is not difficult to see that the homology of a chain complex of R-modules also defines an R-module. To finish this proof, we need to do define a morphism between homology modules. Let $C = (C_*, \partial_*)$ and $D = (D_*, \partial_*)$ be two chain complexes of R modules and $f : C_* \to D_*$ be a chain map. Then we define $f_* : H_*(C) \to H_*(D)$ by $[c] \mapsto [f(c)]$. This gives us a *contravariant functor*, which can be considered a regular functor by considering the dual category in the domain of the functor. □

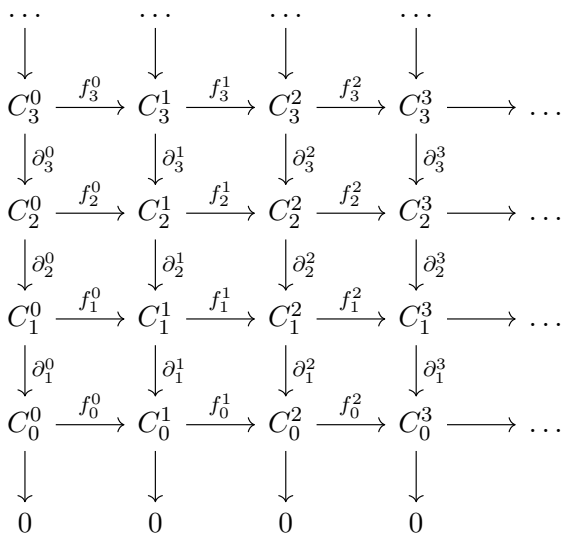


Figure 2.1.1: A portion of a persistence complex.

Definition 2.1.28 (Persistence complex)**.** A *persistence complex* is a sequence of chain complexes and chain maps $\mathcal{C} = \{(C^i, f^i)\}_{i\in\mathbb{N}_0}$ over a commutative ring R, where $C^i = (C^i_*, \partial^i_*)$ and $f^i : C^i_* \to C^{i+1}_*$.

To illustrate this definition, one may draw a commutative diagram akin to Figure 2.1.1. For a persistence complex $\mathcal{C} = \{(C^i, f^i)\}_{i\in\mathbb{N}_0}$, we define its homology as one may expect, $H_*(\mathcal{C}) = \{(H_*(C^i), f^i_*)\}_{i\in\mathbb{N}_0}$ where f^i_* is the standard induced map on homology.

Definition 2.1.29 (Persistence module)**.** A *persistence module* is a sequence of R-modules and module homomorphisms $\mathcal{M} = \{(M^i, \varphi^i)\}_{i\in\mathbb{N}_0}$ over a commutative ring R, where $\varphi^i : M^i \to M^{i+1}$.

It is an immediate consequence of Lemma 2.1.27 that the homology of a persistence complex is a persistence module.

Definition 2.1.30 (Finite type persistence complex)**.** A persistence complex $\mathcal{C} = \{(C^i, f^i)\}_{i\in\mathbb{N}_0}$ over a commutative ring R is of *finite type* if each C^i is finitely generated and there is some $m \in \mathbb{N}_0$ such that f^i is an isomorphism for all $i \geq m$.

We similarly define *persistence modules of finite type*.

We aim to apply the decomposition in Equation 2.2 to our persistence module, to motivate the following construction.

Definition 2.1.31 (Graded construction of a persistence module)**.** Let $\mathcal{M} = \{(M^i, \varphi^i)\}_{i\in\mathbb{N}_0}$ be a persistence module over a commutative ring R. We equip $R[t]$ with the standard degree grading and define a graded module over $R[t]$ by

$$\alpha(\mathcal{M}) = \bigoplus_{i\in\mathbb{N}_0} M^i$$

where the R-module structure is the sum of the individual components and the action of t is given by $tm^i = \varphi^i(m^i)$.

Multiplication by t can be thought of as an upwards shift in the gradation.

For a commutative ring R, α may be considered a functor between the category of persistent modules of finite type over R (the objects and morphisms are clear here) and the category of

finitely generated graded modules over $R[t]$ (where our morphisms all correspond to multiplication by t), and in fact gives us an *equivalence* of these categories.

Theorem 2.1.32 (ZC-Representation Theorem [5])**.** *Let R be a commutative ring. α defines an equivalence of categories between the category of persistence modules of finite type over R and the category of finitely generated graded modules over $R[t]$.*

The proof for this can be found the Artin-Rees theory section from Eisenbud [6].

By Corollary 2.1.9, we have established a bijective correspondence between the isomorphism classes of persistence modules of finite type over R and isomorphism classes of finitely generated graded $R[t]$-modules.

This correspondence does not provide particular assistance in decomposing persistence modules when R is not a field; the classification of modules over $\mathbb{Z}[t]$ is complicated. Although, by setting $R = F$ for some field F, we get a particularly succinct decomposition. $F[t]$ is a graded principal ideal domain and its only graded ideals are of the form (t^n), for $n \in \mathbb{N}_0$. Thus, applying Theorem 2.1.21 we get

$$M \cong \left(\bigoplus_{i=1}^{n} \Sigma^{\alpha_i} F[t]\right) \oplus \left(\bigoplus_{i=1}^{m} \Sigma^{\gamma_i} F[t]/(t^{n_i})\right) \tag{2.3}$$

where $\alpha_i, \gamma_i \in \mathbb{Z}$, and Σ^k denotes a k-shift upward in grading (note that coincides with multiplication by t^k in our case), for some finitely generated graded $F[t]$-module M.

We now form a bijection between this decomposition to the positive real intervals.

Definition 2.1.33 ($\mathcal{P}$-interval)**.** A $\mathcal{P}$-interval is a tuple $(i, j) \in \mathbb{N}_0 \times \mathbb{N}^\infty$ such that $i < j$.

For the above definition, we define $\mathbb{N}^\infty = \mathbb{N} \cup \{+\infty\}$.

Proposition 2.1.34. *Let $\mathcal{S}$ be the set of $\mathcal{P}$-intervals and $\mathcal{F}$ the set of isomorphism classes of finitely generated graded $F[t]$-modules. Define $Q' : \mathcal{S} \to \mathcal{F}$ by $(i, j) \mapsto \Sigma^i F[t]/(t^{j-i})$ and $(i, +\infty) \mapsto \Sigma^i F[t]$. Then the map $Q : \mathcal{P} \to \mathcal{F}$ defined by*

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

where $\mathcal{P}$ is the set of finite multisets containing elements of $\mathcal{S}$, is a bijection.

Proof. Let M be a graded $F[t]$-module. Then by Equation 2.3,

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

where $\alpha_i, \gamma_i \in \mathbb{Z}$, and Σ^k denotes a k-shift upward in grading. We then define $Q^{-1} : \mathcal{F} \to \mathcal{P}$ by

$$Q^{-1}(M) = \{(\alpha_i, \infty) : i = 1, \ldots, n\} \cup \{(\gamma_i, n_i + \gamma_i) : i = 1, \ldots, m\}.$$

It is then straight forward to check that $Q \circ Q^{-1} = \mathrm{id}_{\mathcal{F}}$ and $Q^{-1} \circ Q = \mathrm{id}_{\mathcal{P}_{<\infty}(\mathcal{S})}$. □

Corollary 2.1.35. *There is a bijection between the isomorphism classes of persistence modules of finite type over a field F and the finite multisets of $\mathcal{P}$-intervals.*

We have now established that every (isomorphism class of a) persistence module has a one-to-one correspondence to a multiset of $\mathcal{P}$-intervals.

Definition 2.1.36 (Persistent barcodes)**.** Let $\mathcal{C}$ be a persistence complex with persistence module $H_*(\mathcal{C})$. Then the *persistence barcodes* of $H_*(\mathcal{C})$, denoted $\text{Pers}(H_*(\mathcal{C}))$ are precisely the corresponding $\mathcal{P}$-intervals as defined above.

Barcodes of the form (i, j) are given by the torsional portion of the decomposition of a persistence module, while barcodes of the form (i, ∞) are given by the free portion of a persistence module.

The standard algorithm for persistent homology (which we present in Chapter 4) aims to, given a persistence complex (namely a filtered Vietoris-Rips complex), compute the persistent barcodes where we take $F = \mathbb{Z}/2$.

2.1.4 An aside on cohomology

Cohomology is a dual concept to homology, and is particularly useful in algebraic topology; it may be given a ring structure with the cup product. We will not delve into the details of this, but elementary understanding is needed for a particular speed-up to the persistent homology algorithm.

Definition 2.1.37 (Cochain complex)**.** Let R be a commutative ring. A *cochain complex* $C = (C^*, \delta^*)$ is a sequence of R-modules $\{C_i\}_{i\in\mathbb{N}_0}$ connected by module homomorphism $\{\delta^i\}_{i\in\mathbb{N}_0}$ with $\delta^i : C_i \to C_{i+1}$ such that $\partial_i \circ \partial_{i-1} = 0$ for all $i \in \mathbb{Z}$. We may write a cochain complex as follows.

$$\ldots \xrightarrow{\partial_0} C_0 \xrightarrow{\partial_1} C_1 \xrightarrow{\partial_2} C_2 \xrightarrow{\partial_3} \ldots$$

Cochain complexes closely resemble chain complexes, but we flip the direction of the boundary map.

Definition 2.1.38 (Cohomology of a cochain complex)**.** Let $C = (C^*, \delta^*)$ be a cochain complex. An element $c \in C_n$ is called an *n-cochain*. An n-cochain that is in the kernel of δ^n is called an *n-cocycle*, and an n-cochain that is in the image of δ^{n-1} is called an *n-coboundary*. The *nth cohomology group* of C, denoted $H^n(C)$ is the group of n-cocycles modulo the group of n-coboundaries; that is,

$$H^n(C) = \frac{\ker \delta^n}{\text{im}\, \delta^{n-1}}.$$

We will show that each chain complex in fact has a corresponding cochain complex, from which we can define cohomology (of a chain complex).

Let $f : A \to B$ be a group homomorphism and G be an abelian group. Then there is an induced group homomorphism

$$\begin{aligned} f^* : \text{Hom}(B, G) &\to \text{Hom}(A, G), \\ (\varphi : B \to G) &\mapsto (\varphi \circ f : A \to B \to G). \end{aligned}$$

Thus for a chain complex $C = (C_*, \partial_*)$, we may define the corresponding cochain complex $D = (D^*, \delta^*)$ by the following.

$$0 \to \text{Hom}(C_0, G) \xrightarrow{\partial_1^*} \text{Hom}(C_1, G) \xrightarrow{\partial_2^*} \text{Hom}(C_2, G) \xrightarrow{\partial_3^*} \ldots$$

Thus we define cohomology for a chain complex as follows.

Definition 2.1.39 (Cohomology of a chain complex)**.** Let $C = (C_*, \partial_*)$ be a chain complex. The *nth cohomology group* of C is defined by

$$H^n(C) = \frac{\ker \partial_{n+1}^*}{\text{im}\, \partial_n^*}.$$

It turns out that the persistent barcodes of the homology and cohomology of a persistent complex coincides.

Lemma 2.1.40. *Let $\mathcal{C}$ be a persistence complex and F be a field. Then*

$$\mathrm{Pers}(H_n(\mathcal{C})) \cong \mathrm{Pers}(H^n(\mathcal{C}))$$

for each $n \in \mathbb{N}_0$.

Proof. By the universal coefficients theorem for field coefficients [7], there is a natural isomorphism

$$H^k(\mathcal{C}; F) \cong \mathrm{Hom}(H_k(\mathcal{C}; F), F).$$

By natural, we mean that the induced maps

$$H_k(K_i, F) \to H_k(K_j, F) \quad \text{and} \quad H^k(K_j, F) \to H^k(K_i, F)$$

are *adjoint* (that is, one is the transpose of the other) and thus they have the same rank. Persistent (co)homology is uniquely determined by these induced map, and thus the barcodes must be the same. □

We will use this fact to our advantage, the cohomology boundary map has superior properties when computing persistent homology (for certain persistent complexes).

2.2 Complexes

Persistent homology is typically applied to *filtered simplicial complexes*, which we will construct in this section. We also introduce two different constructions, one from point cloud data and one from graphs.

2.2.1 Simplicial complexes

A *simplicial complex* may be considered as an embedding of points, line segments, triangles, and n-dimensional counterparts in Euclidean space; however, it is often beneficial to look at simplicial complexes as a purely combinatorial object; we omit details on how it will be embedded into Euclidean space. Strictly, this combinatorial view would be referred to as an *abstract simplicial complex*, but we will refer to it as a *simplicial complex*.

Definition 2.2.1 (Simplicial complex)**.** An *simplicial complex* is a family of sets that is closed under taking subsets.

Note the existence of the empty simplex that arises from this definition. Some definitions chose to disregard this as a valid simplex, and we will do so here.

Definition 2.2.2. Let K be a simplicial complex.

(i) An element $\sigma \in K$ is called a *p-simplex*, where $p = \dim \sigma = |\sigma| - 1$.

(ii) Let σ be a p-simplex. An element $\tau \subset \sigma$ is called a *face* of σ.

(iii) Let σ be a p-simplex. An element $\tau \supset \sigma$ is called a *coface* of σ.

(iv) Let τ be a face of a p-simplex σ, where $p \geq 0$. If $\dim \tau = p - 1$, then τ is called a facet of σ.

(v) Let τ be a coface of a p-simplex σ. If $\dim \tau = p + 1$, then τ is called a cofacet of σ.

(vi) $K\big|_p$ denotes the set of p-simplices in K (this is not typically a simplicial complex).

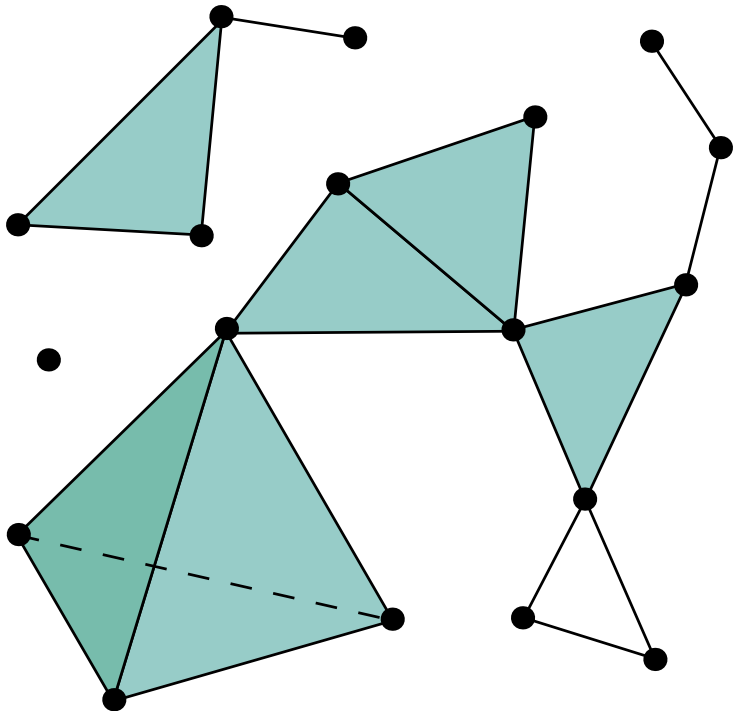


Figure 2.2.1: A simplicial complex of dimension 3.

(vii) The *p-skeleton* of K is defined as $\mathrm{sk}_p(K) = \bigcup_{i\in\{0,\dots,p\}} K\big|_i$.

(viii) The dimension of K is the greatest p such that $K\big|_p$ is non-empty, denoted by $\dim K$.

(ix) A simplicial complex K' is said to be a *subcomplex* of K if $K' \subset K$.

(x) K is said to be finite if $|K| < \infty$.

We will be working exclusively with finite simplicial complex, so we assume all simplicial complexes are finite unless otherwise stated.

An *orientation* of a k-simplex $\sigma = \{v_0, \dots, v_k\}$ is an equivalence class of the orderings of the vertices of σ. We let $(v_0, \dots, v_k) \sim (v_{\tau(0)}, \dots, v_{\tau(k)})$ if the sign of the permutation τ is 1. Denote an oriented simplex by $[\sigma]$, and denote the set of oriented p-simplices of K by $\left[K\big|_p\right]$.

Although we ignore the details of embedding, it is often useful to visualise simplicial complexes in $\mathbb{R}^2$, see Figure 2.2.1.

Definition 2.2.3 (Simplicial chain complex)**.** Let K be a simplicial complex and R be a commutative ring. The *simplicial chain complex of K with coefficients in R* is a chain complex $C(K;R) = (C_*, \partial_*)$ where

$$C_i = R\left\langle \left[K\big|_i\right]\right\rangle$$
$$\partial_i([v_0, \dots, v_i]) = \sum_{j=0}^{i} (-1)^j [v_0, \dots, v_{j-1}, v_{j+1}, \dots, v_i]$$

where $[v_0, \dots, v_i]$ is an oriented i-simplex. We extend ∂_i to the full domain of C_i by extending linearly.

Example 2.2.4. We consider the simplicial complex K consisting of the 2-simplex $[a, b, c]$ and all of its faces. Let $(C_*, \partial_*) = C(K;\mathbb{Z})$. We will present the boundary maps as matrices, so we

pick the basis $([a],[b],[c])$ for C_0, $([ab],[ac],[bc])$ for C_1, and $([abc])$ for C_2. Thus we have

$$C_0 \cong \mathbb{Z}^3, \qquad M_{\partial_0} = \begin{array}{c} \\ [0] \end{array} \begin{array}{c} \begin{array}{ccc} [a] & [b] & [c] \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{array};$$

$$C_1 \cong \mathbb{Z}^3, \qquad M_{\partial_1} = \begin{array}{c} \\ [a] \\ [b] \\ [c] \end{array} \begin{array}{c} \begin{array}{ccc} [ab] & [ac] & [bc] \end{array} \\ \begin{pmatrix} -1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix} \end{array};$$

$$C_2 \cong \mathbb{Z}, \qquad M_{\partial_2} = \begin{array}{c} \\ [ab] \\ [ac] \\ [bc] \end{array} \begin{array}{c} \begin{array}{c} [abc] \end{array} \\ \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \end{array}.$$

Definition 2.2.5 (Simplicial homology)**.** Let K be a simplicial complex and R be a commutative ring. Then the *nth homology group* of K is defined as the homology of the simplicial chain complex of K; that is,

$$H_n(K;R) = H_n(C(K;R)).$$

When $R = \mathbb{Z}$, we simply write $H_n(K;\mathbb{Z}) = H_n(K)$.

Example 2.2.6. We now present some worked examples of homology on simplicial complexes. Note here we are using a shorthand, so a strictly should be written $\{a\}$ and ab similarly should be written $\{a,b\}$.

(i) Let K be the simplicial complex with 0-simplices $[a_1],\dots,[a_n]$ and a 1-simplex $[a_i,a_{i+1}]$ for each $i \in \{1,\dots,n-1\}$. We consider the chain complex of K, $C(K;\mathbb{Z}) = (C_*,\partial_*)$:

$$0 \to \mathbb{Z}^{n-1} \xrightarrow{\partial_1} \mathbb{Z}^n \to 0.$$

For C_0 we pick the basis $([a_1],\dots,[a_n])$, and for C_1 we pick the basis $([a_1a_2],\dots,[a_{n-1}a_n])$. Firstly, as $C_i = 0$ for all $i \geq 2$, $H_i(K) = 0$ for all $i \geq 2$. We now look at the matrix representation of ∂_1.

$$\begin{array}{c} \\ [a_1] \\ [a_2] \\ [a_3] \\ [a_4] \\ \dots \\ [a_{n-1}] \\ [a_n] \end{array} \begin{array}{c} \begin{array}{cccccc} [a_1a_2] & [a_2a_3] & [a_3a_4] & \dots & [a_{n-2}a_{n-1}] & [a_{n-1}a_n] \end{array} \\ \begin{pmatrix} -1 & 0 & 0 & \dots & 0 & 0 \\ 1 & -1 & 0 & \dots & 0 & 0 \\ 0 & 1 & -1 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \end{array}$$

By performing the row operations $R_i \leftarrow R_i + R_{i-1}$ in the order $i \in \{2,3,\dots,n\}$, we get the following matrix (that is in Smith normal form).

$$\begin{pmatrix} -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & -1 & 0 & \dots & 0 & 0 \\ 0 & 0 & -1 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & -1 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

Thus we read off the kernel and image of ∂_1:

$$\operatorname{im}\partial_1 = \mathbb{Z}^{n-1}, \qquad \ker\partial_1 = 0.$$

Thus we get

$$H_0(K) = \frac{\mathbb{Z}^n}{\operatorname{im}\partial_1} = \mathbb{Z}, \qquad H_1(K) = \frac{0}{0} = 0.$$

The 0th and 1st integral homology groups of graphs have quite succinct interpretations: the rank of the 0th homology group corresponds to the number of connected components there are and the rank of the 1st homology group corresponds to the number of *linearly independent cycles* there are. In this example, we have the path graph, which has one connected component and no cycles.

(ii) We let K be the simplicial complex with two 0-simplices, $[a]$ and $[b]$, and n 1-simplices of the form $[a,b]$. We have the chain complex $(C_*, \partial_*) = C(K;\mathbb{Z})$ written as

$$0 \to \mathbb{Z}^n \xrightarrow{\partial_1} \mathbb{Z}^2 \to 0.$$

Again we have $C_i = 0$ for $i \geq 2$, thus $H_i(K) = 0$ for $i \geq 2$. We have the following matrix representation of ∂_1.

$$\begin{array}{c} \\ [a] \\ [b] \end{array} \begin{array}{c} \begin{array}{ccccc} [ab] & [ab] & [ab] & \dots & [ab] \end{array} \\ \begin{pmatrix} -1 & -1 & -1 & \dots & -1 \\ 1 & 1 & 1 & \dots & 1 \end{pmatrix} \end{array}$$

This reduces to the following matrix.

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

Thus we get

$$\operatorname{im}\partial_1 = \mathbb{Z}, \qquad \ker\partial_1 = \mathbb{Z}^{n-1}.$$

Therefore,

$$H_0(K) = \mathbb{Z}^2/\mathbb{Z} = \mathbb{Z}, \qquad H_1(K) = \mathbb{Z}^{n-1}/0 = \mathbb{Z}^{n-1}.$$

To interpret this example, we have two points and n edge connecting them; thus, we have 1 connected component and $n-1$ linearly independent cycles.

(iii) Our last example moves away from graphs. Consider the simplicial complex K consisting of all proper faces of the 3-simplex $[a,b,c,d]$. We may think of K as a the boundary of a tetrahedron (homeomorphic to S^2). We have the following integral chain complex.

$$0 \to \mathbb{Z}^4 \xrightarrow{\partial_2} \mathbb{Z}^6 \xrightarrow{\partial_1} \mathbb{Z}^4 \to 0.$$

We now look the matrix representations of the (non-trivial) boundary maps and their reduction.

$$M_{\partial_1} = \begin{array}{c} \\ [a] \\ [b] \\ [c] \\ [d] \end{array} \begin{array}{c} \begin{array}{cccccc} [ab] & [ac] & [ad] & [bc] & [bd] & [cd] \end{array} \\ \begin{pmatrix} -1 & -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \to \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$M_{\partial_2} = \begin{array}{c} \\ [ab] \\ [ac] \\ [ad] \\ [bc] \\ [bd] \\ [cd] \end{array} \begin{array}{c} \begin{array}{cccc} [abc] & [abd] & [acd] & [bcd] \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$M_{\partial_3} = \begin{matrix} \\ [abc] \\ [abd] \\ [acd] \\ [bcd] \end{matrix} \begin{matrix} [abcd] \\ \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix} \end{matrix} \to \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Thus we get the homology groups

$$H_n(K) = \begin{cases} \mathbb{Z} & n \in \{0, 2\}, \\ 0 & \text{otherwise.} \end{cases}$$

The rank of the 0th homology group can still be interpreted as the number of connected component. The rank of the 2th homology group can be interpreted as the number of 2-dimensional *holes* in K (or S^2); that is, the number of *voids*. As our tetrahedron is hollow, it has one void (precisely the space the boundary encompasses)

Filtrations are widely used in algebraic topology (particularly in homological algebra). We have already seen gradings for rings (and modules), and these two are closely linked.

Definition 2.2.7 (Filtration)**.** A *filtration* is a family $\{S_i\}_{i\in\mathcal{I}}$ of subobjects of a given structure S, where $\mathcal{I}$ is some totally ordered set, and for $i, j \in I$ we have

$$i \leq j \implies S_i \subset S_j.$$

A filtered ring is a generalisation of a graded ring.

Definition 2.2.8 (Filtered simplicial complex)**.** A *filtered simplicial complex* is a filtration of simplicial complexes $\mathbb{K} = \{K_i\}_{i\in\mathcal{I}}$ (that is, if $i < j$, then K_i is a subcomplex of K_j).

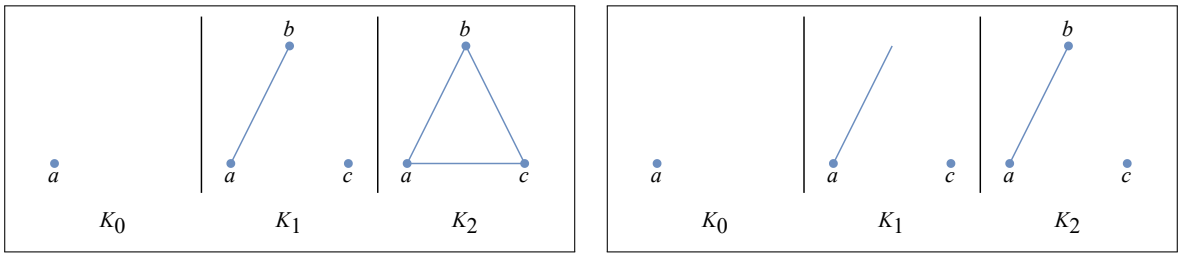
We will be working with finite filtered simplicial complexes. Thus, as with our definition of grading, we will take $\mathcal{I} = \{0, \ldots, n\}$ for some $n \in \mathbb{N}$. A filtration of a simplicial complex K is a filtered simplicial complex $\mathbb{K} = (K_i)_{i=0}^n$ such that $K_n = K$.

Example 2.2.9. We consider two examples, one that is a valid filtered simplicial complex and another that is not.

(i) Consider the sequence

$$\underbrace{\{a\}}_{K_0} \subset \underbrace{\{a, b, c, ab\}}_{K_1} \subset \underbrace{\{a, b, c, ab, ac, bc\}}_{K_2}.$$

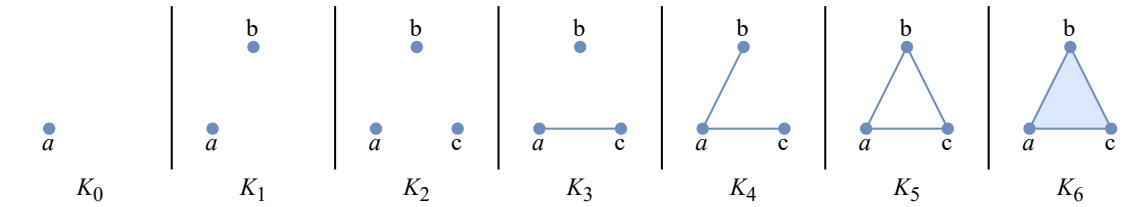
This is indeed a filtered simplicial complex, as shown in Figure 2.2.2a.



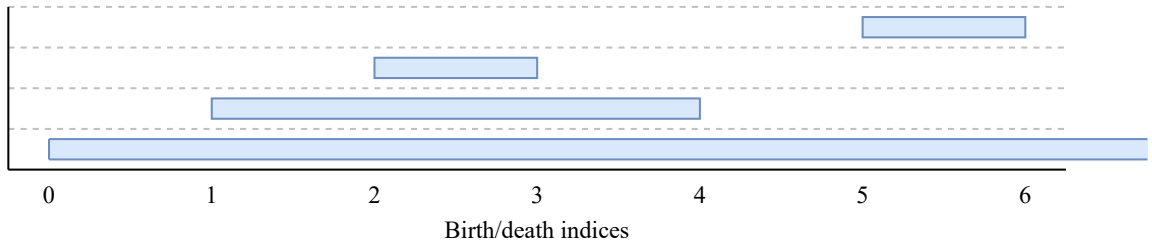
(a) A valid filtered simplicial complex, each filtration level is a subset of the next and are all valid simplicial complexes.

(b) An invalid filtered simplicial complex, each filtration level is a subset of the next, but K_1 is not a valid simplicial complex.

Figure 2.2.2: Two sequences of simplicial complex, one that defines a filtration and another that does not.



(a) A filtered simplicial complex.



(b) The persistent barcodes for the filtered simplicial complex. Note the barcode at the bottom corresponds to the $\mathcal{P}$-interval $(0, \infty)$.

Figure 2.2.3: A filtered simplicial complex alongside the persistent barcode.

(ii) Consider the sequence

$$\underbrace{\{a\}}_{K_0} \subset \underbrace{\{a, c, ab\}}_{K_1} \subset \underbrace{\{a, b, c, ab\}}_{K_2}.$$

This is *not* a filtered complex, although each level is a subset of the next, observe that $ab \in K_1$ but $ab \supset b \notin K_1$.

We note that a maximal filtration of simplicial complexes induces an ordering on the simplices. For example, consider the filtered simplicial complex

$$\{\{1\}\} \subset \{\{1\}, \{2\}\} \subset \{\{1\}, \{2\}, \{1, 2\}\}.$$

This filtration corresponds to the ordering $(\{1\}, \{2\}, \{1, 2\})$.

For a non-maximal filtration of simplicial complexes, there are multiple induced orderings. For example, consider the filtered simplicial complex

$$\{\{1\}, \{2\}\} \subset \{\{1\}, \{2\}, \{1, 2\}\}.$$

This may correspond to the ordering above or $(\{2\}, \{1\}, \{1, 2\})$. We call such a corresponding ordering a *compatible ordering of simplices* from the filtration.

Lemma 2.2.10. *Any filtered simplicial complex is a persistence complex.*

Proof. As in Definition 2.2.3, we take $C^i = C(K_i; R) = (C^i_*, \partial^i_*)$ for some commutative ring R. For the chain maps, we use the inclusion map: $f^i = \iota_i : C^i_* \to C^{i+1}_*$. It is clear that these are in fact chain maps. □

Definition 2.2.11 (Persistence barcodes of a filtered simplicial complex)**.** Let $\mathbb{K} = \{K_i\}_{i=0}^n$ be a filtered simplicial complex. The *persistence barcodes* of $H_*(\mathbb{K})$ are simply the persistence barcodes of the homology of the corresponding persistence complex $\mathcal{C}$; that is,

$$\mathrm{Pers}(H_*(\mathbb{K})) = \mathrm{Pers}(H_*(\mathcal{C})).$$

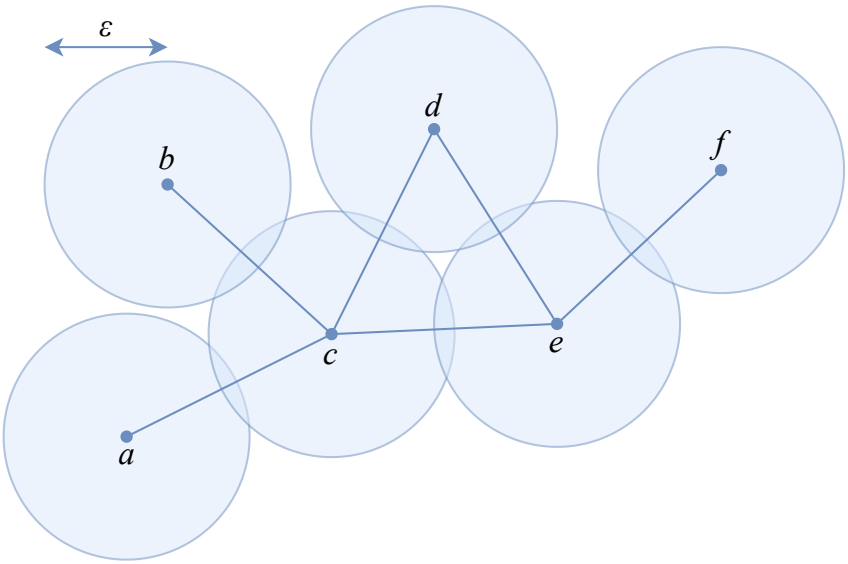


Figure 2.2.4: Six points with pairwise intersection among balls indicated by a straight edge connecting their centre. The Vietoris-Rips complex fills in all the triangles (one in this example) and higher-dimensional counterparts.

Example 2.2.12. We consider the maximal filtered simplicial complex $\mathbb{K}$ corresponding to the following simplex ordering:

$$(a, b, c, ac, ab, bc, abc)$$

(note we are using the shorthand $a = \{a\}$, $ab = \{a, b\}$, etc.). We detail the span of the homology classes:

(i) the 0-dimensional homology class created by a persists through to the end complex;

(ii) the 0-dimensional homology class created by b is killed by ab;

(iii) the 0-dimensional homology class created by c is killed by ac; and

(iv) the 1-dimensional homology class created by bc is killed by abc.

Figure 2.2.3 visualises this example: in Figure 2.2.3a we have a visualisation of the filtration levels of $\mathbb{K}$ and in Figure 2.2.3b we have the persistent barcodes for $\mathbb{K}$.

2.2.2 Vietoris-Rips complex

We move to our first construction of a filtered simplicial complex, which is built from point cloud data (a subset of points from some metric space).

Definition 2.2.13 (Vietoris-Rips complex)**.** Let (M, d) be a metric space, $S \subset M$ be finite, and $\varepsilon > 0$. The *Vietoris-Rips complex* of S at scale ε, denoted $\mathrm{VR}(S; \varepsilon)$, is a simplicial complex defined as

$$\mathrm{VR}(S; \varepsilon) = \{\sigma \subset S : d(u, v) \leq \varepsilon \ \ \forall u, v \in \sigma\}.$$

Example 2.2.14. Figure 2.2.4 shows an example of a Vietoris-Rips complex. In this example, we have

$$\begin{aligned}
\mathrm{VR}(S; \varepsilon)\big|_0 &= \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}\}, \\
\mathrm{VR}(S; \varepsilon)\big|_1 &= \{\{a, c\}, \{b, c\}, \{c, d\}, \{c, e\}, \{d, e\}, \{, f\}\}, \\
\mathrm{VR}(S; \varepsilon)\big|_2 &= \{\{c, d, e\}\}.
\end{aligned}$$

Lemma 2.2.15. *Any Vietoris-Rips complex is a simplicial complex.*

Proof. Let (M, d) be a metric space, $S \subset M$ be finite, $\varepsilon \geq 0$, and $K = \mathrm{VR}(S; \varepsilon)$. Let $\sigma \in K$ and $\sigma' \subset \sigma$. Trivially, $\varnothing \in K$. Let $u, v \in \sigma'$ be (not necessarily distinct) points. Then $u, v \in \sigma$, so $d(u, v) \leq \varepsilon$. Thus $\sigma' \in K$. □

We now use the Vietoris-Rips complex to form a filtered simplicial complex.

Definition 2.2.16 (Vietoris-Rips filtration)**.** Let (M, d) be a metric space and $S \subset M$ be finite. Define the *Vietoris-Rips filtration* as $\{\mathrm{VR}(S; \varepsilon)\}_{\varepsilon \in \mathbb{R}_{\geq 0}}$.

Corollary 2.2.17 (of Lemma 2.2.15)**.** *A Vietoris-Rips filtration is a filtered simplicial complex.*

We now introduce an alternative definition of a filtered Vietoris-Rips complex, by defining a weight function. This will be useful when we consider fast constructions in Section 4.1.

Definition 2.2.18 (Weight function)**.** Let K be a simplicial complex. $w : K \to \mathbb{R}$ is a *weight function* (defined on each simplex of K) if $\{w^{-1}(-\infty, \varepsilon]\}_{\varepsilon \in \mathbb{R}}$ is a filtered simplicial complex.

We call a simplicial complex with a weight function (K, w) a *weight-filtered simplicial complex.*

Definition 2.2.19 (Vietoris-Rips filtration, by weight function)**.** Let (M, d) be a metric space and $S \subset M$ be finite. We define the filtered Vietoris-Rips complex as $\{w(-\infty, \varepsilon)\}_{\varepsilon \in \mathbb{R}_{\geq 0}}$ where we define the weight function as

$$w(\sigma) = \begin{cases} 0 & \text{if } \dim \sigma \leq 0 \\ d(u, v) & \text{if } \sigma = \{u, v\}, \\ \max_{\tau \subset \sigma} w(\tau) & \text{otherwise.} \end{cases}$$

Lemma 2.2.20. *Definition 2.2.16 and Definition 2.2.19 are equivalent.*

Proof. Let (M, d) be a metric space, $S \subset M$ be finite, and $\varepsilon \in \mathbb{R}_{\geq 0}$. We will show that $w^{-1}(-\infty, \varepsilon] = \mathrm{VR}(S; \varepsilon)$. Let $\sigma \subset S$. Trivially, if $\dim \sigma \leq 0$ then $\sigma \in w^{-1}(-\infty, \varepsilon]$ and $\sigma \in \mathrm{VR}(S; \varepsilon)$. Now suppose $\dim \sigma = 1$, so $\sigma = \{u, v\}$ (distinct points). Then $w(\sigma) = d(u, v)$ and so $\sigma \in w^{-1}(-\infty, \varepsilon]$ if and only if $\sigma \in \mathrm{VR}(S; \varepsilon)$. Now suppose $\dim \sigma \geq 2$. Then

$$w(\sigma) = \max_{\sigma' \subset \sigma} w(\sigma') = \max_{\{u,v\} \subset \sigma} d(u, v)$$

and so $w(\sigma) \leq \varepsilon$ if and only if $d(u, v) \leq \varepsilon$ for all $\{u, v\} \subset \sigma$. □

2.2.3 Power filtrations

Finally, we move to our last construction of a filtered simplicial complex. The typical road-map for computation of persistent homology is to take a point cloud in some metric space, construct a filtered Vietoris-Rips complex, and then compute the persistent homology of the complex. In this subsection, we explore an alternative filtered simplicial complex we may construct from graphs that we may apply persistent homology to. Some applications of such a construction can be found in Chapter 5. Many of the definitions here are taken from the work by Ferri, Bergomi, and Zu [8].

Definition 2.2.21 (Graph)**.** A *undirected graph* is a tuple $G = (V, E)$ where V is a set of objects called *vertices* and $E \subset \{\{x, y\} : x, y \in V, x \neq y\}$ is a set of *edges*.

This is a traditional definition of a graph, but we can simply view it as a 1-simplicial complex where the vertices are the 0-simplices and the edges are 1-simplices. The above definition may be referred to as an *undirected simple graph*. A directed graph may be similarly defined where the vertices in the edges are ordered (resembling ordered simplices), and a simple graph is one with no self loops (that is, $\{x\} \notin E$) and no two vertices may have more than one edge between them.

Definition 2.2.22. Let $G = (V, E)$ be an undirected graph.

(i) (*Adjacency*) The vertices in an edge may be referred to as *endpoints*, and two points are *adjacent* if there exists an edge between them.

(ii) (*Paths and cycles*) A *path* in a graph is a ordered collection of vertices such that there is an edge between any two adjacent vertices. A *path* is called a *cycle* if the first and last vertex coincide. The *length* of a path $p = (v_1, \ldots, v_n)$ is $|p| = n - 1$.

(iii) (*Subgraphs*) Let $G = (V, E)$ and $G' = (V', E')$ be graphs. G' is a *subgraph* of G if $V' \subset V$ and $E' \subset E$.

(iv) (*Induced subgraphs*) Let $S \subset V$. Then the subgraph *induced by* S, denoted $G[S]$, is the graph whose vertex set is S and whose edge set consists of all edges in E that have both endpoints in S.

(v) (*Neighbourhoods*) The *neighbourhood* of a vertex $v \in V$ is the subgraph of G induced by all vertices adjacent to v.

(vi) (*Graph metric*) We may impart a graph with a metric, define

$$\begin{aligned} d : V \times V &\to \mathbb{N}_0 \cup \{\infty\}, \\ d(u, v) &= \min\{p : p \text{ is a path from } u \text{ to } v\}. \end{aligned}$$

If $u = v$, then define $d(u, v) = 0$. If no such path from u to v exists, then define $d(u, v) = \infty$. From this, we derive a sense of the size of a graph: define the *diameter* of G as

$$\operatorname{diam}(G) = \max_{(u,v) \in V^2} d(u, v).$$

(vii) (*Graph power*) Define the *kth power* G^k of G as

$$G^k = \left(V, \{\{u, v\} \subset V : 1 \leq d(u, v) \leq k\}\right).$$

See Figure 2.2.5 for an example of the powers of a small graph.

(viii) (*Clique*) G is said to be a *clique* if every two distinct vertices are adjacent. Alternatively, if $\operatorname{diam} G = 1$.

For a graph G, we may sometimes denote the set of vertices of G by $V(G)$, and the set of edges of G by $E(G)$.

Definition 2.2.23 (Clique complex)**.** Let $G = (V, E)$ be an undirected graph. The *clique complex* of G is defined as

$$\operatorname{Cl}(G) = \{S \subset V : G[S] \text{ is a clique}\}.$$

Since every subset of a clique is itself a clique, $\operatorname{Cl}(G)$ is well-defined. The clique complex of a graph simply *fills in* higher dimensional simplices for which all faces are present.

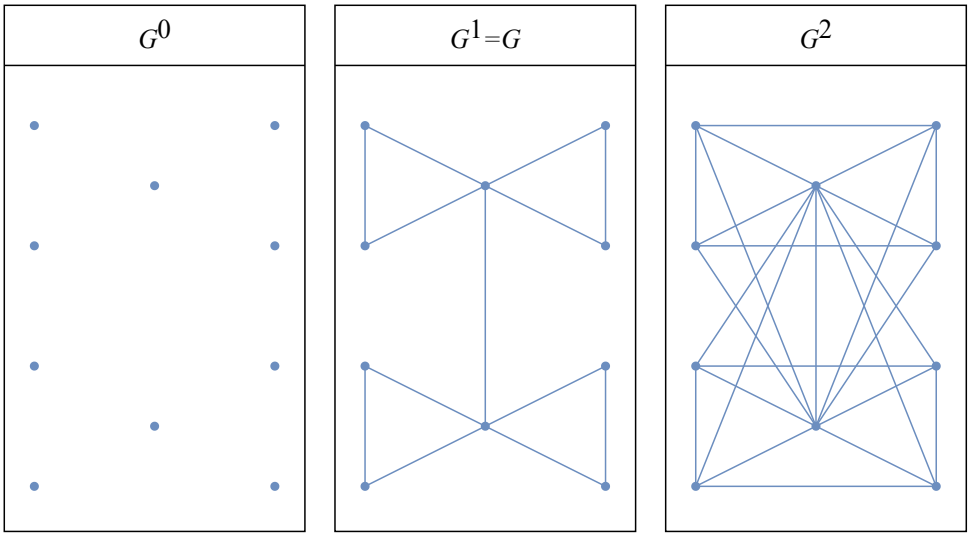


Figure 2.2.5: An example of G^k for $k \in \{0, 1, 2\}$, for a small 10-vertex graph G.

Definition 2.2.24 (Power filtration)**.** Let $G = (V, E)$ be an undirected graph. Define the *power filtration* $\mathrm{Pow}(G)$ of G as the filtration

$$\mathrm{Pow}(G) : \mathrm{Cl}(G^0) \subset \mathrm{Cl}(G^1) \subset \ldots \subset \mathrm{Cl}(G^{\mathrm{diam}\, G}).$$

The power filtration of a graph is a commonly used construction for applying persistent homology.

We will later make use of the *Jaccard index*, which gives us a notion of how *similar* two nodes are, based on their neighbourhoods.

Definition 2.2.25 (Jaccard index)**.** Let $G = (V, E)$ be an undirected graph. The *Jaccard index* $J : V^2 \to [0, 1]$ is defined by

$$J(u, v) = \frac{|V(N(u)) \cap V(N(v))|}{|V(N(u)) \cup V(N(v))|}.$$

If two nodes $u, v \in V$ share the same neighbours, then $J(u, v) = 1$. Conversely, if two nodes share no neighbours, then $J(u, v) = 0$.

Chapter 3

Computational complexity theory

3.1 Modelling computation

This subsection serves as an introduction to formalizations of our intuitive notions of *problems* and *algorithms*. We will look at how one may precisely define a problem, and use the abstract framework of Turing machines to explore the computability of such problems.

3.1.1 Computational problems

In order to study computational problems, we *encode* them so we may study them as objects derived from a finite set.

Definition 3.1.1 (Kleene star)**.** Given a set X, define $X^0 = \{\varepsilon\}$ (the set containing only the empty string) and $X^1 = X$. For each $i \in \{2, 3, \ldots\}$, we recursively define the set

$$X^{i+1} = \{wv : w \in X^i, v \in X\}.$$

Definition 3.1.2 (Formal language)**.** A *formal language* (or just *language*) $\mathcal{L}$ over an *alphabet* (some none empty set of symbols, which are called *letters*) Σ is some subset of Σ^*. A *word* $w \in \mathcal{L}$ is an element of a language.

Simple encodings can be used to represent general mathematical objects as strings of bits (that is, words of a language over $\{0, 1\}$), but we avoid dealing with low level representation details. We use $\langle x \rangle$ to denote some canonical binary representation of an object x, but we will typically omit this notation and simple use x to refer to the object and its representation.

Formally, a *computational problem* is a problem that we may expect a computer to be able to solve. We first consider two examples of a simple type of computational problem.

Problem 3.1.3 (Primality**).**
Instance: let $n \in \mathbb{N}$.
Question: is n prime?

Problem 3.1.4 (Reachability**).**
Instance: Let $G = (V, E)$ be a graph and $v, w \in V$ two vertices.
Question: is there a path from v to w in G?

Both Primality and Reachability expect a *yes* or *no* answer. Problems of this form are called *decision problems*, and we may formally define them as below.

Definition 3.1.5 (Decision problem)**.** A *decision problem* is a yes-or-no question on an infinite set of inputs. Formally, it is a tuple (I, Y) where $I \subset \{0,1\}^*$ is an alphabet of *inputs* and $Y \subset I$ is a language of inputs for which the answer is *yes*.

We may rewrite the above problems as

$$\begin{aligned} \textsc{Primality} &= (\mathbb{N}, \{n \in \mathbb{N} : n \text{ is prime}\}), \\ \textsc{Reachability} &= (I, Y) \end{aligned}$$

where $I = \{(G, u, v) : G = (V, E)$ is a graph and $u, v \in V\}$ and $(G, u, v) \in Y$ if and only if there is a path from u to v in G.

Definition 3.1.6 (Function problem)**.** A *function problem* is a relation $P \subset \{0,1\}^* \times \{0,1\}^*$. An algorithm solves P if for every $x \in \{0,1\}^*$, the algorithm produces $y \in \{0,1\}^*$ such that $(x, y) \in P$ (if such a y exists).

Problem 3.1.7 (TSP**).**
Instance: let H be an undirected weighted graph.
Question: find the shortest possible Hamiltonian cycle (that is, a path that visits each vertex once and starts and ends at the same vertex).

Finding the persistent barcode of a filtered simplicial complex is a function problem, as is the construction of a Vietoris-Rips complex.

3.1.2 Turing machines

One may informally define an algorithm as a collection of simple instructions for carrying out some task. Turing [9] introduced the now ubiquitous model of computation. We will not bother ourselves with the granular detail of the definition of a Turing machine, as many variants exist that are all equivalent.

Theorem 3.1.8 (Turing [9])**.** *The intuitive notion of an algorithm is equivalent to the mathematical concept of an algorithm defined by Turing machines.*

A Turing machine M consists of the following components:

- a finite set of states;
- an infinite tape with storage cells, with cells containing a single symbol from some alphabet Π;
- a device called the head that can read and write on a cell, and move along the tape; and
- a transition function.

M will start its computation at an initial state, with an input on the tape, and the head at the start of the input. The head will read the contents of the cell, and by also considering the state, use the transition function to decide what to write on the cell, whether to move right or left along the tape, and what state to enter. The Turing machine will run until its enters a halt state (although it may also run forever), and the output of the machine will be the contents of the cells on the tape when the machine halts. We denote the output of a Turing machine on input $x \in \Sigma^*$ ($\Sigma \subset \Pi$ is the *input alphabet*) as $M(x)$ (note this may be undefined if the Turing machine does not halt). See Figure 3.1.1 for a schematic of a Turing machine.

For decision problems, we disregard the machine's output but introduce two new halt states: an accept state and a reject state. If M halts on input x in the accept state, we say that M *accepts* x and similarly for M rejecting x. We denote $L(M)$ as the language of strings that M accepts. If there is a Turing machine M such that $\mathcal{L} = L(M)$, then the language L is said to

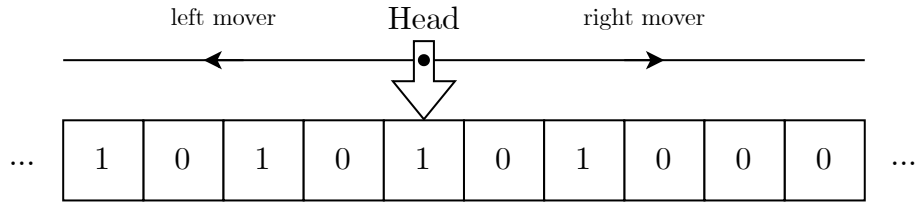


Figure 3.1.1: Schematic of a Turing machine.

be *semidecidable*. We say a language is *decidable* if it is semidecidable and rejects all strings outside of the language.

Decision problems offer a simple introduction to modelling computation, but most of the problems we will look at expect an answer more than just *yes* or *no*.

We take a brief moment to comment on the existence on *undecidable* problems; that is, a problem for which no algorithm decides it.

Problem 3.1.9 (HaltingProblem).
Instance: let M be a Turing machine and $x \in \Sigma^*$ an input.
Question: does M halt on x?

Theorem 3.1.10. *There is no Turing machine that decides* HaltingProblem.

Sketch of proof. We omit the details of the proof, as it requires some constructs that are of little use in our context, but we provide a rough outline. We construct an adversary Turing machine D that takes as input a Turing machine M. D will accept if M rejects with itself (or rather, an encoding of itself) as input. If M accepts itself, then D loops indefinitely. By considering how D runs on the encoding of itself, we find a contradiction. □

3.2 Computational complexity

The *complexity* of an algorithm is the amount of resources required to run it, but we focus on the *time* (time complexity) and *memory* (space complexity) requirements.

Definition 3.2.1 (Time complexity)**.** Let M be a Turing machine that halts on all inputs. The *time complexity* (or *running time*) of M is a function $T_M : \mathbb{N} \to \mathbb{N}$ where $T_M(n)$ denotes the maximum number of steps that M uses on an input of length n.

We have already discussed variations of Turing machines that are equivalent that may lead to subtle differences to running times of algorithm, and we need a tool to deal with this.

Time complexities of algorithms are given in terms of Big O notation, which has the analogue of $\leq$ on the reals. We also have a similar notation for an analogue of $\geq$ and $=$ on the reals.

Definition 3.2.2 (Big O notations)**.** Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$

(i) (Big O notation, $\leq$) $f(n) = O(g(n))$ if there is $C \in \mathbb{R}$ and $k \in \mathbb{N}$ such that for all $n \geq k$, $f(n) \leq Cg(n)$;

(ii) (Big Ω notation, $\geq$) $f(n) = \Omega(g(n))$ if there is $C \in \mathbb{R}$ and $k \in \mathbb{N}$ such that for all $n \geq k$, $f(n) \geq Cg(n)$; and

(iii) (Big Θ notation, $=$) $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Table 3.2.1 shows some common time complexity and algorithms with that complexity.

Table 3.2.1: A list of common time complexities.

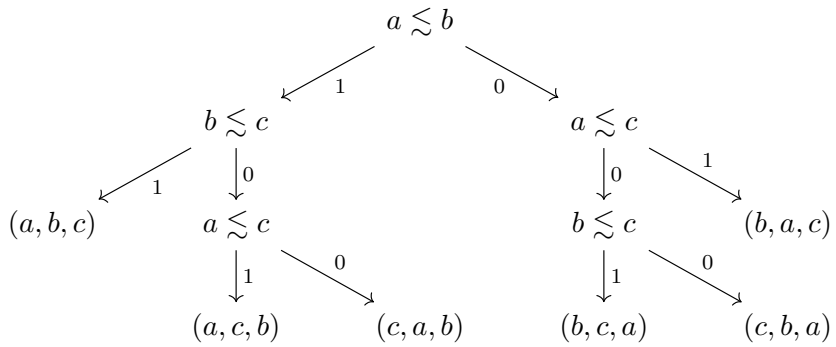
Notation	Name	Example
$O(1)$	Constant	Search in hash table
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Searching an unsorted list
$O(n \log n)$	Linearithmic	Fastest comparison-based sorting
$O(n^c)$, $c \geq 1$	Polynomial	REACHABILITY
$O(2^n)$	Exponential	Dynamic programming solution to TSP
$O(n!)$	Factorial	Brute-force solution to TSP

Problem 3.2.3 (SORTING).
Instance: let $(A, \lesssim)$ be a totally preordered finite set.
Question: find an enumeration $\{a_i\}_{i=1}^{|A|}$ of A, so $a_i \lesssim a_j$ for all $i < j$.

We informally introduce the *comparison model* for sorting algorithms. A *comparison sort* is a sorting algorithm that may only read the elements of a totally preordered set $(A, \lesssim)$ through the operation $\lesssim$.

Theorem 3.2.4. *Any deterministic comparison sort algorithm has time complexity* $\Omega(n \log n)$.

Proof. We recall that our definition of time complexity considers the *worst-case performance* of an algorithm. Let M be a sorting algorithm (Turing machine) with time complexity $T_M(n)$. We now fix a set A of $n \in \mathbb{N}$ elements. We may construct a tree G where the leaves correspond to all possible outputs of our algorithm depending on the total preorder given on A, and the internal nodes correspond to a comparison. For example, below we see such a decision tree for $A = \{a, b, c\}$.



The number of leaves for such a decision tree is the number of permutations of the input set; that is, $n!$. It is understood this balanced binary tree has a height $h = \log(n!)$. It follows that

$$h = \log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

and so, as h corresponds to the maximum number of operations we may perform, $T_M(n) = \Omega(n \log n)$. □

Definition 3.2.5 (Space complexity)**.** Let M be a Turing machine that halts on all inputs. The *space complexity* of M is a function $S_M : \mathbb{N} \to \mathbb{N}$ where $S_M(n)$ denotes the maximum number of distinct tape cells that M visits on an input of length n.

One may see that, for any Turing machine M, that $S_M(n) = \Omega(n)$ as we have to store the input at the start of the computation. In order to allow for meaningful analysis into algorithms that use sublinear space, we consider two tapes on our Turing machine: a read-only tape in which the input sits and a read/write tape which we consider to be the *working space*.

Chapter 4

Problems in the computation of persistent homology

Here we apply the knowledge built up in Chapter 2 and Chapter 3 to introduce some problems in the computation of persistent homology. As previously mentioned, we focus on two main steps in the pipeline:

(i) construction of a filtered simplicial complex (Section 4.1); and

(ii) computation of the corresponding persistent barcodes (Section 4.2).

For (i), we focus on Vietoris-Rips filtrations, as it is the prevalent complex used for point-cloud data [1].

We conduct many experiments in this section. All experiments were computed on a 6-Core processor with 3.60 GHz base clock speed, and 16 GB RAM with clock speed 3733 MHz. The programming language used is *Python* [10]. Source code for this project can be found in Appendix C.

4.1 Construction of Vietoris-Rips complexes

We dedicate this section to the problem of constructing the Vietoris-Rips filtration of a given metric space. We first introduce the problem formally.

Problem 4.1.1 (VRFilt).
Instance: let (M, d) be a metric space and $S \subset M$ be a finite set of points.
Question: compute a compatible ordering of simplices from the filtered Vietoris-Rips complex $(\mathrm{VR}(S;\varepsilon))_{\varepsilon\in\mathbb{R}_{\geq 0}}$.

For a metric space (M, d), $S \subset M$, we note that $K = \mathrm{VR}(S;\infty)$ is the simplicial complex containing a single $|S|$-simplex and all of its faces, so the total count of faces is the sum of the first n triangular numbers (including the zeroth triangular number), which is not computationally feasible. It is for this reason that we upper bound ε in our construction, giving the following derived problem.

Problem 4.1.2 ($\hat{\varepsilon}$-VRFilt).
Instance: let (M, d) be a metric space, $S \subset M$ be a finite set of points, and $\hat{\varepsilon} \in \mathbb{R}_{\geq 0}$.
Question: compute a compatible ordering of simplices from the filtered Vietoris-Rips complex $(\mathrm{VR}(S;\varepsilon))_{\varepsilon\in[0,\hat{\varepsilon}]}$.

We now follow the approach given by Zomorodian [11]. Let (M, d) be a metric space, $S \subset M$ be a finite set of points, and $\hat{\varepsilon} \in \mathbb{R}_{\geq 0}$. We may split the computation of the Vietoris-Rips complex into the following phases:

(i) compute the weighted graph (G, w) where $G = (V, E)$ is the 1-skeleton of $\mathrm{VR}(S; \hat{\varepsilon})$ and $w : E \to \mathbb{R}_{\geq 0}$ is defined by $w(u, v) = d(u, v)$;

(ii) compute the *expansion* of (G, w) to the weight-filtered simplicial complex $(\mathrm{VR}(S; \hat{\varepsilon}), w)$; then

(iii) obtain the simplex ordering by sorting the simplices according to their weights.

Let n be the number of points in S. We assume that we can compute $d(u, v)$ for any $u, v \in S$ in $O(1)$ time. It is clear (iii) reduces to SORTING, which we can achieve in time $\Theta(n \log n)$. Thus, we bring our focus to (i) and (ii), algorithms for which will be hence force referred to as the *skeleton method* and *expansion method* respectively.

4.1.1 Skeleton methods

We first formalise the skeleton method, which reduces to the following problem.

Problem 4.1.3 (ε-NNS).
Instance: let (M, d) be a metric space, $S \subset M$ be a finite set of points, and $\varepsilon \in \mathbb{R}_{\geq 0}$.
Question: for each $x \in S$, compute $\{y \in S \setminus \{x\} : d(x, y) \leq \varepsilon\}$.

ε-NNS has the following approximate analogue.

Problem 4.1.4 ($(1 + \delta, \varepsilon)$-ANNS).
Instance: let (M, d) be a metric space, $S \subset M$ be a finite set of points, and $\varepsilon \in \mathbb{R}_{\geq 0}$.
Question: for each $x \in S$, compute $\{y \in S \setminus \{x\} : d(x, y) \leq (1 + \delta)\varepsilon\}$.

We first recognise the brute-force solution, which may run in $O(n^2)$ time and has the benefit of being exact.

Arya et al. [12] made use of a particular tree structure (called BBD-trees) to achieve a solution in the case when $M = \mathbb{R}^d$, with query time of $O(c(d) \log n)$ and $O(dn)$ space, with $O(dn \log n)$ processing time. The function $c(d)$ is some function dependent on the dimension of M, there is no information on the asymptotic behaviour of this function presented in the literature, but empirical evidence suggests that it depends heavily on the underlying distribution of $S \subset M$.

The exact algorithm provided by Arya et al. [12] is in fact an approximation algorithm that allows the value of δ to be set (precisely to $\delta = 0$ for the exact case). In the case $\delta > 0$, the function c given above also depends on δ but all other complexities remain the same. A lower bound of c is given as

$$c(d, \delta) \leq d\lceil 1 + 6d/\delta \rceil^d.$$

A recent benchmark study [13] found the fastest current algorithm to solve $(1 + \delta, \varepsilon)$-ANN as one based on *navigable small-world graphs with controllable hierarchy* [14]. We briefly touch on some underlying theory before introducing this algorithm.

A graph is a *navigable small-world graph* if the greedy graph routing (shown in Algorithm 4.1.1) runs in $O(\log^k n)$ time, for $k > 1$. We combine such a graph with the concept of *skip lists*.

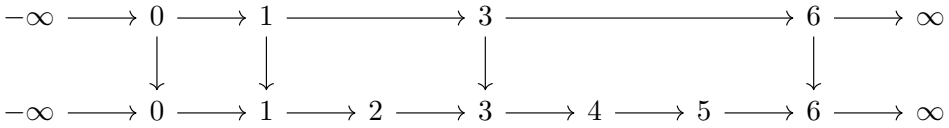
A *skip list* is a probabilistic data structure that allows $O(\log n)$ average search complexity and insertion complexity within an ordered sequence of n elements.

The initial idea: for a sorted list, we create a duplicate list where a given element of the initial list is duplicated with probability 0.5. For every duplicated item, we store a pointer at the item

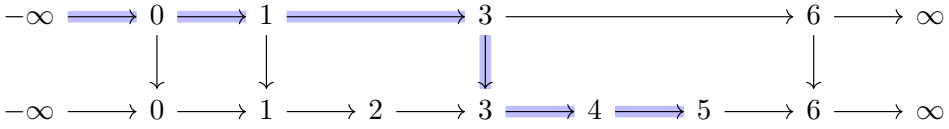
Algorithm 4.1.1 A greedy routing algorithm for ε-NNs.

```
1: function GreedyRouting(graph G, heuristic h, start node v)
2:     best ← v
3:     while true do
4:         bestNeighbour ← arg max_{n∈N_G(best)} h(n)
5:         if h(bestNeighbour) > h(best) then
6:             best ← bestNeighbour
7:         else
8:             return best
9:         end if
10:    end while
11: end function
```

in the second list back to the first list. We also add *sentinel nodes* for safety. We do this for the sorted list $(0, 1, 2, 3, 4, 5, 6)$ as follows.

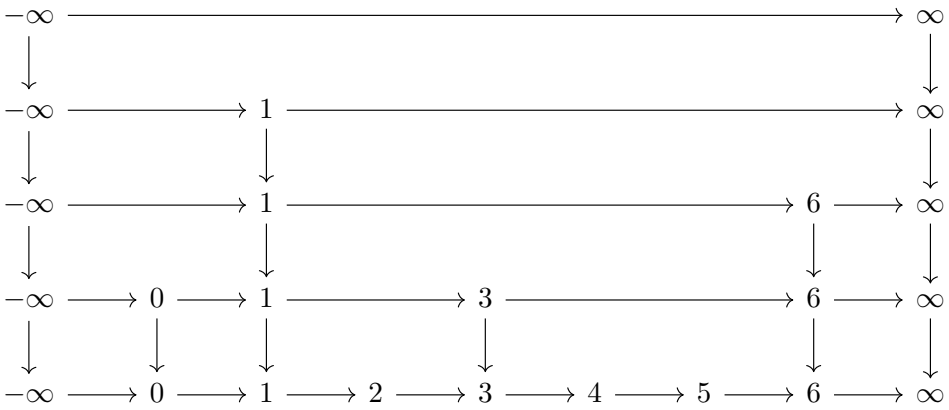


Searching for 5 is shown below.



The probability a given duplicated node is followed by k non-duplicated nodes is $\frac{1}{2^k}$. Thus, the expected number of nodes examined when searching the initial list is $\sum_{k=1}^{\infty} \frac{1}{2^k} = 2$. So by adding this *shortcut list*, we have reduced the expected time complexity from n to $\frac{n}{2} + O(1)$.

An improvement: keep adding shortcut lists of the top shortcut list until we are out of elements.



The expected number of levels is $\log n$, which is easy to prove. At each level, we cut search time in half (excluding overhead, which we assume is $O(1)$). This gives us a search time of $O(\log n)$.

To combine this with navigable small-world networks: we build layers of graphs up sequentially. The bottom layer (as in skip lists) is the graph where the vertices are our points and two points have an edge if they are an ε-neighbour. Vertices have a fixed probability of moving up to the next layer, and the way we choose neighbours in higher layers is by using a heuristic that favours diverse connections.

Hierarchical navigable small-world networks are a relatively new technique for nearest neighbour search, and as such has found little application to problems such as persistent homology. We emphasise this as a material for further research, comparing its use for Vietoris-Rips construction against other established techniques. We will not cover any more of this algorithm here.

4.1.2 Expansion methods

Methods presented here are introduced by Zomorodian [11], as well as the complexity analysis.

The expansion of a 1-skeleton effectively *fills in* higher dimensional simplices whose faces are present, and assigns a weight to these simplices. That is, for a graph G, calculate the clique complex $\mathrm{Cl}(G)$ with a corresponding weight function $w : \mathrm{Cl}(G) \to \mathbb{R}_{\geq 0}$.

We have seen that we can easily compute the weights on the simplices (Lemma 2.2.20), thus we focus on the construction of the clique complex.

Problem 4.1.5 (CliqueComplex).
Instance: let $G = (V, E)$ be a graph.
Question: compute $\mathrm{Cl}(G)$.

We are now discarding all embedding information of S, and focusing solely on the topological features of the 1-skeleton given by the skeleton method.

We first introduce the *inductive expansion algorithm.* We fix some arbitrary ordering on the vertices of our 1-skeleton, to give meaning to the LowerNghbrs function in Algorithm 4.1.2.

Algorithm 4.1.2 The inductive expansion algorithm.

```
function LowerNghbrs(graph G, vertex u ∈ G)
    return {v ∈ V(G) : v < u, {u, v} ∈ E(G)}
end function
function InductiveExpansion(graph G, level k ∈ {2, 3, ...})
    K ← V(G) ∪ E(G)
    for i ← 1 to k do
        for each i-simplex τ ∈ K do
            N ← ∩_{u∈τ} LowerNghbrs(G, u)
            for each v ∈ N do
                K ← K ∪ {τ ∪ {u}}
            end for
        end for
    end for
end function
```

We briefly share some intuition of Algorithm 4.1.2 (a sketch of correctness): the skeleton of this code follows closely to the brute force method. For each i-simplex, we check if it forms the face of a $(i+1)$-simplex (using the fixed ordering to avoid double checking simplices). If it does, we add that $(i+1)$-simplex to the complex.

Analysing the complexity of expansion algorithms is tricky as we have an output that has exponential size (in the size of the input), but we can form a rough lower bound on the first level of expansion: from line segments to triangles. For each edge, we look at the (lower) neighbours of the endpoints to compute the set of shared (lower) neighbours. We then perform a constant time operation on each of the shared neighbours, thus we get the lower bound $\sum_{v \in V} (\deg v)^2$.

This was was bounded by Caen [15]:

$$\frac{4|E|^2}{|V|} \leq \sum_{v \in V} (\deg v)^2 \leq |E| \left(\frac{2|E|}{|V|-1} + |V| - 2 \right).$$

Thus, the first level of expansion runs in time $O(n^3)$. We repeat the subsequent levels by a similar sum on the degrees on the triangles, tetrahedron, etc.; however, we typically fix the number of dimensions to expand into, so we take the overall time complexity to be $O(n^3)$. Further research is needed into the restriction of this bound.

We highlight a key inefficiency in the inductive algorithm (to motivate the incremental algorithm): when we compute the neighbours of a simplex, we repeat computations already done for its faces. Thus, instead of inducting on dimension, we can incrementally add vertices and construct cofaces for which the vertex is maximal.

Algorithm 4.1.3 The incremental expansion algorithm, where we induct on the dimension.

```
function AddCofaces(graph G, level k, simplex τ, N, K)
    if dim τ ≥ k then
        return
    end if
    for each v ∈ N do
        σ ← τ ∪ {v}
        M ← N ∩ LowerNghbrs(G, v)
        AddCofaces(G, k, σ, M, K)
    end for
end function
function IncrementalExpansion(G, k)
    K ← ∅
    for each u ∈ V(G) do
        N ← LowerNghbrs(G, u)
        AddCofaces(G, k, {u}, N, K)
    end for
end function
```

In Algorithm 4.1.3, we start with an empty complex. We than add all the cofaces of each vertex for which the vertex is maximal. The initial calls of AddCofaces in IncrementalExpansion will add all simplices (of the k-skeleton) for which u is a maximal vertex.

Further restriction of the worst-case running time (than the one presented for the inductive algorithm) is yet to be seen.

4.1.3 Comparison of methods

As discussed above, computing worst-case running time bounds is difficult for both expansion methods. Thus, we empirically examine the running time of the methods presented.

We first introduce the dataset we will use to compare the methods outlined above: the noisy unit circle (NUC) with fluctuation α. Points are taken uniformly at random from the annulus with outer radius $1+\alpha/2$ and inner radius $1-\alpha/2$ (we will take $\alpha = 0.1$ unless otherwise stated). The underlying population of this test set is homeomorphic to the open cylinder $S^1 \times (0, 1)$.

Three experiments were conducted on an instance of NUC to investigate the performance of the methods outlined above for $\hat{\varepsilon}$-VRFilt:

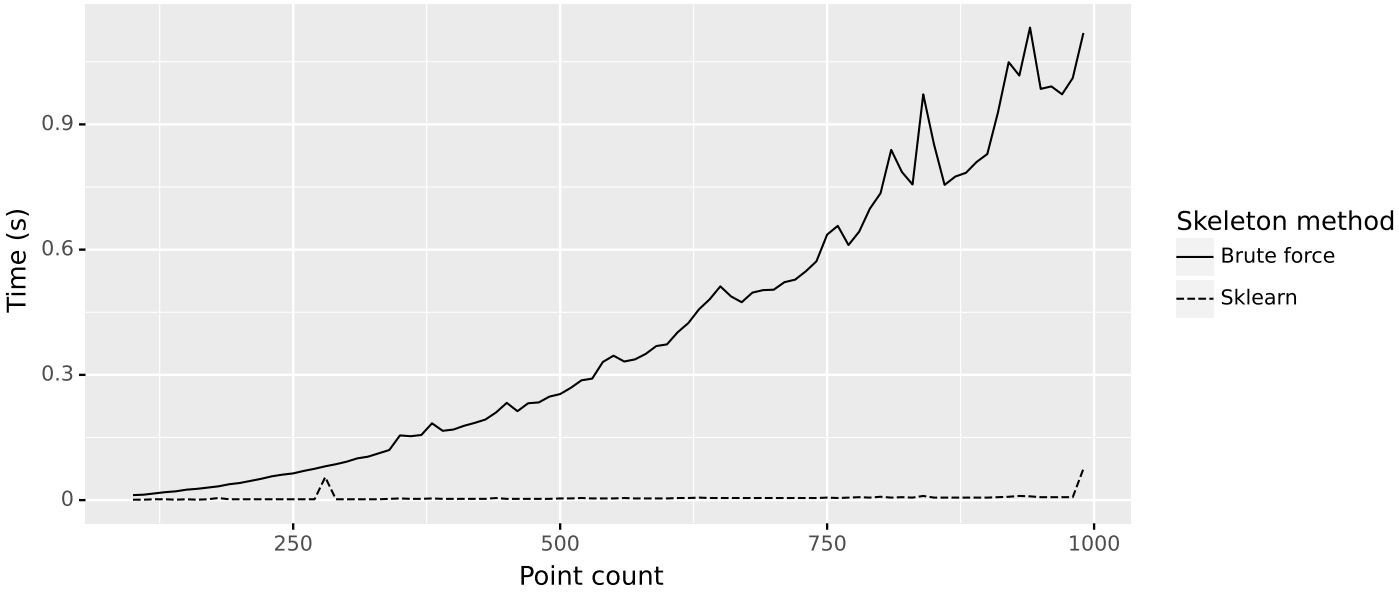


Figure 4.1.1: A plot of the performance of two skeleton methods (with varying point count): the brute force method and the method provided by sklearn. The test set is the noisy unit circle (with radius fluctuation 0.1) with $\varepsilon = 0.01$.

(i) investigating the running time of the brute force skeleton method and the method provided by sklearn [16] with varying point count;

(ii) investigating the running time of the three expansion methods with varying $\hat{\varepsilon}$; and

(iii) investigating the running time of the two (non-brute force) expansion methods with varying $\hat{\varepsilon}$.

For experiment (i), we take $\hat{\varepsilon} = 0.01$. Note that the method provided by sklearn does not use the state-of-the-art hierarchical navigable small-world networks. In fact, it does not use a single method. The package will scan the underlying dataset and pick the optimal algorithm, one possible algorithm uses BBD-trees as mentioned from Arya et al. [12]. For experiments (ii) and (iii) our test set consists of 1000 points and we calculate expansion up to 10 dimensions.

Figure 4.1.1 shows the results of experiment (i), and we can clearly see that the brute force methods has poor performance. This figure does not show the performance of the sklearn method at higher point counts: the sklearn method (typically) runs in under a second for test sets of size 1 000 000 and higher (depending on $\hat{\varepsilon}$). From this, it is reasonable to conclude that the expansion method is the bottleneck in calculating the Vietoris-Rips filtration (the subsequent experiments reaffirm this claim).

Figure 4.1.2 shows the results of experiment (ii), and reaffirms what the reader may expect: brute force algorithms are not very efficient. It is interesting to note that, at the range of $\hat{\varepsilon}$ plotted here, the inductive algorithm has better performance than the incremental algorithm (although they are similar).

Figure 4.1.3 shows the results of experiment (iii) and shows that the incremental algorithm outperforms the inductive algorithm significantly, even though we saw the inductive algorithm perform slightly better for lower values of $\hat{\varepsilon}$.

To conclude, the incremental algorithm is the best investigated algorithm for expansion, and the sklearn methods outperform the brute-force solution. We again highlight the need for further research investigating hierarchical navigable small-world networks in this context.

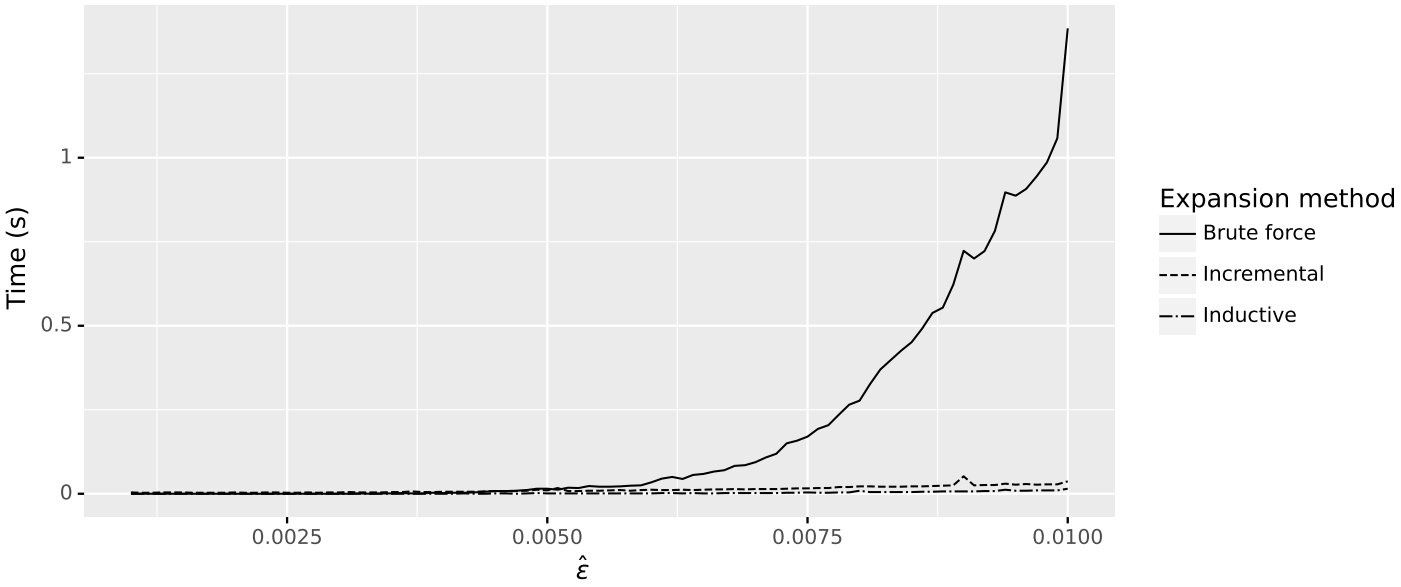


Figure 4.1.2: A plot of the performance of three expansion methods (with varying ε): the brute force method, the inductive algorithm, and the incremental algorithm. The test set is is the noisy unit circle (with radius fluctuation 0.1) and the expansion is calculated up to 10 dimensions.

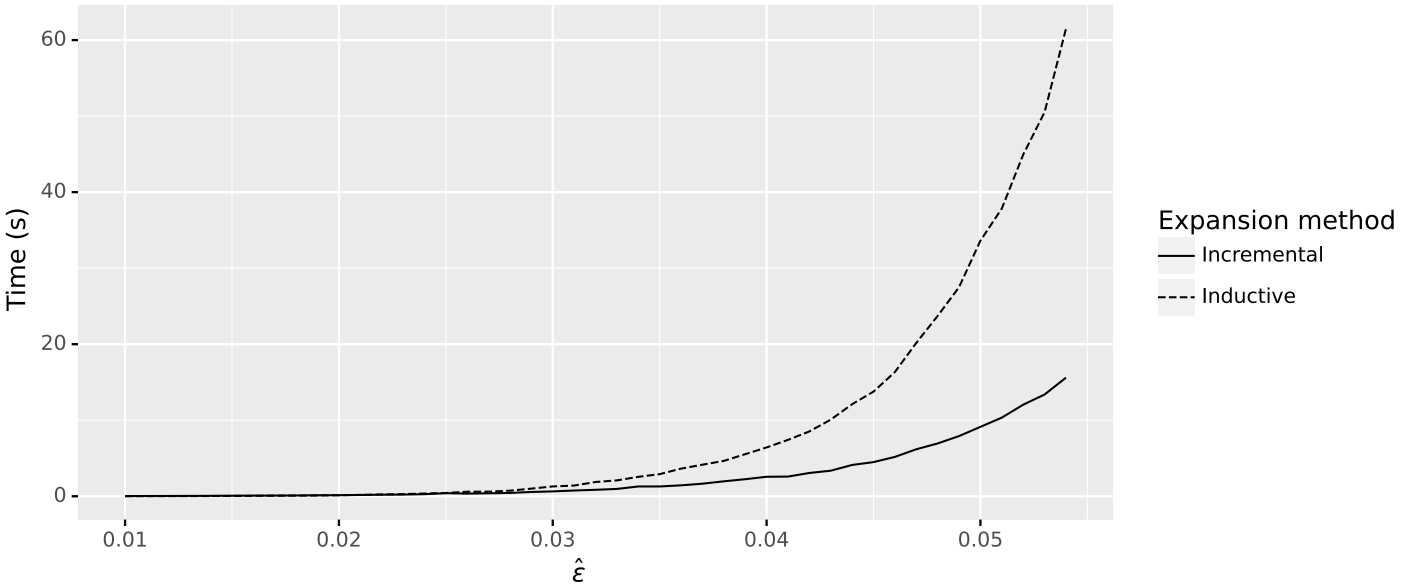


Figure 4.1.3: A plot of the performance of two expansion methods (with varying ε): the inductive algorithm and the incremental algorithm. The test set is is the noisy unit circle (with radius fluctuation 0.1) and the expansion is calculated up to 10 dimensions.

4.2 Computing persistent homology

This section outlines the standard algorithm for computing persistent homology, and some speed-ups that are utilised by Ripser [3]. Some experiments are conducted to verify speed-ups, and as we add each speed-up we compare it to the previous original algorithm. We introduce the algorithms in the order:

(i) the standard algorithm (no speed-ups);

(ii) sparse matrix representation;

(iii) reduction by killing (clearing); and

(iv) reducing the cohomology boundary matrix.

Homology will be computed with coefficients in $\mathbb{F}_2$, given the amenability to a fast algorithm (our algorithms may also be adapted to any field). It is generally agreed that the torsional subgroups of the homology groups with coefficients in some non-divisible group yield no practical information. There are also many challenges to overcome for computing persistent homology over a non-divisible group; for example, our structure theorem for persistence modules does not hold for algebraic structures that are not fields (such as $\mathbb{Z}$). We pose this as a topic of further research, as not much material exists on its potential uses. Note that the notion of *oriented simplices* does not make much sense over $\mathbb{F}_2$, as $\overline{1} = \overline{-1}$.

In the following exposition, we assume homology is calculated up to some fixed dimension $d \in \mathbb{N}$. Also, for a matrix M, we will denote the ith column of M by M_i.

We have discussed the existence of a compatible ordering of simplices for a filtered simplicial complex. To make our exposition clearer, we will be considering only *maximal* filtration; that is, filtrations of the form $\{K_i\}_{i=0}^n$ where $K_i = \bigcup_{i=1}^n \{\sigma_i\}$. Thus moving up the filtration by one level corresponds to adding a single simplex. We show how this method can be extended to any filtered simplicial complex.

Let $\mathbb{L} = \{L_i\}_{i=0}^m$ be any filtered simplicial complex and let $\sigma_1, \ldots, \sigma_n$ be a compatible simplex ordering. We form the maximal filtration $\mathbb{K}$ by

$$\mathbb{K} = \{K_i\}_{i=0}^n, \qquad K_i = \bigcup_{i=1}^{n} \{\sigma_i\}.$$

We then construct the surjective map

$$\begin{aligned} f : \{1, \ldots, n\} &\to \{0, \ldots m\}, \\ i &\mapsto \min\{j : \sigma_i \in L_j\}. \end{aligned}$$

This induces a map

$$\begin{aligned} f_* : \mathrm{Pers}(H_*(\mathbb{K})) &\to \mathrm{Pers}(H_*(\mathbb{L})) \\ (i, j) &\mapsto (f(i), f(j)) \end{aligned}$$

(with $f_*(i, \infty) = (f(i), \infty)$) which is a bijection.

4.2.1 Standard algorithm

We first introduce our problem formally.

Problem 4.2.1 ($\mathbb{F}_2$-PersBarcodes).
Instance: let $\mathbb{K}$ be a maximal (finite) filtered simplicial complex.
Question: compute $\mathrm{Pers}(H_*(\mathbb{K}; \mathbb{F}_2))$.

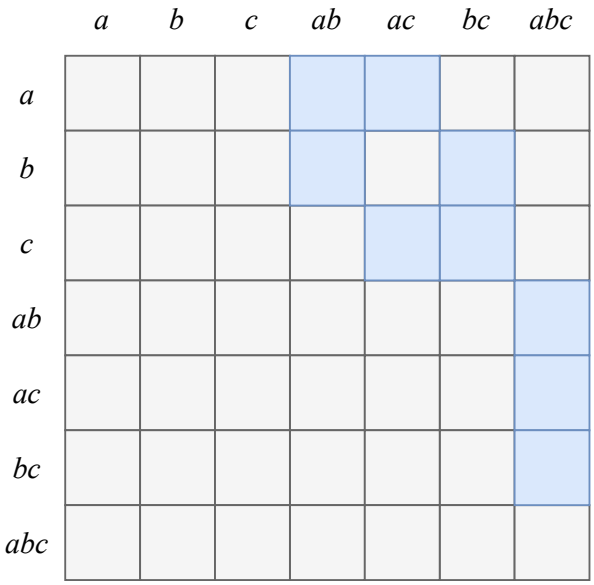


Figure 4.2.1: The boundary matrix for the simplex ordering $(a, b, c, ab, ac, bc, abc)$. A grey cell corresponds to a 0, and a blue cell corresponds to a 1.

Let K be a finite simplicial complex and $\mathbb{K} = \{K_i\}_{i=0}^n$ be a maximal filtration of K. Let $\sigma_1, \ldots, \sigma_n$ be the (unique) ordering of the simplices of K induced by $\mathbb{K}$.

We consider the simplicial chain complex of K with coefficients in $\mathbb{F}_2$: $C(K; \mathbb{F}_2) = \{(K\big|_i, \partial_i)\}_{i=0}^{\dim K}$. Let $n_i = \big|K\big|_i\big|$. By picking canonical bases for each $K\big|_i$, we may represent ∂_i as a $n_{i-1} \times n_i$ matrix with entries in $\mathbb{F}_2$ for each $i \geq 1$. We combine the boundary maps of each chain group into a single map, $\partial : \bigoplus_{i=0}^{\dim K} C_i \to \bigoplus_{i=0}^{\dim K} C_i$. To do this, we pick our basis as the simplex ordering $\sigma_1, \ldots, \sigma_n$ and let ∂ be a $n \times n$ matrix defined by

$$\partial[i, j] = \begin{cases} 1 & \text{if } \sigma_i \text{ is a face of } \sigma_j, \\ 0 & \text{otherwise.} \end{cases}$$

This map agrees with the boundary maps ∂_i of the simplicial chain complex, and ∂ is strictly upper triangular. We call ∂ the *boundary matrix* of $\mathbb{K}$.

Example 4.2.2. Figure 4.2.1 is a visualisation the boundary matrix for the simplex ordering $(a, b, c, ab, ac, bc, abc)$.

The prevalent method for solving $\mathbb{F}_2$-PersBarcodes this problem is matrix reduction (effectively a variant of the Smith normal form algorithm). We let $M \in M_n(\mathbb{F}_2)$ and define $\text{low}_M(i) = \max\{j : M[i, j] \neq 0\}$; that is, $\text{low}(i)$ is the largest row index of a non-zero entry in column i (we let it take some sentinel value otherwise). A column operation of the form $M_i \leftarrow M_i + M_j$ is called *reducing* if $j < i$ and $\text{low}_M(i) = \text{low}_M(j)$. A matrix is called *reduced* if no more reducing operations can be performed on it. A matrix R is a *reduction* of M if R is reduced and arises from a sequence of reducing operations from M.

Lemma 4.2.3. *Let K be a finite simplicial complex and $\mathbb{K} = (K_i)_{i=0}^m$ be a filtration of K, such that $K_m = K$. Let ∂ be the boundary matrix of $\mathbb{K}$ (with basis some ordering of the simplices compatible with $\mathbb{K}$) and R a reduction of ∂. Then $(i, j) \in \text{Pers}(\mathbb{K})$ if and only if*

(i) $j \neq \infty$ and $\text{low}_R(j) = i$;

(ii) $j = \infty$ and $\text{low}_R^{-1}(i) = \varnothing$.

Proof. We can consider the reduction algorithm as a variant of a Smith normal form algorithm (although the reduction matrix is not quite in Smith normal form), and so by the structure we gave the persistence module in Section 2.1.3 we see that the reduction algorithm indeed gives us the persistent barcodes as above. □

Lemma 4.2.3 shows us that it is sufficient to find a reduction of the boundary matrix to compute the persistent barcodes.

Before looking at methods for computing the reduction, we look at how to read off the persistent barcodes. We split $\mathrm{Pers}(H_*(\mathbb{K}))$ into two sets:

$$\mathrm{Pers}(H_*(\mathbb{K})) = \mathrm{Pers}_0(H_*(\mathbb{K})) \cup \mathrm{Pers}_\infty(H_*(\mathbb{K}))$$

where $\mathrm{Pers}_0(H_*(\mathbb{K}))$ comprises of the finite intervals (i, j) and $\mathrm{Pers}_\infty(H_*(\mathbb{K}))$ comprises of the infinite intervals (i, ∞). We can compute $\mathrm{Pers}_0(\mathbb{K})$ by iterating over each $j \in \{1, \ldots, n\}$ and computing $\mathrm{low}_\partial(j) = i$, which forms $(i, j) \in \mathrm{Pers}_0(\mathbb{K})$. For $\mathrm{Pers}_\infty(H_*(\mathbb{K}))$, we just note the $i \in \{1, \ldots, n\}$ that does not have a corresponding $j > i$ such that $\mathrm{low}_\partial(j) = i$. The running time of this interpretation depends on the implementation of the low function; for example, using a hash table gives us a constant time evaluation and $O(n)$ time to update after a column operation, which would give us a constant time interpretation.

We now introduce the problem of finding the reduction of ∂.

Problem 4.2.4 ($\mathbb{F}_2$-PersReduction).
Instance: let ∂ be the boundary matrix of some maximal filtered simplicial complex.
Question: compute a reduction R of ∂.

Algorithm 4.2.1 The standard reduction algorithm for $\mathbb{F}_2$-PersReduction.

```
1: function Low(n × n matrix R, i ∈ {1, ..., n})
2:     return lowest non-zero entry of column i
3: end function
4: function IsColReduced(n × n matrix R, i ∈ {1, ..., n})
5:     return whether there is i' < i such that Low(R, i') = Low(R, i)
6: end function
7: function StdReduction(n × n boundary matrix ∂)
8:     R ← ∂
9:     for i ← 1 to n do
10:        while not IsColReduced(R, i) do
11:            for j = 1 to i do
12:                if Low(R, i) = Low(R, j) then
13:                    add column j to i and break for loop
14:                end if
15:            end for
16:        end while
17:    end for
18:    return R
19: end function
```

Algorithm 4.2.1 is the standard reduction algorithm for persistent homology, first introduced by Edelsbrunner, Letscher, and Zomorodian [2]. We will refer to this algorithm as S.

The running time of Algorithm 4.2.1 is $O(n^3)$ as

(i) adding a column to another column runs in $O(n)$ time; and

(ii) calling the function Low can be achieve in $O(1)$ time using a hash table, this hash table can be updated in $O(n)$ for each column operation.

We are clear on the correctness of such a reduction, if the algorithm terminates then it is clear we meet the requirements of R being a reduction. But we must prove that Algorithm 4.2.1 terminates. On to the main loop (line `9` onwards), we first claim that this terminates. Although not immediately obvious, we assert that IsColReduced(i) will evaluate true by repeating `11` to `15`. Trivially, IsColReduced(1) is always true. We now let $i > 1$ and assume that IsColReduced(j) is true for each $j \in \{1, \dots i\}$. As IsColReduced(i) is false, there is $i_1 < i$ such that Low(i_1) = Low(i). So we add the i_1th column to the ith column, making the new value of Low(i) strictly less than Low(i_1) (as, by definition, every entry below Low(i_1) is zero). To appreciate this, we recall that we are in $\mathbb{F}_2$. If IsColReduced(i) is true, we are done; otherwise, there is $i_2 < i$ such that Low(i_2) = Low(i) < Low(i_1). Again, we perform the appropriate column operation. We continue this operation, and as there are only a finite number of rows (that is, a finite number of unique values for Low), then we must reach a point at which IsColReduced(i) is false. The final conclusion is clear by induction.

4.2.2 Spare matrix representation

The first speed-up we look at utilises the *sparseness* of the boundary matrix. To motivate, we consider a triangulation of S^n. Let K be a simplicial complex consisting of the proper faces of some $(n+1)$-simplex and $\mathbb{K}$ be a maximal filtration of K where we add the 0-simplices, then the 1-simplices, and so on. We note that there is $2^{n+2} - 2$ simplices. For $n = 2$, Figure 4.2.2 shows a boundary matrix for $\mathbb{K}$ (recall that we are in $\mathbb{F}_2$, a blue square represents a 1 and a grey square represents a 0).

Figure 4.2.2 shows a *sparse matrix*; most of the entries are zero. Boundary matrices for general filtered simplicial complex are also sparse. Although most entries are 0, we are still storing each one: the standard algorithm stores a value for each $(i, j) \in (1, \dots, n)^2$. There are many efficient methods for storing a sparse matrix, and most make use of assuming all entries are zero, then provide additional information for the non-zero entries. We will provide one such way of doing this, which is suited to our problem.

Let $A \in M_n(\mathbb{F}_2)$ be a matrix. For each column $j \in \{1, \dots n\}$, we store a set (hash table), denoted S_j, such that $i \in S_j$ if and only if $A[i, j] = 1$. This allows constant time for many elementary operations (such as membership queries, insertion, and deletion) and constant space. Figure 4.2.3 shows an example of this representation. We further comment on the running time of operations with this data structure, specific to the reduction algorithm. For $i \in \{1, \dots, n\}$, denote the number of non-zero entries in column i by m_i, and we have $O(m_i) = O(d)$.

(i) Column operations: let $i, j \in \{1, \dots, n\}$. Then the time to add column j to i is $O(m_i + m_j)$. Given that we typically fix the dimensions to compute homology to, we can take this as constant (or as $O(d)$ for maximum dimension d).

(ii) Retrieve the last non-zero entry in column: let $i \in \{1, \dots, n\}$. Finding the last non-zero entry in i is $O(m_i)$.

As we are just modifying the underlying data structure, the algorithm does not change significantly and thus will not be repeated here. We refer to the standard algorithm with sparse matrix representation as SS.

This analysis provides a convincing argument the effectiveness of this speed-up. To quantify this, an experiment was conducted in which measured the running time for both loading (that is, taking a filtration and computing its boundary matrix) and the matrix reduction for the

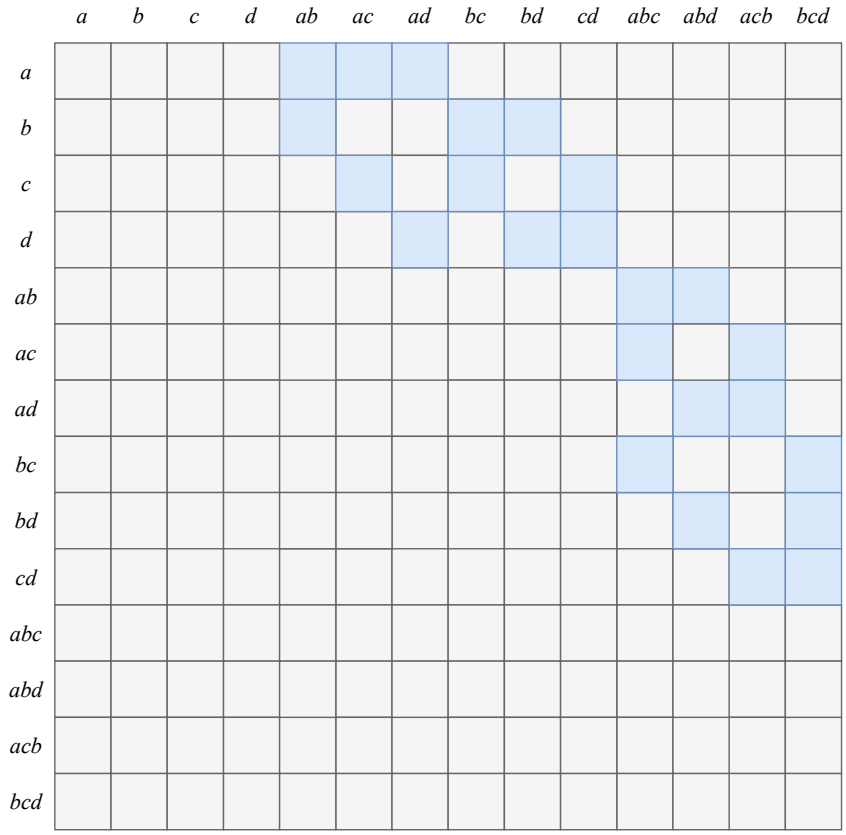


Figure 4.2.2: A sparse boundary matrix for a filtration of a triangulation of S^2. A grey square represents a 0 entry in the matrix, and a blue square represents a 1.

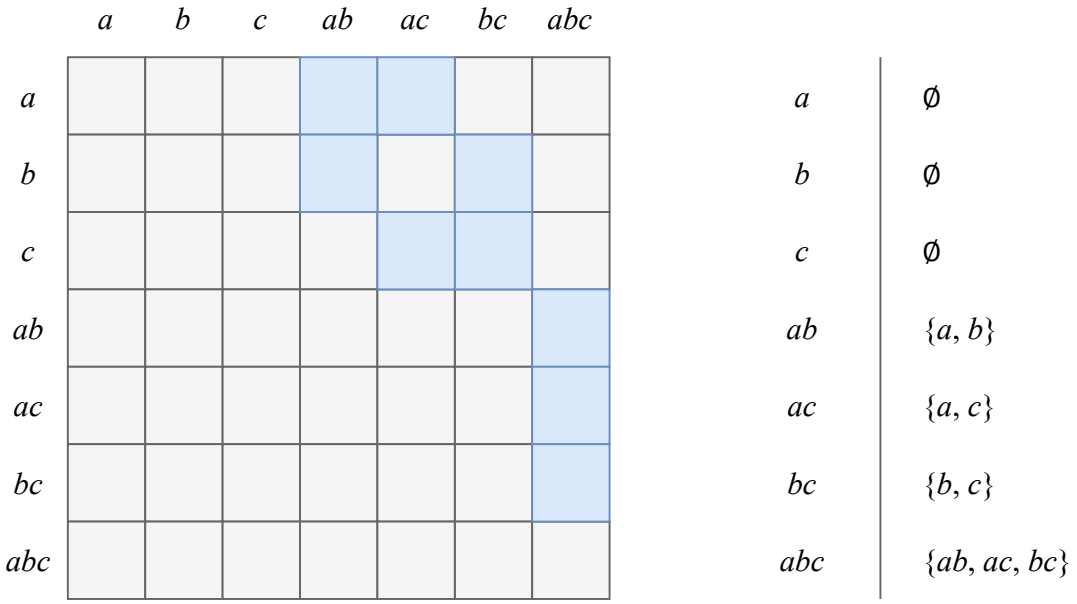


Figure 4.2.3: On the left, the boundary matrix for the simplex ordering $(a, b, c, ab, ac, bc, abc)$. On the right, the sparse matrix representation. A grey square represents a 0 entry in the matrix, and a blue square represents a 1.

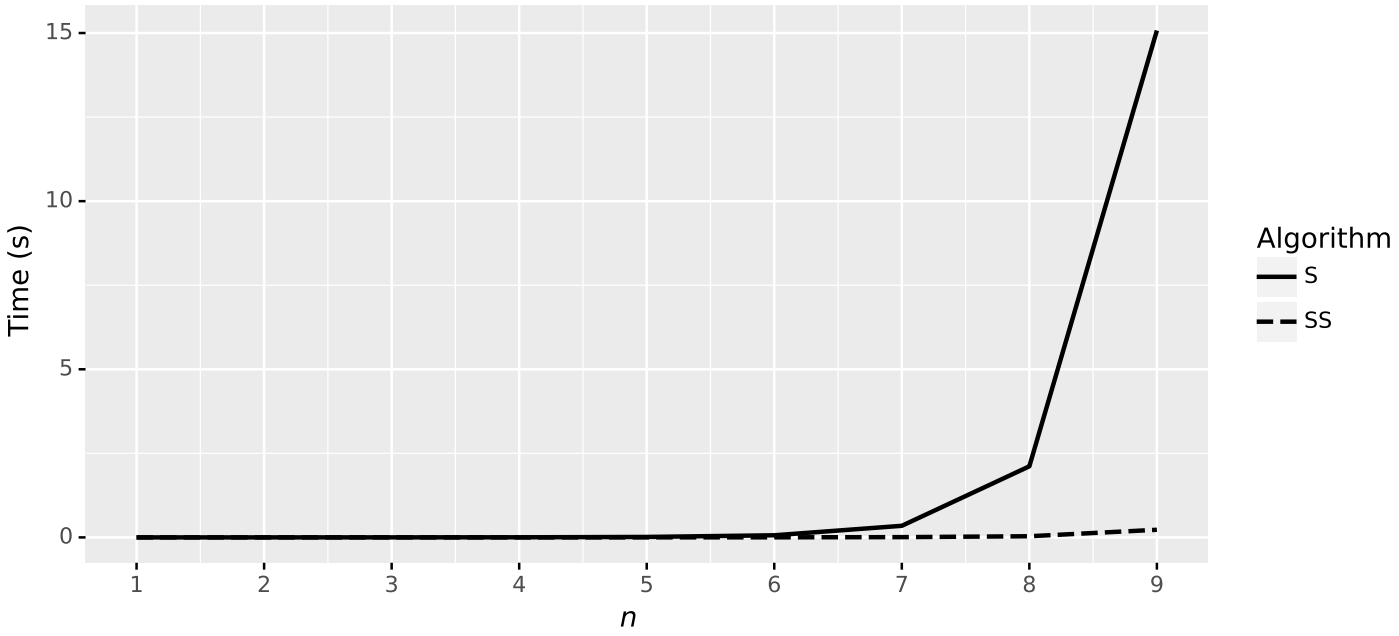


Figure 4.2.4: A plot of the running time of the SS algorithm versus the S algorithm for filtrations of triangulations of S^n.

standard triangulation of S^n for $n \in \{1, \ldots, 9\}$. The running times can be seen in Figure 4.2.4 and the loading running times can be seen in Figure 4.2.5.

From this experiment, we see that the sparse matrix representation significantly decreases the time taken by the reduction; however, it does not decrease the loading time. This is to be expected as we still need to iterate over every the facets of every simplex; however, we note that this input serves as a worse case for loading time, so we would expect a large running time for loading compared to other inputs.

Another experiment was conducted using a Vietoris-Rips complex (with $\hat{\varepsilon} = 0.01$) constructed from point-cloud data sampled from a noisy unit circle (radius fluctuation 0.1). Each algorithm was executed on a complex constructed from n points, where $n \in \{100, 200, \ldots, 1500\}$. Figure 4.2.6 shows the running time of both of these algorithms, and it is clear that the sparse representation is still significantly faster.

We note that there are other data structure choices that could be made instead of hash tables. For example, for each column we may instead store some sorted data structure. If we use a list sorted by index then we can retrieve the lowest non-zero entry in constant time, but we have $O(n)$ time for insertion, deletion, and membership queries. Binary search trees could also be used, which would allow us to find the lowest non-zero entry in $O(\log n)$ average time, as well as $O(\log n)$ average time for insertion, deletion, and membership queries. Utilising self-balancing tree structures (such as B-trees), the worst-case time can also be brought down to $O(\log n)$.

We highlight the use of various data structures for storing a boundary matrix to be a topic of further research.

4.2.3 Reduction by killing

Reduction by killing allows us to *clear* (set to $\mathbf{0}$) certain columns throughout the reduction process, allowing us to avoid unnecessary column operations that would lead the column to being zero anyway. This technique was first introduced by Chen and Kerber [17].

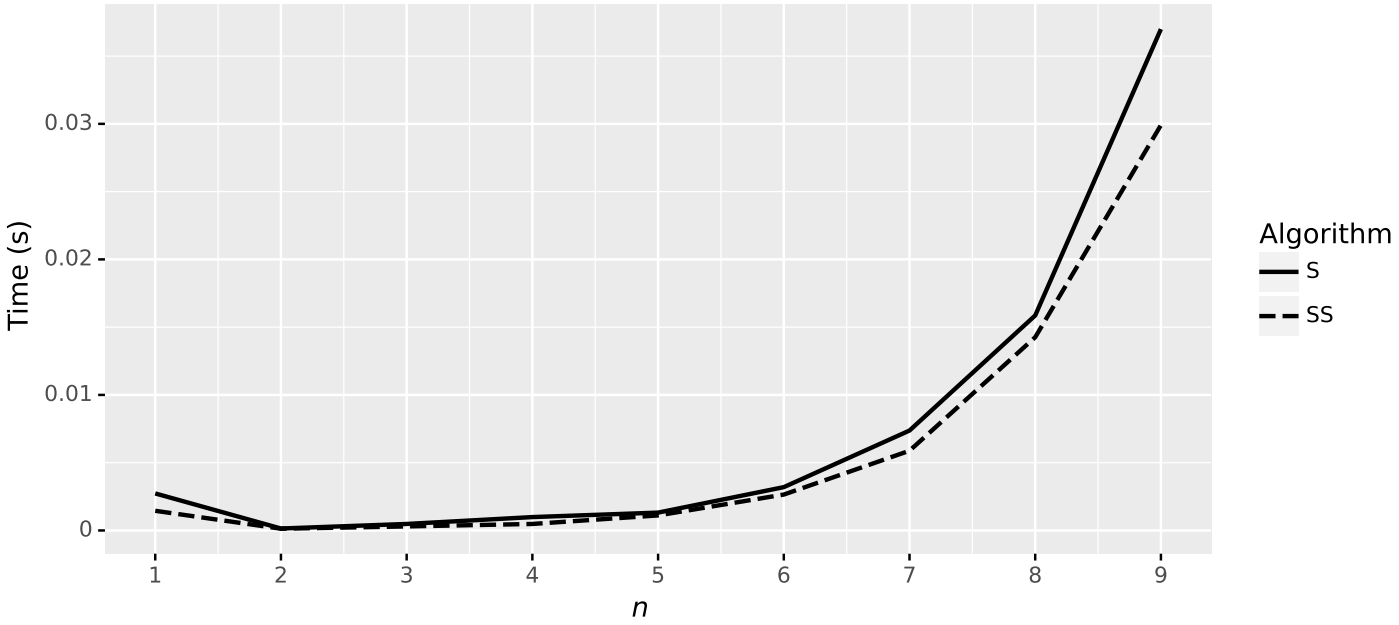


Figure 4.2.5: A plot of the running time of the construction of the boundary matrix (loading time) using the standard matrix representation versus the sparse matrix representation, for filtrations of triangulations of S^n.

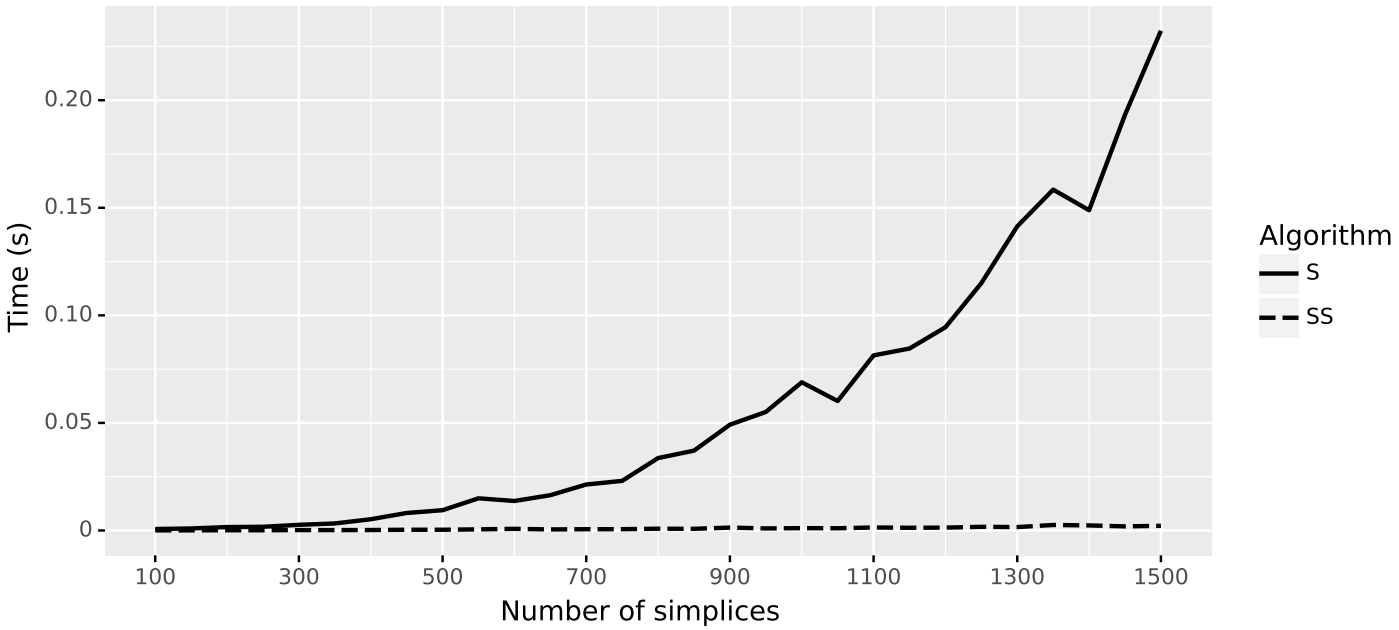


Figure 4.2.6: A plot of the running time of the SS algorithm versus the S algorithm for Vietoris-Rips complexes constructed from noisy unit circle data.

We recall that persistent homology aims to track the changes to the homology groups of a chain complex as we move through a filtration. We now formalise how simplices may change the homology groups.

Definition 4.2.5 (Positive and negative simplices)**.** Let $\mathbb{K} = \{K_i\}_{i=0}^n$ be a maximal filtered simplicial complex of a simplicial complex K with induced simplex ordering $\sigma_1, \ldots, \sigma_n$. Let $\iota_i : K_i \hookrightarrow K_{i+1}$ be the inclusion map and $\bar{\iota}_i : H_*(K_i) \to H_*(K_{i+1})$ the induced map on homology.

(i) $\sigma_i \in K$ is said to be a *positive simplex* if its addition creates a homology class. That is, for all $[c] \in H_*(K_{i-1})$ we have $\bar{\iota}_{i-1}([c]) \neq [\sigma_i] \in H_*(K_i)$.

(ii) $\sigma_i \in K$ is said to be a *negative simplex* if it its addition destroys a homology class. That is, there is $[c], [c'] \in H_*(K_{i-1})$ with $[c] \neq [c']$ such that $\bar{\iota}_{i-1}([c]) = \bar{\iota}_{i-1}([c']) = [\sigma] \in H_*(K_i)$.

We can pair each negative simplex $\sigma_j \in K$ with the positive simplex $\sigma_i \in K$ $(i < j)$ that created the class that σ_j destroys. We call (i, j) (or (σ_i, σ_j)) a *persistence pair*, and $j - i$ its *index persistence*. It is clear (from the definition of homology) that for a persistence pair (σ_i, σ_j), we have $\dim \sigma_i = \dim \sigma_j - 1$.

Corollary 4.2.6. *Let $\mathbb{K} = \{K_i\}_{i=0}^n$ be a maximal filtered simplicial complex of a simplicial complex K. Then every simplex $\sigma \in K$ is either* positive *or* negative*.*

This is a direct consequence of Lemma 4.2.3.

Note that, although each simplex either creates or destroys a homology class, not every simplex belongs to a persistence pair as described above. In particular, certain simplices create homology classes that persist through to the final complex, $K_n = K$. Such (positive) simplices are called *essential simplices*.

The following is the key observation for reduction by killing.

Lemma 4.2.7. *Let $\mathbb{K} = \{K_i\}_{i=0}^n$ be a maximal filtered simplicial complex of a simplicial complex K with induced simplex ordering $\sigma_1, \ldots, \sigma_n$, let R be the reduction of the boundary matrix of $\mathbb{K}$ (with respect to our ordering), and let $(i, j) \in \mathrm{Pers}(H_*(\mathbb{K}))$. Then $R_i = \mathbf{0}$.*

Proof. As $(i, j) \in \mathrm{Pers}(H_*(\mathbb{K}))$, $\mathrm{low}_R(j) = i$. Suppose $R_i \neq 0$, then $\mathrm{low}_R(i) = i'$ for some $i \in \{1, \ldots, n\}$. But then $(\sigma_{i'}, \sigma_i)$ is a persistence pair, contradicting Corollary 4.2.6. □

So if we discover a negative simplex σ, we can clear the column corresponding to the simplex whose homology class σ kills. However, the standard algorithm S starts at the left side of the boundary matrix and moves to the right, so we will always discover positive simplices first. Thus we do not save any column operations.

To take advantage of Lemma 4.2.7, we instead reduce the boundary matrix right-to-left. We first clear the $d = \dim K$ simplices (from left-to-right), then the $(d-1)$-simplices, and so on.

Algorithm 4.2.2 shows the modified algorithm, and we denote this speed-up with the sparse matrix representation on top of the standard algorithms as SSR.

An experiment was conducted comparing SS and SSR. A Vietoris-Rips filtration was constructed from a sample of 1000 points from the noisy unit circle (radius fluctuation ± 0.05). We run both algorithms from $\varepsilon = 0.01$ to $\varepsilon = 0.06$. Figure 4.2.7 shows the result of this experiment.

It can be seen that SSR significantly decreases computation time as the number of simplices increase, due to the reduction of column operations.

Algorithm 4.2.2 The reduction by killing (SSR) algorithm for $\mathbb{F}_2$-PersReduction.

```
1: function StdReduction(n × n boundary matrix ∂)
2:     R ← ∂
3:     for k ← d to 1 do
4:         for i ← 1 to n do
5:             if simplex i does not have dimension k then
6:                 continue to next simplex
7:             end if
8:             while not IsColReduced(R, i) do
9:                 for j = 1 to i do
10:                    if Low(R, i) = Low(R, j) then
11:                        add column j to i and break for loop
12:                    end if
13:                end for
14:            end while
15:            if R_j ≠ 0 then
16:                clear column low(R, j)
17:            end if
18:        end for
19:    end for
20:    return R
21: end function
```

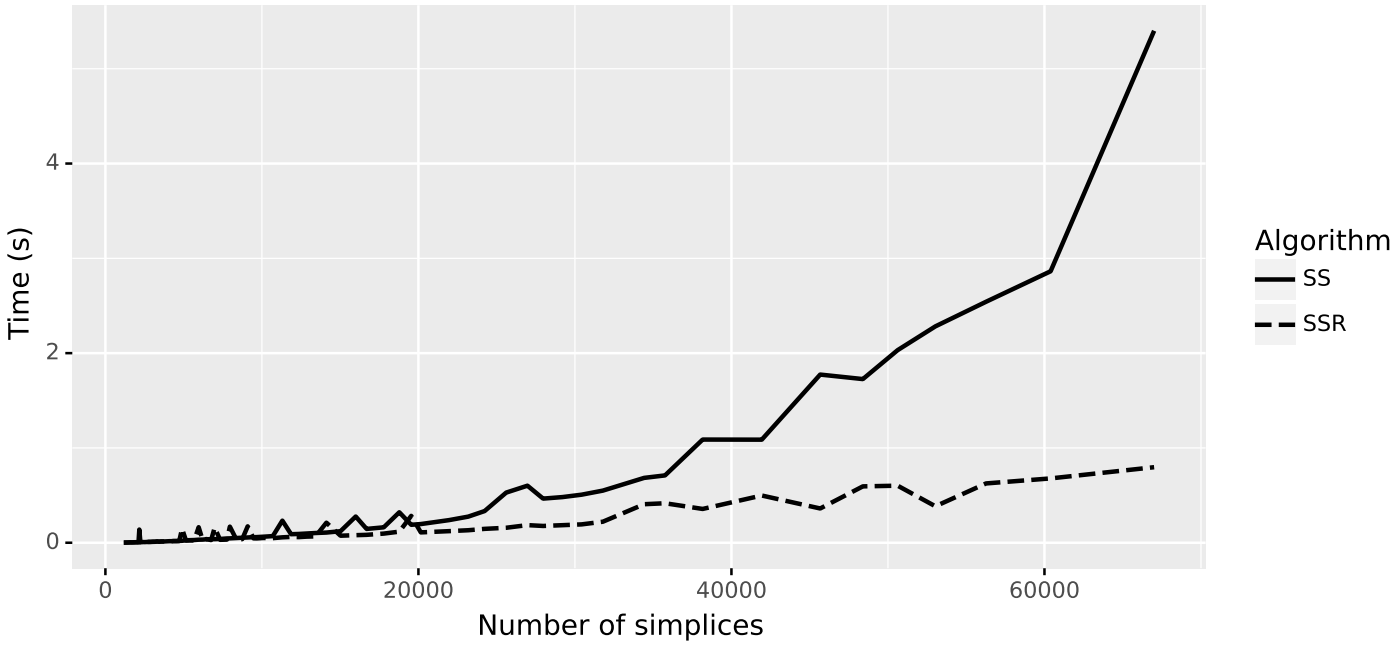


Figure 4.2.7: A plot of the running time of SS and SSR algorithms on a Vietoris-Rips complex constructed from a noisy unit circle.

4.2.4 Reducing the coboundary matrix

The reduction by killing method (clearing) was found to be utilised to a much greater degree when using the cohomology (instead of the homology) of Vietoris-Rips filtrations. De Silva, Morozov, and Vejdemo-Johansson [18] first published this cohomology algorithm; however, it compares the use of cohomology to the standard algorithm without clearing and thus his results may artificially bloat the effectiveness of this speed-up.

There are two main observations to understand this speed-up:

(i) the persistent barcodes from homology and cohomology are identical; and

(ii) computing persistent barcodes on cohomology allows for more clearing.

We have already seen that the persistent barcodes for homology and cohomology coincide (see Lemma 2.1.40).

Let $\mathbb{K} = \{K_i\}_{i=0}^n$ be a maximal filtered simplicial complex and let $\sigma_1, \ldots, \sigma_n$ be the induced simplex ordering. The *coboundary matrix* of $\mathbb{K}$ is defined by

$$\delta[i,j] = \begin{cases} 1 & \text{if } \sigma_{n+1-i} \text{ is a cofacet of } \sigma_{n+1-j}, \\ 0 & \text{otherwise.} \end{cases}$$

δ is simply the *anti-transpose* (flip over the anti-diagonal) of the boundary matrix ∂, and we can construct it in the same time. We note that the basis for this matrix are the *cochains* of the simplicial cochain complex, and thus mapping between the barcodes from persistent cohomology and persistent homology is just reversing the ordering of our basis (and mapping barcodes of the form (∞, i) to the form (j, ∞) appropriately).

We now argue that cohomology allows for more clearing. In fact, this is not true for the general case. It is the properties of the Vietoris-Rips filtration that allow for more clearing.

The clearing optimisation (SSR) sets a column R_i to $\mathbf{0}$ if i is the pivot index of another column R_j. In terms of persistent barcodes, R_i is a *birth column* and R_j is the *death column*. It has already been noted that $\dim(\sigma_i) = \dim(\sigma_j) - 1$. Thus to clear the column of a k-simplex, we must reduce the column of a $(k+1)$-simplex. Conversely, in cohomology, clearing the column a k-simplex requires the reduction of the column of a $(k-1)$-simplex. Thus, as

For Vietoris-Rips filtrations, the degree we calculate persistent homology to is often small, and there are often many more $(d+1)$-simplices than simplices of lower dimension. In homology, clearing is not available for the $(d+1)$-simplices. However, in cohomology, clearing is not available for the 0-simplices. As there are (often) more $(d+1)$-simplices then 0-simplices, cohomology allows for more clearing for Vietoris-Rips filtrations.

We refer to the algorithm utilising sparse matrix representation, reduction by killing, and the cohomology boundary matrix as SSRCoh.

An experiment was conducted comparing SSR and SSRCoh. A Vietoris-Rips filtration was constructed from a sample of 1000 points from the noisy unit circle (radius fluctuation ± 0.05), as in the last experiment. We run both algorithms from $\varepsilon = 0.01$ to $\varepsilon = 0.06$. Figure 4.2.8 shows the result of this experiment, and empirically verifies our predicted speed-up.

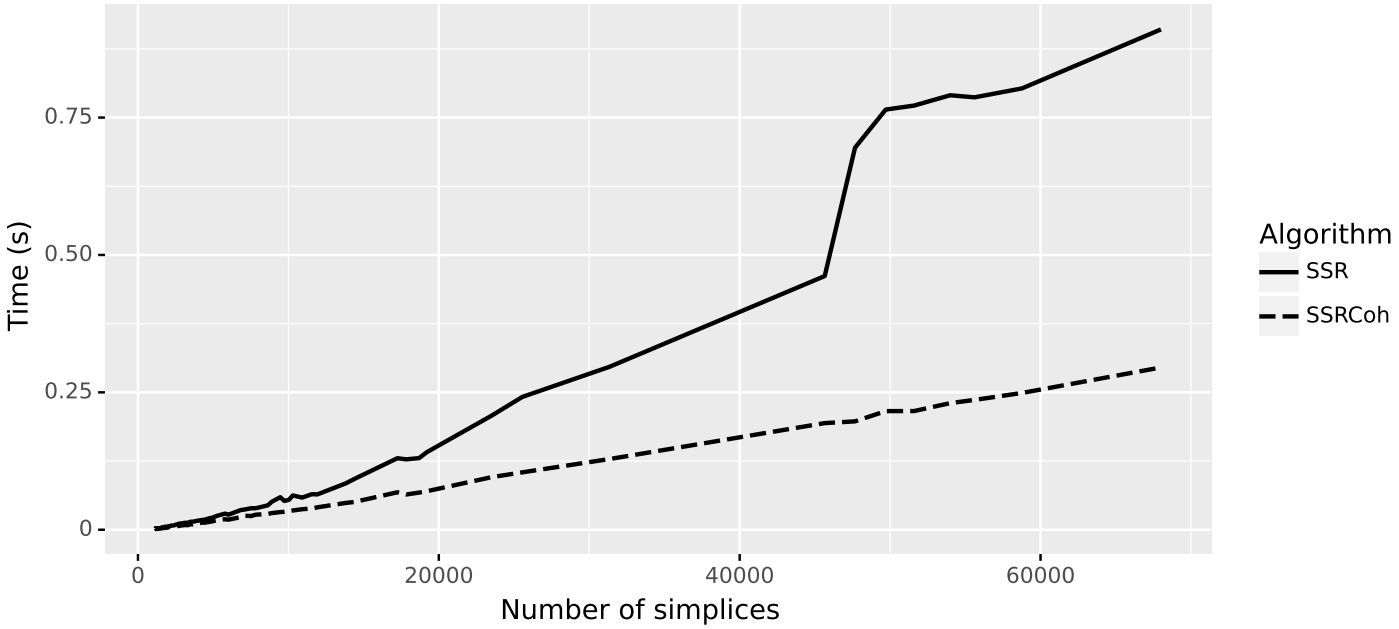


Figure 4.2.8: A plot of the running time of SSR and SSRCoh algorithms on a Vietoris-Rips complex constructed from a noisy unit circle.

Chapter 5

Applications of persistent homology

5.1 Information networks

Information networks are a popular way of capturing complex relationships across many different fields.

Example 5.1.1 (Citation network)**.** A *citation network* is a directed graph $G = (V, E)$ that describes citations within a collection of documents. Each vertex $v \in V$ corresponds to a document, and there is an edge $(v, w) \in E$ if and only if document v cites document w. Thus the out-degree of a document is the number of documents it cites, and the in-degree of a document is the number citations it has.

The use of persistent homology on information networks is becoming more prevalent, and this section serves to outline some of these applications, with a novel suggestions for further research.

5.1.1 Betti numbers as a measure of complexity

An information network may be described as *complex* if it exhibits substantial *non-trivial topological features*. A precise definition does not exist; however, some such features include:

(i) a high *clustering coefficient* (a measure of how *clique-like* each neighbourhood of a graph is is);

(ii) a high assortativity (the tendency for *similar* nodes to attach to each others); and

(iii) a high disassortativity (defined as you would expect).

Some examples of graphs that are not *complex* (we may call these *simple graphs*) include: random graphs and lattice graphs.

1-dimensional Betti numbers may be used as a measure of a graph's *complexity*, although this is not a perfect measure (and given the ambiguity of the definition, a perfect measure can't exist). In fact, Benzekry et al. [19] found a correlation between a specific 1-dimensional Betti number of a protein-protein interaction graph and the survival of cancer patients.

A *protein-protein interaction graph* is a graph $G = (V, E)$ such that each vertex $v \in V$ corresponds to a specific protein (in a biological process), and each edge $(v, w) \in E$ corresponds to some form of interaction between the proteins v and w. Benzekry et al. [19] utilised the power filtration of such graphs in order to measure the complexity of specific proteins, by measuring the drop of a specific Betti number when dropping proteins (vertices).

Lemma 5.1.2. *Let $G = (V, E)$. Then $\beta_1^{1,1}(\mathrm{Pow}(G))$ is the number of independent cycles of length 4 or more in G.*

Proof. $\beta_1^{1,1}(\text{Pow}(G))$ denotes the number of 1-holes that are present in $\text{Cl}(G^1) = \text{Cl}(G)$; that is, the number of linearly independent cycles. If a linearly independent cycle in G has length 3, then it is filled in $\text{Cl}(G)$ as all faces are present. But if a linearly independent cycle has length 4 or more, it is present in $\text{Cl}(G)$. □

Benzekry et al. [19] took a protein-protein interaction graph $G = (V, E)$ consisting of proteins with a high impact of cancer progression and computed $\beta_1^{1,1}(\text{Pow}(G))$, which we will denote β_0. Then, for each protein $v \in V$ they dropped its node from G and calculated $\beta_1^{1,1}(\text{Pow}(G - v))$, which we will denote β_v. Then for each $v \in V$, they calculated

$$\tilde{\beta}_v = \beta_0 - \beta_v,$$

which shows the difference in the Betti number with that specific protein removed. This shows us how much *complexity* is imparted by a specific protein in G, and may give us a good indication of what proteins to target for drug treatment.

Benzekry et al. [19] also did significant exploratory work into the correlation of Betti numbers and clinical results; for example, they found a significant negative correlation between the Betti number discussed and the five year survival rates for a certain cancer network, reaffirming that its indication for protein targets are worthwhile.

A review of this publication highlighted that this Betti number alone was not sufficient in the selection of proteins to target, and the author agreed with this sentiment. Further posing that persistent homology is but another tool to make rational decisions, and should be used in conjunction with other techniques.

5.1.2 Novel: compartmental models of infectious disease

Infectious disease may be modelled on a graph: we form an undirected graph $G = (V, E)$ (called a *contact graph*) where each $v \in V$ corresponds to a person in a population edge $\{v, w\} \in E$ corresponds to two people having contact. We compartmentalise the population into the following categories:

(i) S: susceptible to infection;

(ii) I: infectious; and

(iii) R: recovered.

Thus $V = S \cup I \cup R$. We run the simulation in ticks, with some initial infected population. At each tick $t \in \mathbb{N}$, an infected node (person) $v \in V$ infects each of its susceptible neighbours with some probability $p \in [0, 1]$. After a number of ticks, say $l \in \mathbb{N}$, an infectious node moves to the recovered category. This is called the *SIR model*.

We can extend this model further, we introduce two new categories:

(i) V: vaccinated; and

(ii) VI: vaccinated and infectious.

Vaccinated nodes can still be infected, and be infectious (as suggested), but the probability of being infected and infecting someone else may be considered less than p (if a vaccine if effective). This is called the *SIRV* model.

Figure 5.1.1 shows the result of a SIRV model where vaccines are ineffective, and Figure 5.1.2 shows the result of a SIRV model where vaccines reduce transmission and infections by 50%.

The model whose results are shown in Figure 5.1.2 uses a random vaccination strategy; individuals are chosen at random to be vaccinated. In reality, infectious diseases are much more

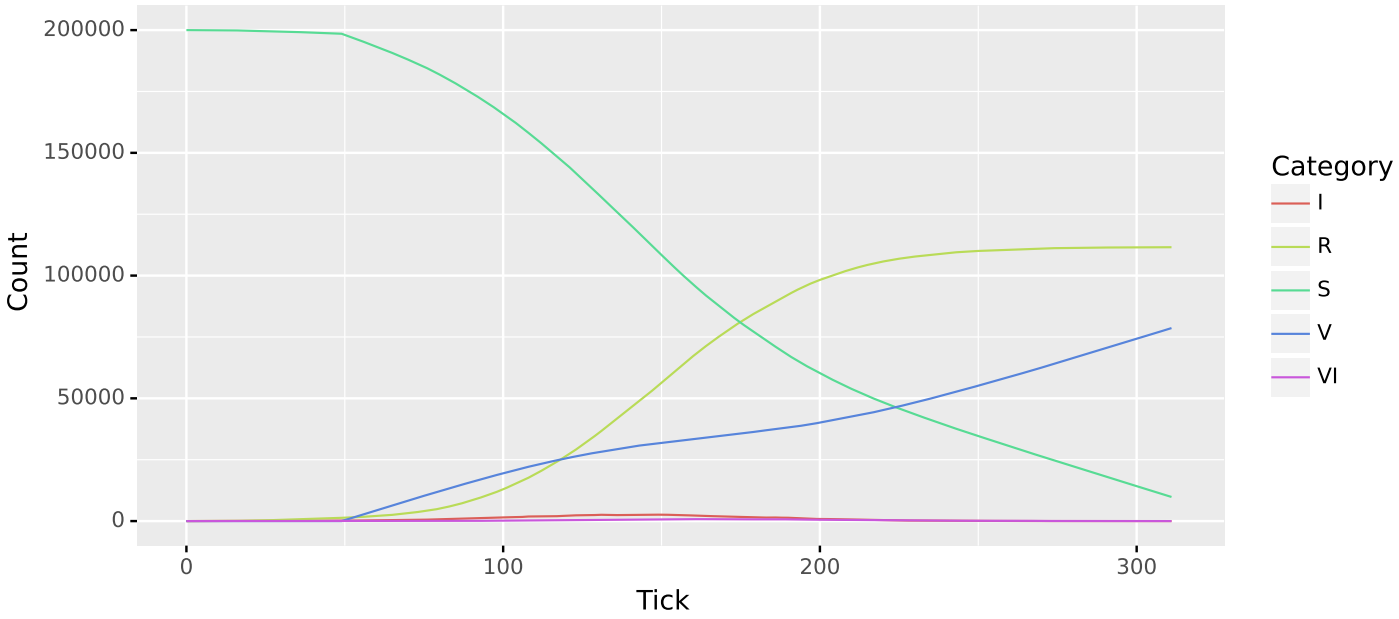


Figure 5.1.1: A plot showing the results of a SIRV model where vaccinations have no effect.

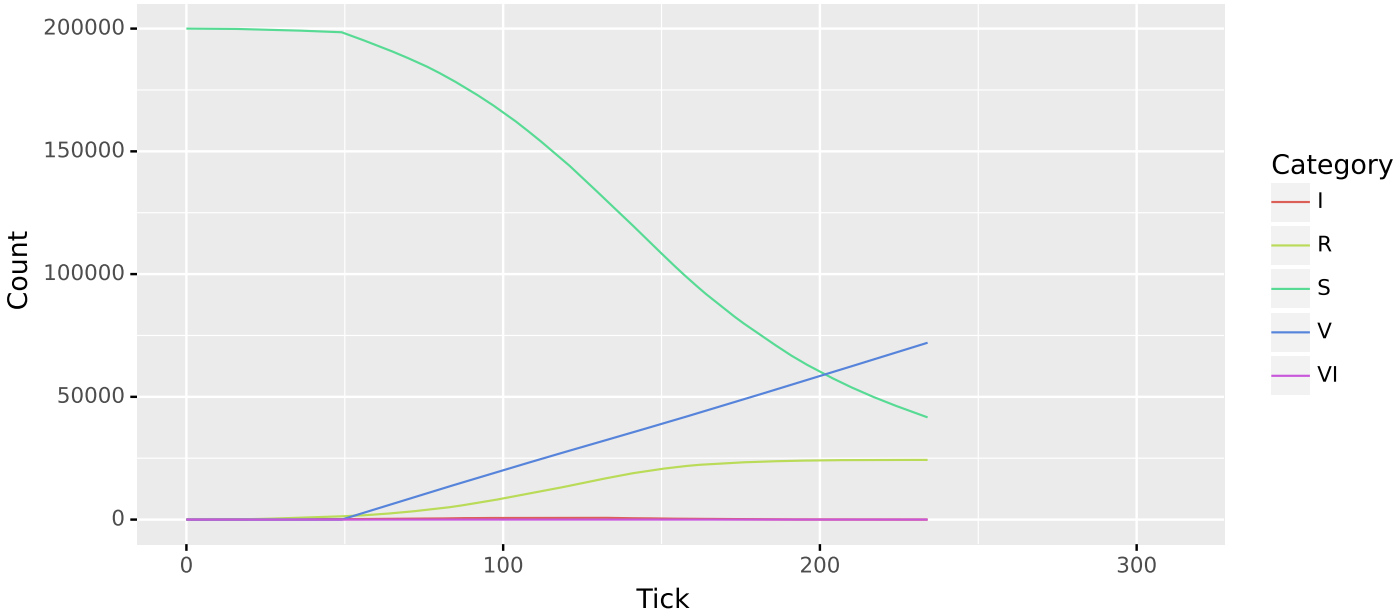


Figure 5.1.2: A plot showing the results of a SIRV model where vaccinations decrease transmission and infection by 50%.

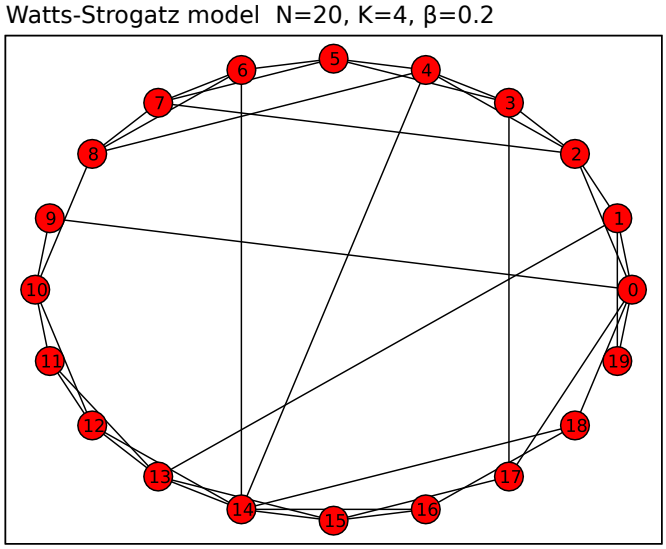


Figure 5.1.3: An example of a Watts-Strogatz graph, WS(20, 4, 0.2). By Arpad Horvath - This W3C-unspecified plot was created with Matplotlib., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4121767.

complex, and some individuals are at risk of dying from infection (so we might introduce a new category, but we do not go into such detail here). People who are particularly susceptible may be given priority to vaccination; however, we can also pick a vaccination strategy based on the topological features of G.

One such (naïve) method for choosing candidates for vaccinates may be a simple node property that measures how likely the node is to infect other people; for example, the degree of the node. To make a more involved choice, we may move to understand the underlying structure of a contact graph.

The common choice of G to model contact in a population is the Watts-Strogatz (WS) model [20].

Definition 5.1.3 (Watts-Strogatz model)**.** The *Watts-Strogatz model* is a random graph generation model. Given the desired numbers of nodes $n \in \mathbb{N}$, the mean degree $k \in 2\mathbb{N}$, and parameter $\beta \in [0, 1]$, we construct a Watts-Strogatz graph $\mathrm{WS}(n, k, \beta)$ as follows.

(i) Construct a regular ring graph: vertices $\{0, \ldots, n-1\}$ and edges of the form $\{v, v+1 \bmod n\}$ for all $n \in \{0, \ldots, n-1\}$.

(ii) For each vertex, connect it to the $k/2$ vertices on each size; that is, include edges $\{v, v+2 \bmod n\}, \{v, v+3 \bmod n\}, \ldots, \{v, v+k/2 \bmod n\}$.

(iii) For each vertex v, take every edge containing v and rewire it with probability β to another vertex picked uniformly at random (avoiding duplicate edges and self-loops).

An example of a WS graph is shown in Figure 5.1.3.

k may be interpreted as the amount of contact each person has in a population; however, it is not unreasonable to assume that people have differing levels of contact. Thus an obvious extension to the WS model is the variable degree Watts-Strogatz (VDWS) model, in which k is taken from a discrete zero-truncated exponential distribution for each vertex (we call this the *local degree* of the vertex, we may multiply this by two to ensure an even parameter) with the

parameter being a relabelled k. We denote this model by VDWS(n, k, β), and this is the model we will move forward with.

We have two types of nodes that may be good candidates for vaccination:

(i) nodes with a high number of rewires; and

(ii) nodes with a high local degree.

This graph is a *model* of what we may expect in the population, so we do not have access to properties such as *local degree* and *number of rewires*.

Let $G \sim \text{VDWS}(n, k, \beta)$. We first comment that this graph is (almost always) connected. If we examine the clique complex, $\text{Cl}(G)$, we see that the local neighbours will be filled in with 2-simplices (or higher dimensional simplices). Thus, these neighbourhoods will not contribute 1-holes in the homology. Instead, 1-holes are induced by edges that were rewired: between the endpoints of a rewired edge, there is likely a cycle going around the ring of the graph and the rewired edge. We propose that dropping out vertices (as we did in the previous section) and examining the change of $\beta_1(\text{Cl}(G))$ may give a good indication of nodes that connect otherwise distance communities, and so may be good candidates for vaccination.

Alternatively, we may further examine the power filtration of G:

$$\text{Pow}(G) : \text{Cl}(G^0) \subset \text{Cl}(G^1) \subset \text{Cl}(G^2) \subset \ldots \subset \text{Cl}(G^{\text{diam}\, G})$$

and pick nodes with 1-holes that persist as we move through the filtration, giving us a cycle through the graph. The path which persists the longest is likely a path around the ring of the graph, so we can discard this. The remaining paths with high persistence likely make use of rewired edges, but we need to separate the rewired edges from the edges on the ring of the graph. To do this, we may use the Jaccard index. The edges whose endpoints have the highest Jaccard index are most likely the rewired edges, thus giving us good candidates for vaccination. For a given (suspected) rewired edge, it is probably beneficial to only vaccinate one. To do this, we may compare them using a simple measure, such as vertex degree.

No experiments on the performance of this technique has been conducted here, though we pose it as an area of further study. As noted in the last section, such a new technique should not be the sole selection criteria for a vaccination strategy. Persistent homology should be used in conjunction with other well-tested methods.

There does not seem to be any literature present that applies this technique in persistent homology to compartmental models of infectious disease. As such, we present this as novel work.

5.2 Cyclomatic complexity

Definition 5.2.1 (Graph homology)**.** Let $G = (V, E)$ be a directed graph. The corresponding integral simplicial chain complex is $C(G; \mathbb{Z}) = (C_*, \partial_*)$ where

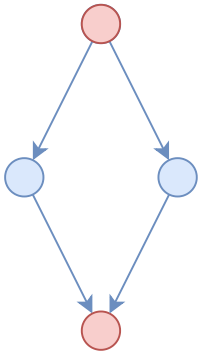
$$C_0 = \mathbb{Z}\langle V \rangle, \qquad C_1 = \mathbb{Z}\langle E \rangle$$

and $C_i = 0$ for all $i \geq 2$ and the only non-trivial boundary map $\partial : C_1 \to C_0$ is defined by

$$\partial(v, w) = w - v,$$

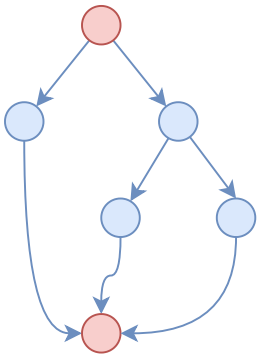
and extending linearly. The *nth homology group of* G is simply the nth homology group of $C(G; \mathbb{Z})$; that is,

$$H_n(G) = H_n(C(G)) = \begin{cases} \text{coker}\, \partial & n = 0, \\ \ker \partial & n = 1, \\ 0 & \text{otherwise.} \end{cases}$$



(a) The control flow graph of an if-else.

(b) The control flow graph of a while loop.



(c) The control flow graph of an if-elseif-else.

Figure 5.2.1: Control-flow graphs of some simple programs.

Proposition 5.2.2. *Let $G = (V, E)$ be a graph and C be the set of connected components of G. Then the graph homology of G is given by*

$$H_n(G) = \begin{cases} \mathbb{Z}^{|C|} & n = 0, \\ \mathbb{Z}^{|E|+|C|-|V|} & n = 1, \\ 0 & \textit{otherwise.} \end{cases}$$

That is, the rank of the first homology is the number of connected components in G and the rank of the second homology is the number of linearly independent cycles in G (the concept of linearly independent paths is later formalised).

Proof. Recall that $H_0(G) = \ker \partial_0 / \operatorname{im} \partial_1$. Now we consider a component of G. We see that any vertex can be obtained by another vertex by adding elements of $\operatorname{im} \partial_1$. Thus all vertices belong to the same equivalence class. It is clear that we cannot do the same for two vertices within different components, thus $H_0(G) \cong \mathbb{Z}^{|C|}$. Now we let $T \subset E'$ be a spanning tree of a component $G' = (V', E')$. We observe that if we add another edge (not in T) to T, we get a cycle. Thus each edge in $E' \setminus T$ corresponds to a cycle in G'. $|E' \setminus T| = |E'| - (|V'| - 1)$. Considering every component, we get $H_1(G) \cong \mathbb{Z}^{|E|-|V|+|C|}$. The rest of the homology groups is trivial, given that the boundary map is trivial. □

Definition 5.2.3 (Control-flow graph)**.** A *control-flow graph* of a program is a directed graph $G = (V, E)$ where

(i) each vertex $v \in V$ corresponds to a block (piece of code) without any jumps; and

(ii) each edge $(v, w) \in E$ corresponds to a jump in the control flow (that is, the order in which statements are executed).

We designate two special types vertices:

(i) v_{en}: the *entry block*, in which control enters the graph; and

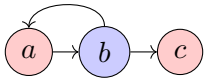
(ii) v_{ex}: the *exit block*, in which control leaves the graph.

Here, we will only be considering graphs in which there exists a path from the entry block to every other block. Figure 5.2.1 shows examples of control-flow graphs for some simple programs.

Definition 5.2.4 (Linearly independent paths)**.** Let $G = (V, E)$ be a directed graph and Q a set of paths in G. Q is said to be *linearly independent* if, for each $p \in Q$, there is an edge in p that is not included in any other path $q \in Q \setminus \{p\}$. Q is said to be a *basis* of G if it is maximal.

It is simple exercise to prove that all bases of a graph G have the same cardinality.

Example 5.2.5. Consider the graph drawn below.



One example of a set of linearly independent paths (from a to c) is

$$Q = \{(a, b, c), (a, b, a, b, c)\}$$

and indeed this set forms a basis. But the set

$$Q' = \{(a, b, a, b, c), (a, b, a, b, a, b, c)\}$$

is not linearly independent as every edge in the second path is present in the first path.

Definition 5.2.6 (Cyclomatic complexity)**.** Consider a program P and let G be its control-flow graph. The *cyclomatic complexity* of P, denoted $M(P)$, is the number of linearly independent paths from the entry block to the exit block in G.

Example 5.2.7.

(i) For the control-flow graphs shown in Figure 5.2.1a and Figure 5.2.1b, $M(P) = 2$.

(ii) For the control-flow graph shown in Figure 5.2.1c, $M(P) = 3$.

Lemma 5.2.8. *Let P be a program, G its control-flow graph, Q be a basis of paths between v_{en} and v_{ex} in G, and C be a basis of cycles based at v_{en} in $G/\{v_{en}, v_{ex}\}$. Then*

$$|Q| = |C|.$$

Proof. Let $p = (v_{\text{en}}, v_1, \ldots, v_n, v_{\text{ex}}) \in Q$. Then the induced path (from G to $G/\{v_{\text{en}}, v_{\text{ex}}\}$) $\overline{p}$ is clearly a cycle, and we claim that it is linearly independent. Assume otherwise, then for all $\overline{e} \in \overline{p}$ there is a path $\overline{p}_{\overline{e}} \in \overline{Q}$ such that $\overline{e} \in \overline{p}_{\overline{e}}$. As G is simple, the quotient map forms a bijection between the edges of G and the edges of $G/\{v_{\text{en}}, v_{\text{ex}}\}$ (and thus a bijection between the paths). Thus for all $e \in p$, there is a path $p_e \in Q$ such that $e \in p_e$ and therefore, Q is not linearly independent; a contradiction. We also claim that $\overline{Q}$ is maximal. Indeed, let $\overline{c}$ be a cycle in $G/\{v_{\text{en}}, v_{\text{ex}}\}$ such that $\overline{Q} \cup \{\overline{c}\}$ is linearly independent. But then $Q \cup \{c\}$ is linearly independent, and so Q is not maximal; a contradiction. □

Figure 5.2.2 shows the control flow graph G of an if-elseif-else program, alongside the quotient graph $G/\{v_{\text{en}}, v_{\text{ex}}\}$.

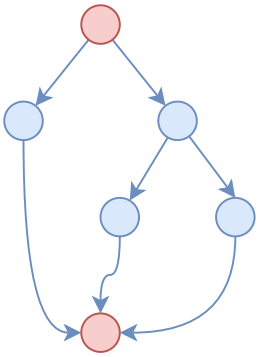
The following is a direct result from Proposition 5.2.2 and Lemma 5.2.8.

Corollary 5.2.9. *Let P be a program and $G = (V, E)$ be its control-flow graph. Then*

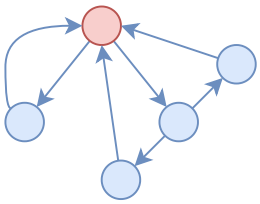
$$M(P) = \operatorname{rank} H_1(G, \{v_{en}, v_{ex}\}) = |E| + 2|C| - |V|$$

where C is the connected components of G.

Cyclomatic complexity is a commonly used metric for the testability of code, and our persistent homology algorithm presents an efficient method for its computation.



(a) The control flow graph G of an if-elseif-else program.



(b) The quotient graph $G/\{v_{\text{en}}, v_{\text{ex}}\}$.

Figure 5.2.2: Control-flow graphs of some simple programs.

Chapter 6

Conclusions and further work

In Chapters 2 and 3, we built up the required theory to understand the computation of persistent homology. We then moved to understand the problems and corresponding algorithms for both: the construction of Vietoris-Rips filtrations (Section 4.1; and the computation of the persistent barcodes (Section 4.2), presenting empirical evidence to the real-world run time of such algorithms. We first outline the main findings from our experiments.

We first introduced the construction of Vietoris-Rips filtrations, where we concluded that the methods provided by sklearn [16] gave exceptionally increased performance over the brute-force algorithm. For the expansion method, we found that the incremental algorithm outperformed the similar inductive algorithm, which both greatly improved on the brute-force algorithm.

We then moved to the persistent homology algorithm, where we introduced the standard algorithm [2] and posed three *speed-ups* that allowed, what is effectively the same algorithm at the core, to run significantly faster. We first showed the benefit of *sparse matrix representation*, allowing us to avoid storing redundant elements in sparse matrices. We then moved to show a *clearing* technique [17] that allowed significantly less column operations to be performed. We then concluded on the use of the coboundary matrix for reduction [18], which provided more opportunities in the use of the clearing technique.

We then moved onto potential applications of persistent homology in Chapter 5, showing how our framework can be applicable to many different problems. Note that one of the main challenges found in persistent homology is the interpretation of persistent barcodes. Uses of persistent homology on point-cloud data is vast [1], thus we focussed on more novel techniques. We showed how Betti numbers can give a measure on the complexity of a graph, both in the context of protein-protein interaction graphs and for control-flow graphs. We additionally posed the use of power filtration of a contact graph to determine vaccination strategies to reduce the spread of infectious disease.

We conclude our work here, not without a highlight of potential topics for further research, as well as some further study for engrossed readers.

The following is some topics posed as further work.

HNSW Hierarchical navigable small-world networks (HNSW) [14] are positioned as the current state-of-the-art algorithm, for ε-NNs, but application of this technique to the skeleton method in the Vietoris-Rips construction step in the pipeline of persistent homology is yet to be seen.

Theoretical complexity bounds for expansion methods Our understanding on the inherent *difficulty* of the expansion methods seem much more scarce, and more research

into the underlying complexity theory would be beneficial.

Persistent homology with coefficients in a non-divisible group Substantial research has been conducted studying efficient methods of calculating the Smith normal form of an integer matrix [21], but applying such a method to persistent homology as well as existing speed-ups we have outlined is yet to be seen. We regret the omission of such content; however, the required context delves deep into number theory which we could respectfully fit here. We do note a need for a rework of theoretical foundations needed for such an undertaking, as our succinct decomposition of persistence modules relies on coefficients in a field. Additional considerations must also be taken by the use; that is, does torsional information induced by such coefficients give any useful insight.

Alternative sparse matrix representations In our exposition, we outlined one of many techniques for storing our boundary matrix. Many alternatives exist, and the use of different structures may lead to input-dependent benefits. We pose not only the use of sorted tree-like data structures, but also the use of more unique data structures such as *simplex trees*, introduced by [22].

Persistent homology as an analogue to cyclomatic complexity Huntsman [23] posed the use of *path homology* as a stronger analogue of cyclomatic complexity. In a similar vein, can we derive useful measures of program complexity from persistent homology (such as using the power filtration)? Here, we have simply posed the persistent homology algorithm as a fast method of computing cyclomatic complexity.

Relationship between persistent homology and other complexity measures on graphs A related proposal to the previous, are there any relationships between graph (or complex) complexity measures derived from persistent homology and other, more established graph (or complex) complexity measures (such as clustering coefficients, assortativity, etc.).

Novel filtrations as inputs to the persistent homology algorithms The use of the persistent homology algorithms lies, almost exclusively, in the choice of filtration we build from our data. For example, the we have shown the *power filtration* to give more insight into topological features of graphs, without needing any more information [19]. To allow persistent homology to find applications in more fields, rigorous investigation is needed on complexes that may been constructed that allow us to interpret persistent homology in a useful way.

The following is some further material, for interested readers.

Spectral sequences *Spectral sequences* and *persistent homology* are both tools within algebraic topology which are defined on a filtration, and can be applied to the study of the topological features of the underlying spaces. They are both deeply related, but spectral sequences require a much more rigorous familiarity with algebraic topology [24]. Note that problems we outlined with non-divisible groups, such as $\mathbb{Z}$, are still present.

Stability We introduced persistent homology as a tool robust against noise. The *bottleneck distance* is a natural distance function defined on the space of persistent diagrams, and the use of this function can be used to prove the robustness of persistent homology [25].

Cohomology and duality For those interested in more theoretical study, a substantial amount of research has been conducted into the relations between homology and cohomology, providing particularly succinct results [26, 27].

Bibliography

[1] Nina Otter et al. "A roadmap for the computation of persistent homology". In: *EPJ Data Science* 6 (2017), pp. 1–38.

[2] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. "Topological persistence and simplification". In: *Proceedings 41st annual symposium on foundations of computer science*. IEEE. 2000, pp. 454–463.

[3] Ulrich Bauer. "Ripser: efficient computation of Vietoris–Rips persistence barcodes". In: *Journal of Applied and Computational Topology* 5.3 (2021), pp. 391–423.

[4] Primoz Skraba and Mikael Vejdemo-Johansson. "Persistence modules: algebra and algorithms". In: *arXiv preprint arXiv:1302.2015* (2013).

[5] Afra Zomorodian and Gunnar Carlsson. "Computing persistent homology". In: *Discrete & Computational Geometry* 33.2 (2005), pp. 249–274.

[6] David Eisenbud. *Commutative Algebra*. Springer New York, 1995. DOI: 10.1007/978-1-4612-5350-1.

[7] Jonathan Rosenberg and Claude Schochet. "The Künneth theorem and the universal coefficient theorem for Kasparov's generalized K-functor". In: *Duke Mathematical Journal* 55.2 (1987), pp. 431–474.

[8] Massimo Ferri, Dott Mattia G Bergomi, and Lorenzo Zu. "Simplicial complexes from graphs towards graph persistence". In: *arXiv preprint arXiv:1805.10716* (2018).

[9] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[10] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[11] Afra Zomorodian. "Fast construction of the Vietoris-Rips complex". In: *Computers & Graphics* 34.3 (2010), pp. 263–271.

[12] Sunil Arya et al. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions". In: *Journal of the ACM (JACM)* 45.6 (1998), pp. 891–923.

[13] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms". In: *Information Systems* 87 (2020), p. 101374.

[14] Yu A Malkov and Dmitry A Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs". In: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018), pp. 824–836.

[15] Dominique de Caen. "An upper bound on the sum of squares of degrees in a graph". In: *Discrete Mathematics* 185.1-3 (1998), pp. 245–248.

[16] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[17] Chao Chen and Michael Kerber. "Persistent homology computation with a twist". In: *Proceedings 27th European workshop on computational geometry*. Vol. 11. 2011, pp. 197–200.

[18] Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. "Dualities in persistent (co) homology". In: *Inverse Problems* 27.12 (2011), p. 124003.

[19] Sebastian Benzekry et al. “Design principles for cancer therapy guided by changes in complexity of protein-protein interaction networks”. In: *Biology direct* 10.1 (2015), pp. 1–14.

[20] Duncan J Watts and Steven H Strogatz. “Collective dynamics of ‘small-world’networks”. In: *nature* 393.6684 (1998), pp. 440–442.

[21] George Havas, Bohdan S Majewski, and Keith R Matthews. “Extended gcd and Hermite normal form algorithms via lattice basis reduction”. In: *Experimental Mathematics* 7.2 (1998), pp. 125–136.

[22] Jean-Daniel Boissonnat and Clément Maria. “The simplex tree: An efficient data structure for general simplicial complexes”. In: *Algorithmica* 70.3 (2014), pp. 406–427.

[23] Steve Huntsman. “Path homology as a stronger analogue of cyclomatic complexity”. In: *arXiv preprint arXiv:2003.00944* (2020).

[24] Ana Romero et al. “Spectral sequences for computing persistent homology of digital images”. In: *Málaga* (2013).

[25] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. “Stability of persistence diagrams”. In: *Discrete & computational geometry* 37.1 (2007), pp. 103–120.

[26] Charles Richard Francis Maunder. *Algebraic topology*. Courier Corporation, 1996.

[27] Allen Hatcher. *Algebraic topology*. Cambridge: Cambridge Univ. Press, 2000.

Appendices

Appendix A

Some foundations of topology

First, we cover some elementary group theory.

Definition A.0.1 (Free abelian group)**.** A *free abelian group* on generators $\{e_\alpha\}_{\alpha\in\mathcal{A}}$ is the group of formal sums

$$\mathbb{Z}\langle\{e_\alpha\}_{\alpha\in\mathcal{A}}\rangle = \left\{\sum_{\alpha\in\mathcal{A}} n_\alpha e_\alpha : n_\alpha \in \mathbb{Z},\ \text{finitely many } n_\alpha \neq 0\right\}.$$

If $\mathcal{A}$ has size k, then $\mathbb{Z}\langle\{e_\alpha\}_{\alpha\in\mathcal{A}}\rangle \cong \mathbb{Z}^k$.

In general, we have a notion of direct product of groups; however, for free abelian groups we have the restriction that formal sums must have finitely many non-zero terms. Thus the direct product of infinite families of free abelian groups may not be free abelian. From this, we introduce the notion of *direct sum*.

Definition A.0.2 (Direct sum)**.** Let $\{A_i\}_{i\in I}$ be a family of abelian groups. Then the *direct sum* is

$$\bigoplus_{i\in I} A_i = \left\{(a_i)_{i\in I} \in \prod_{i\in I} A_i : \text{finitely many } a_i \neq 0\right\}.$$

We now recall some elementary topology.

Definition A.0.3 (Topological space)**.** A *topological space* (or just *space*) is an ordered pair (X,τ) where X is a set and $\tau \subset \mathcal{P}(X)$ such that

(i) the empty set and X both belong to τ;

(ii) any arbitrary union of members of τ belongs to τ; and

(iii) any finite intersection of members of τ belongs to τ.

τ is called a *topology* on X, the elements of τ are called the *open sets* of X.

Note we may just refer to a space (X,τ) as X. We may also define a topology in terms of a basis: a basis $\mathcal{B}$ of a space X is a collection of open sets such that every open set is the union of elements from $\mathcal{B}$.

Definition A.0.4 (Subspace topology)**.** Given a topological space (X,τ) and a subspace $S \subset X$, the *subspace topology* on S is defined by

$$\tau_S = \{S \cap U : U \in \tau\}.$$

Example A.0.5 (Standard topology on $\mathbb{R}^n$)**.** A topology may be derived from any metric, and when we refer to the *standard topology* in $\mathbb{R}^n$ we will assume it is the one induced from the Euclidean metric: the distance from $\boldsymbol{x}$ to $\boldsymbol{y}$ is given by

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

The topology induced by this is generated by the basis of open balls.

Example A.0.6 (Standard subsets of $\mathbb{R}^n$)**.** When considering subsets of $\mathbb{R}^n$, we assume the subspace topology of the standard topology.

- (Closed unit interval) $I = \{x \in \mathbb{R} : x \in [0, 1]\}$.
- (Closed n-disc) $D^n = \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| \leq 1\}$.
- (Open n-disc) $E^n = \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| < 1\}$.
- (n-sphere) $S^n = \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| = 1\}$.

Definition A.0.7 (Product topology)**.** Let X and Y be spaces. The *cartesian product* of X and Y is

$$X \times Y = \{(x, y) : x \in X, y \in Y\}$$

and may be given the *product topology* generated by the basis

$$\mathcal{B}_{X \times Y} = \{U \times V : U \text{ is open in } X \text{ and } V \text{ is open in } Y\}.$$

Example A.0.8 (Product constructions)**.** The product construction gives us some more basic spaces.

- (Torus) $\mathbb{T} = S^1 \times S^1 = \{(x, y) : x, y \in S^1\}$.
- (Cylinder) $S^1 \times I = \{(x, y) : x \in S^1, y \in [0, 1]\}$.

Let us define our first morphism on spaces.

Definition A.0.9 (Continuous)**.** Let X and Y be spaces. A function $f : X \to Y$ is called *continuous* if for every open subset $U \subset Y$, $f^{-1}(U) \subset X$ is also open.

We may refer to a continuous function as a *map*.

Definition A.0.10 (Homeomorphism)**.** Let X and Y be spaces. A map $h : X \to Y$ is called a *homeomorphism* if h is bijective and h^{-1} is continuous.

If a homoemorphism exists between two spaces, they are said to be homeomorphic and we write $X \cong Y$, and indeed being homeomorphic is an equivalence relation on the set of spaces. If X and Y are homeomorphic, we may imagine that we can continuously stretch and bend X into Y (if we think of the spaces as geometric objects).

Definition A.0.11 (Quotient topology)**.** Let X be a space and $\sim$ be some equivalence relation on X. Consider the canonical projection map $\pi : X \to X/\!\sim$, $x \mapsto [x]$ (this map is surjective, so it has an inverse). We equip the set $X/\!\sim$ with the topology defined by: $U \subset X/\!\sim$ is open if and only if $\pi^{-1}(U)$ is open in X, we call this the *quotient topology*.

We can also build some of our elementary spaces using a quotient construction.

Example A.0.12. We consider S^1. We can build this space (up to homeomorphisms) by gluing the two ends of an interval. That is, $S^1 \cong I/\!\sim$ where $0 \sim 1$. Intuitively, you would be inclined to believe; however, for this simple example we will be thorough. To be clear,

$S^1 = \{(x, y) \in \mathbb{R}^2 : \|x\| = 1\}$ and $I = \{x \in \mathbb{R} : x \in [0, 1]\}$. We note that we can parametrise S^1 as $S^1 = \{(\cos t, \sin t) \in \mathbb{R}^2 : t \in [0, 2\pi)\}$. From this, we define the homeomorphism $h : S^1 \to I/\sim$ by $(\cos t, \sin t) \mapsto \frac{1}{2\pi}t$. Checking that h is continuous, h is a bijection, and h^{-1} is continuous is easy to do.

Example A.0.13. Consider $\mathbb{T}$. We construct the torus out of the square $I \times I$, identifying the top and bottom points as well as the left and right points. That is, $\mathbb{T} = (I \times I)/\sim$ where $(x, 0) \sim (x, 1)$ and $(0, y) \sim (1, y)$ for all $x, y \in I$. This is indeed homeomorphic to the product construction.

Example A.0.14. We can use this quotient construction to create some variations of simple spaces. For example, we build the Klein bottle $\mathbb{K}$ much like the torus, but with one of the identifications running backwards. That is, $\mathbb{K} = (I \times I)/\sim$ where $(x, 0) \sim (x, 1)$ and $(0, y) \sim (1, 1 - y)$ for all $x, y \in I$.

We now define another equivalence relation on maps.

Definition A.0.15 (Homotopic)**.** Two maps $f, g : X \to Y$ are *homotopic*, written $f \simeq g$, if there is a *homotopy* H, a map $H : X \times I \to Y$ with

$$H(x, 0) = f(x),$$
$$H(x, 1) = g(x)$$

for all $x \in X$.

Establishing this as an equivalence relation is elementary. Two maps being homotopic can be thought of as being able to continuously deform f into g. This relation also induces another relation on spaces.

Definition A.0.16 (Homotopy equivalent)**.** Two spaces X and Y are *homotopy equivalent*, written $X \simeq Y$ if there are maps $\alpha : X \to Y$ and $\beta : Y \to X$ such that $\alpha \circ \beta \simeq \mathrm{id}_Y$ and $\beta \circ \alpha \simeq \mathrm{id}_X$.

Example A.0.17. We denote pt for the one point space. We will show that $D^2 \simeq \mathrm{pt}$. To show this we define $\alpha : D^2 \to \mathrm{pt}$ the only way we can, and $\beta : \mathrm{pt} \to D^2$ by $\beta(\mathrm{pt}) = \mathbf{0}$. We have $\alpha \circ \beta = \mathrm{id}_{\mathrm{pt}} \simeq \mathrm{id}_{\mathrm{pt}}$ and $\beta \circ \alpha = \mathbf{0} \simeq \mathrm{id}_{D^2}$ by the homotopy $H(\boldsymbol{x}, t) = t\boldsymbol{x}$.

Definition A.0.18 (Contractible)**.** We say a space X is *contractible* if it homotopy equivalent to pt.

Appendix B

An more in-depth look of the theory of computation

B.1 Computational problems

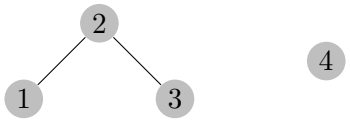
B.1.1 Decision problems

Within this section we aim to formalise the notion of a *problem*; we move towards a rigorous description of a computational problem. This will lay the foundation for our framework of computation of choice: *Turing machines*.

We may look at a *computation problem* as a problem that we may expect a computer to be able to solve, though this is not particularly specific or helpful. Instead, we view a computational problem as a set of *instances* with a (potentially empty) set of *solutions*.

Example B.1.1 (Primality)**.** Consider the following problem: given $n \in \mathbb{N}$, determine whether n is prime. In this scenario, an instance is any $n \in \mathbb{N}$, and our *solutions* (or *yes-instances*) are the prime $n \in \mathbb{N}$.

Example B.1.2 (Reachability)**.** We will come to realise that many computational problems are concerned with graphs. The most basic of which is the following: suppose we have a graph G and $v, w \in V(G)$, is there a path from v to w? We call this problem Reachability. For example an example, let G be a graph with the following geometric realisation.



If $v, w \in \{1, 2, 3\}$, then Reachability would have the answer *yes*, where if either $v = 4$ or $w = 4$ (but not both!), then the answer would be *no*. In this case, our *instance* is a graph and two of its vertices. An instance is a solution only if a path exists between the two vertices.

We now formalise our earlier discussion, focussing first on *decision problems*.

Definition B.1.3 (Decision problem)**.** A *decision problem* is a two-tuple (I, Y) where I is a set of *instances* and $Y \subset I$ is a set of *yes-instances*.

We now revisit our two initial examples,

$$\begin{aligned} \textsc{Primality} &= (\mathbb{N}, \{n \in \mathbb{N} : n \text{ is prime}\}) \\ \textsc{Reachability} &= (I, Y) \end{aligned}$$

where $I = \{(G, u, v) : u, v \in V(G)\}$ and $Y \subset I$ such that $(G, u, v) \in Y$ if and only if there exists a path from u to v in the graph G.

B.1.2 Encoding

We start with some notation and terminology.

Definition B.1.4 (Words over an alphabet)**.** An *alphabet* Σ is some non-empty set of symbols. The elements of Σ are called *letters* and a finite sequence of letters is called a *word* or a *string*.

Let $\Sigma = \{v_i\}_{i \in I}$ be some alphabet. We denote a word $(v_1, v_2, \ldots)$, although we may use $v_1 v_2 \ldots v_n$. We indicate the set of all strings of length $n \in \mathbb{N}$ as Σ^n, and the set of all finite strings as Σ^*. We may use Σ^ω to denote the set of all infinite sequences of Σ. We point out the existence of the empty string, denoted ε, for any alphabet.

Example B.1.5. A common alphabet we will see is $\{0, 1\}$, the *binary alphabet*. We see that $010, 1100, 1111111 \in \{0, 1\}^*$.

Definition B.1.6 (Formal language)**.** A *formal language* (or just *language*) $\mathcal{L}$ over an alphabet Σ is some subset of Σ^*.

We now look to represent computational problems; encoding them so we can study them as objects. In general, we aim to study the complexity of computing a function $f : \{0, 1\}^n \to \{0, 1\}^m$ (input and output are finite sequences of zeroes and ones). Note that we can represent any general object as a string of bits, with a number of different encodings to choose from (for example, we may choose to encode a graph as an adjancency matrix or alternatively an adjancency list). We avoid the details of such an encoding, and denote some unspecified canonical binary representation of an object x as $\langle x \rangle$ (depending on context, we may simply use x to denote both the object and its encoding). For a sequence $(x_1, \ldots, x_n)$ we may denote its binary representation as $\langle x_1, \ldots, x_n \rangle$. For example, we may write $\langle G, u, v \rangle$ for the binary representation of an instance of Reachability.

Now, we consider a Boolean function $f : \{0, 1\}^* \to \{0, 1\}$ and consider the corresponding language $\mathcal{L}_f = \{x : f(x) = 1\} \subset \{0, 1\}^* = \Sigma$. We observe that $(\Sigma, \mathcal{L}_f)$ is a decision problem, and we typically interchange the terms *language* and *decision problems*. When referring to a problem as a language, we may drop alphabet from the tuple; for example, we write

$$\textsc{Primality} = \{\langle n \rangle : n \text{ is prime}\}.$$

Example B.1.7. An *independent set* is a subset of the vertices of a graph such that no two vertices are adjacent. Thus we define

$$\begin{aligned}\textsc{IndSet} = \{\langle G, k \rangle : &\text{ there is } S \subset V(G) \text{ such that} \\ &|S| \geq k \text{ and for all } u, v \in S,\ uv \notin E(G)\}\end{aligned}$$

which corresponds to the problem of deciding whether a graph has an independent set of size k.

From now on decision problems will be formatted as an *instance* and a *question*. We give the above example to illustrate.

Problem B.1.8 (IndSet).
Instance: let G be a graph and $k \in \mathbb{N}$.
Question: does G have an independent set of size k?

The reason for this is to benefit from the natural language description, but also stay close towards the formal language description (which becomes relevant when looking at our model of computation). Building the language from this format using set builder notation is clear.

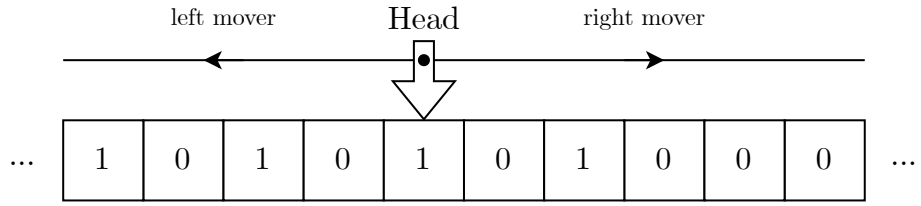


Figure B.2.1: Schematic of a Turing machine.

B.2 Turing machines

Informally, an *algorithm* is a collection of simple instructions for carrying out some task. Algorithms ahve been intuitively understood for thousands of years (for example, Euclid's algorithm), but it is not until the last couple centuries that we have formalised the notion.

Turing [9] defined the now ubiquitous sequential model of computation in 1936, with the primary motivation behind his construction to capture the notion of *computability*; that is, given a problem, does there exist an *algorithm* that can solve it?

First, a model of computation is an abstract framework that we use to study computation; that is, how an output of a mathematical function may be computed given an input. *Turing machines* are a model of computation, it is a primitive machine that is (by the Church-Turing thesis) able to run any algorithm. Turing machines are not of practical interest; they are designed to be simple to allow us to study properties of computation.

A Turing machine has infinite tape (memory). There is a finite-state *program* that controls a tape head. The head can read, write, and move around in both directions on the tape. Figure 3.1.1 gives a visual representation of a Turing machine.

Definition B.2.1 (Turing machine)**.** A Turing machine is a 7-tuple

$$(Q, \Sigma, \Pi, \delta, q_0, q_a, q_r)$$

where

- Q is a finite non-empty set of states;
- Σ is the input alphabet not containing the special blank symbol $\sqcup$;
- Π is the tape alphabet satisfying $\Sigma \subset \Pi$ and $\sqcup \in \Pi$;
- $\delta : Q \times \Pi \to Q \times \Pi \times \{L, R\}$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $q_a \in Q$ is the accept state; and
- $q_r \in Q$ is the reject state ($q_a \neq q_r$).

We now describe computation on a Turing machine. The tape content is unbounded but always finite: the first leftmost blank symbol marks the end of the tape. Let M be a Turing machine as above and let $w \in \Sigma^*$ be an input string for the Turing machine. A *configuration* of M consists of three items: the current state $q \in Q$, the tape content $x \in \Pi^*$, and the head location $k \in \mathbb{Z}$. Let $C_1 = (q_1, (x_1, \ldots, x_n), k_1)$ and $C_2 = (q_2, (y_1, \ldots, y_n), k_2)$ be two configurations M. We say that C_1 *yields* C_2 if M can go from C_1 to C_2 in a single step; that is, either

- $\delta(q_1, x_{k_1}) = (q_2, y_{k_1}, L)$, $x_i = y_i$ for all $i \neq k_1$, and $k_2 = k_1 - 1$; or
- $\delta(q_1, x_{k_1}) = (q_2, y_{k_1}, R)$, $x_i = y_i$ for all $i \neq k_1$, and $k_2 = k_1 + 1$.

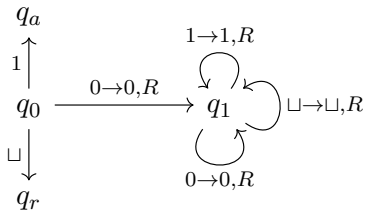
The *start configuration* on input w consists of the start state q_0, w as the tape content, and the head location 1 (at the leftmost position of the tape). An *accepting* configuration is a configuration whose state is q_a, and similarly a *rejecting* configuration has state q_r. Both accepting and rejecting configurations are halting configuration; once these configuration is reached, M stops running.

A Turing machine M *accepts* an input w if there is a finite seuqence of configuration $(C_1, \ldots, C_k)$ such that C_1 is the start configuration of M on w; C_i yields C_{i+1} for all $i \in \{1, \ldots, k-1\}$; and C_k is an accepting configuration. The set of strings accepted by M is called the *language of* M; denoted $L(M)$.

We now look at a worked example to see how a Turing machine may operate on a given input.

Example B.2.2. Consider the Turing machine M defined as

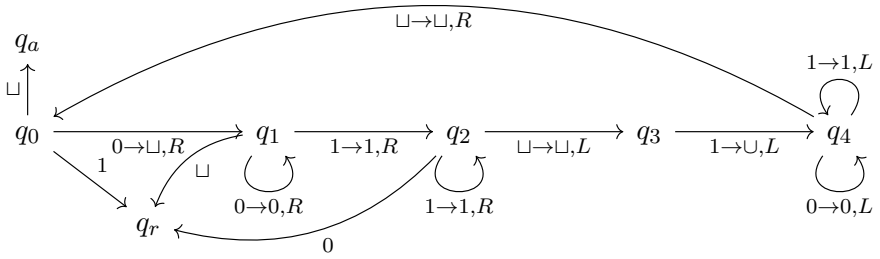
- $Q = \{q_0, q_1, q_r, q_a\}$;
- $\Sigma = \{0, 1\}$;
- $\Pi = \{0, 1, \sqcup\}$; and
- δ as decribed in the state diagram below.



Suppose we have input w and we run M on w. We start in state q_0. M then reads the first tape cell, if it reads 1 it moves to the accept state q_a and halts. If it reads $\sqcup$ it moves to the reject state q_r and halts. Finally, if it reads a 0 it leaves the cell unchange (it writes a 0 over it), it changes state to q_1, and moves the head to the right one. Within q_1, no matter what is read on the head, the machine will leave it unchanged and move to the next cell. We conclude that this Turing machine will reject an empty input, accept an input that starts with 1, and will not halt on any other input. We may say that M recognises $\mathcal{L} = \{w : w \text{ starts with a } 1\}$, or just $L(M) = \mathcal{L}$.

Example B.2.3. Consider the Turing machine M defined as

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_a, q_r\}$;
- $\Sigma = \{0, 1\}$;
- $\Pi = \{0, 1, \sqcup\}$; and
- δ as decribed in the state diagram below.



The general strategy here is: if the string is empty, we accept. Otherwise, we will erase a 0 from the start of the string and a 1 from the end of the string and then return to the beginning of the string and repeat. If at any point we cannot do this, we reject the string. We see that this machine will accept if a string has a finite number of 0s followed by the same number of 1s, we denote this language $\mathcal{L} = \{0^n 1^n : n \in \mathbb{Z}_{\geq 0}\}$ and we say that M *recognises* $\mathcal{L}$ as if $x \in \mathcal{L}$, M accepts x. We also see that M rejects if $x \in \Sigma^* \setminus \mathcal{L}$, we say that M decides $\mathcal{L}$ (that is, accepts when the input is in the language and rejects if it isn't).

We can view Turing machines as a partial Boolean functions $f : \{0,1\}^* \to \{0,1\}$, so we may treat it as one. That is, let M be a Turing machine. Then we may denote $M(w)$ as the result of execution (1 if it accepts, 0 if it rejects, and undefined if it does not halt). If a Turing machine takes two inputs, we may denote this $M(w_1, w_2)$.

Definition B.2.4. A Turing machine is said to recognise a language if it accepts for every string in the language but does not accept for any string outside of the language. A Turing machine is said to decide a language if it accepts for every string in the language and rejects for every other string.

This is an important distinction to make, on a given input a Turing machine may: accept; reject; or never halt. So if a Turing machine recognises a language, it may or may not halt on words not in that language. But if a Turing machine decides a language, it must *always halt.*

Definition B.2.5. A language is said to be *Turing-recognisable* if there is a Turing machine that recognises it. A simiarly definition exists for *Turing-decidable.*

We now introduce a variant of Turing machines.

Definition B.2.6 (Multitape Turing machine)**.** A multitape Turing machine M is defined the same as a regular Turing machine accept with several tapes each with their own head. We only modify the transition function in our formal definition:

$$\delta : Q \times \Pi^k \to Q \to \Pi^k \times \{L, R\}^k$$

where $k \in \mathbb{N}$ is the number of tapes.

Theorem B.2.7. *Every multitape Turing machine has an equivalent single tape Turing machine.*

Sketch of proof. One way to show this is by putting all the tape contents onto a single tape and separating them with a delimiting character. We then introduce a new letter to the tape alphabet for each existing alphabet, we replace a character with its corresponding new letter to indicate where the tape's head is. □

Theorem B.2.8 (Church-Turing)**.** *The intuitive notion of an algorithm is equivalent to concept of an algorithm as defined by Turing machines.*

By this, we mean that any program written in any of the familiar programming languages such as Python can be simulated on a Turing machine.

We recall earlier we introduced the notation $\langle x \rangle$ for a binary encoding of any object x. We can also do this for Turing machines, and in fact for any finite alphabet.

Proposition B.2.9. *Every Turing machine M can be encoded as a word over a finite alphabet.*

Theorem B.2.10. *There is a Turing machine $\mathcal{U}$ that takes a two part input, the encoding of a Turing machine $\langle M \rangle$ and a word w, and simulates M on w.*

Such a Turing machine is called a *universal Turing machine.*

Problem B.2.11 (HaltingProblem).
Instance: let M be a Turing machine and w an input string.
Question: does M terminate on w?

It is clear that HaltingProblem is Turing-recognisable: we run a universal Turing machine on $(\langle M\rangle, w)$ and accept if the computation terminates.

Proposition B.2.12. HaltingProblem *is not Turing-decidable.*

Proof. Let $\mathcal{H}$ be a Turing machine that decides the Halting problem. Construct the Turing machine $\mathcal{D}$ such that

$$\mathcal{D}(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } \mathcal{H}(\langle M\rangle, \langle M\rangle) = 0, \\ \text{loop} & \text{if } \mathcal{H}(\langle M\rangle, \langle M\rangle) = 1, \end{cases}$$

for any Turing machine M. We then run $\mathcal{D}$ on itself. We have two possibilities.

- If $\mathcal{D}$ terminates on $\langle\mathcal{D}\rangle$, then $\mathcal{H}(\langle\mathcal{D}\rangle, \langle\mathcal{D}\rangle) = 0$; that is, $\mathcal{D}$ does not terminate on $\langle\mathcal{D}\rangle$; a contradiction.
- If $\mathcal{D}$ does not terminate on $\langle\mathcal{D}\rangle$, then $\mathcal{H}(\langle\mathcal{D}\rangle, \langle\mathcal{D}\rangle) = 1$; that is, $\mathcal{D}$ does terminate on $\langle\mathcal{D}\rangle$; another contradiction.

□

The HaltingProblem is our first uncomputable function we have seen, and there are many others.

B.3 Complexity theory

We start with some motivation. Consider the language $\mathcal{L} = \{0^n 1^n : n \in \mathbb{Z}_{\geq 0}\}$, we have seen that this is a decidable language. But how much time would a Turing machine need to decide $\mathcal{L}$? We recall the algorithm that we introduced earlier.

(i) Read the initial symbol. If it is blank, we accept. If it is a 1, we reject. Otherwise, we clear the cell and move to the right.

(ii) We scan along each cell until we get to the end. If we find a 0 after a 1 we reject.

(iii) When we get to the end of the string, we clear the far-most 1 and then move back to the first non-blank symbol.

(iv) We repeat this from the start until we accept or reject.

Let M denote the Turing machine representing the above algorithm and we trace M on input $w = 0011$.

Step	State	Tape
0	q_0	$\underline{0}011\sqcup$
1	q_1	$\sqcup\underline{0}11\sqcup$
2	q_1	$\sqcup0\underline{1}1\sqcup$
3	q_2	$\sqcup01\underline{1}\sqcup$
4	q_2	$\sqcup011\underline{\sqcup}$
5	q_3	$\sqcup01\underline{1}\sqcup$
6	q_4	$\sqcup0\underline{1}\sqcup\sqcup$
7	q_4	$\sqcup\underline{0}1\sqcup\sqcup$

Step	State	Tape
8	q_4	$\underline{\sqcup}01\sqcup\sqcup$
9	q_0	$\sqcup\underline{0}1\sqcup\sqcup$
10	q_1	$\sqcup\sqcup\underline{1}\sqcup\sqcup$
11	q_2	$\sqcup\sqcup1\underline{\sqcup}\sqcup$
12	q_3	$\sqcup\sqcup\underline{1}\sqcup\sqcup$
13	q_4	$\sqcup\underline{\sqcup}\sqcup\sqcup\sqcup$
14	q_0	$\sqcup\sqcup\underline{\sqcup}\sqcup\sqcup$
15	q_a	$\sqcup\sqcup\underline{\sqcup}\sqcup\sqcup$

Let $w \in \mathcal{L}$ of length $n \in 2\mathbb{N}$. Then we see that after the algorithm has scanned up the string and back down then repositioned on the first non-zero entry, we are back in q_0 and the string is $w' \in \mathcal{L}$ of length $n-2$. Thus, we have the recurrence relation

$$T(n) = 2n + 1 + T(n-2)$$

where $T(n)$ denotes the (maximum) number of steps M takes to run on an input of length n. This relation has the solution $T(n) = 1/2(n+2)(n+1)$ (this can be derived or proved by induction).

When looking at the running time of an algorithm, we will be looking at the *worst-case*, although one can also investigate the *average-case* and *best-case*.

Definition B.3.1. Let M be a Turing machine that halts on all inputs. The *time complexity* (or *running time*) of M is a function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ denotes the maximum number of steps that M uses on an input of length n.

In the analysis of algorithms, we typically do not care about the specific time complexity function for a given algorithm. Instead, we look at the *limiting behaviour* of the time complexity. To do this, we require some notions to describe this asymptotic behaviour.

Definition B.3.2 (Big O notation)**.** Let f and g be two real-valued functions defined on some unbounded set of real numbers. Then we say $f(n) = O(g(n))$ if for all $\varepsilon > 0$ there is $N \in \mathbb{N}$ such that for all $n > N$ we have $|f(n)| \leq \varepsilon g(n)$.

Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to constant factors.

Example B.3.3. Let $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \in \mathbb{R}[x]$ where $a_n \neq 0$. Then $f(x) = O(x^n)$.

Example B.3.4. Let $f(x) = \log(x^n)$ for some $n \in \mathbb{N}$. Then

$$f(x) = n \log x = O(\log x).$$

Remark B.3.5. We typically use log to denote the natural logarithm (that is, base-e); however, we typically use base-2 when studying computational complexity. We observe that for an arbitrary base $b \in \mathbb{R}_{>0} \setminus \{1\}$,

$$O(\log_2(x)) = O\left(\frac{\log_b(x)}{\log_b(2)}\right) = O(\log_b(x)).$$

Thus, when we write $f(n) = O(\log n)$, specifying the base is not necessary.

O on the set $\{f : S \to \mathbb{R}\}$ is analogous to $\leq$ on the real line, and indeed we have analogues for the relations $<$, $=$, etc.

Definition B.3.6 (Bachmann-Landau notations)**.** Let f and g be two real-valued functions defined on some unbounded set of real numbers.

- (Small O, equivalent to $<$) $f(n) = o(g(n))$ if

$$\forall \varepsilon > 0 \; \exists N \in \mathbb{N} \; \forall n > N : |f(n)| < kg(n).$$

- (Big Omega, equivalent to $\geq$) $f(n) = \Omega(g(n))$ if

$$\forall \varepsilon > 0 \; \exists N \in \mathbb{N} \; \forall n > N : |f(n)| \geq kg(n).$$

- (Small Omega, equivalent to $>$) $f(n) = \omega(g(n))$ if

$$\forall \varepsilon > 0 \; \exists N \in \mathbb{N} \; \forall n > N : |f(n)| > kg(n).$$

- (Big Theta, equivalent to $\approx$) $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Appendix C

Source code

Here we present some selected source code written for this project. We have three main libraries:

(i) a simplex library, implementing efficient data structures for filtered simplicial complexes;

(ii) a Vietoris-Rips construction library, implementing various skeleton and expansion methods; and

(iii) a persistent homology library, implementing various speedups in the computation of persistent homology.

All code was written in Python 3.10.

C.1 Simplex library

This library implements the `Simplex` data structure, and the structures `SimplicialComplex` and `FilteredSimplicialComplex` built on top of this.

C.1.1 Implementation of simplices

The following class implements the `Simplex` data structure.

```
class Simplex:
    __vertices: FrozenSet[int]

    def __init__(self, vertices: Set[int] | FrozenSet[int]):
        if isinstance(vertices, FrozenSet):
            self.__vertices = vertices
        else:
            self.__vertices = frozenset(vertices)

    def __repr__(self):
        output = [str(self.dim), "-Sim("]
        output2 = []
        for vertex in self.vertices:
            output2.append(str(vertex))
        output.append(",".join(output2))
        output.append(")")
        return "".join(output)

```

```
    def __hash__(self) -> int:
        return hash(self.__vertices)

    def __eq__(self, other) -> bool:
        return hash(self) == hash(other)

    def __contains__(self, item: Hashable) -> bool:
        return item in self.__vertices

    def __iter__(self) -> Generator[int, None, None]:
        for vertex in self.__vertices:
            yield vertex

    def __add__(self, other: int):
        vertices = set(self.__vertices)
        vertices.add(other)
        return Simplex(vertices)

    @property
    def dim(self):
        return len(self.__vertices) - 1

    @property
    def vertices(self):
        for vertex in self.__vertices:
            yield vertex

    @property
    def facets(self) -> Generator[Simplex, None, None]:
        for vertex in self.vertices:
            yield Simplex(self.__vertices.difference({vertex}))

    @property
    def faces(self) -> Generator[Simplex, None, None]:
        for face_dim in range(self.dim, 0, -1):
            for face in combinations(self.__vertices, face_dim):
                yield face

    def p_faces(self, p: int) -> Generator[Simplex, None, None]:
        for p_face in combinations(self.__vertices, p + 1):
            yield Simplex(p_face)

    def intersect(self, other: Simplex) -> Simplex:
        simplex = Simplex(self.__vertices.intersection(other.__vertices))
        return simplex

    def union(self, *simplices: Simplex) -> Simplex:
        vertices = self.__vertices
        for simplex in simplices:
            vertices = vertices.union(simplex.__vertices)
        return Simplex(vertices)

```

```
71      def get_vertices(self) -> Set[int]:
72          return set(self.__vertices)
```

C.1.2 Implementation of simplicial complex

The following class implements the `SimplicialComplex` data structure.

```
1   class SimplicialComplex:
2       __simplices: Dict[int, Set[Simplex]]
3       __check_valid: bool
4
5       def __init__(self, check_valid=False):
6           self.__simplices = dict()
7           self.__check_valid = check_valid
8           pass
9
10      def __repr__(self):
11          output = []
12          for p in self.__simplices:
13              for p_simplex in self.__simplices[p]:
14                  output.append(str(p_simplex))
15          return "\n".join(output)
16
17      def __contains__(self, item: Simplex) -> bool:
18          if not isinstance(item, Simplex):
19              raise ValueError(
20                  "Simplicial complex only contains simplex, " +
21                      "contains operation of non-simplex type prohibited.")
22          if item.dim not in self.__simplices:
23              return False
24          return item in self.__simplices[item.dim]
25
26      @property
27      def dim(self):
28          return max(self.__simplices.keys())
29
30      @property
31      def size(self):
32          size = 0
33          for simplices in self.__simplices.values():
34              size += len(simplices)
35          return size
36
37      def __add_vertex(self, simplex: Simplex):
38          if simplex.dim not in self.__simplices:
39              self.__simplices[simplex.dim] = set()
40          self.__simplices[simplex.dim].add(simplex)
41
42      def add_simplex(self, simplex: Simplex):
43          if simplex.dim == 0:
44              self.__add_vertex(simplex)
```

```
            return
        if self.__check_valid:
            for facet in simplex.facets:
                if simplex.dim - 1 not in self.__simplices:
                    raise ValueError("Not all facets are in the complex, "
                      + "cannot add simplex.")
                if facet not in self.__simplices[simplex.dim - 1]:
                    raise ValueError("Not all facets are in the complex, "
                      + "cannot add simplex.")
        self.__add_vertex(simplex)
```

C.1.3 Implementation of filtered simplicial complex

The following class implements the `FilteredSimplicialComplex` data structure.

```
class FilteredSimplicialComplex:
    __simplices: Dict[int, Set[Simplex]]
    __weights: Dict[Simplex, float]
    __check_valid: bool

    def __init__(self, check_valid=False):
        self.__simplices = dict()
        self.__weights = dict()
        self.__check_valid = check_valid
        pass

    def __repr__(self) -> str:
        output = []
        for p in self.__simplices:
            for p_simplex in self.__simplices[p]:
                output.append(f"{str(p_simplex)} \
                    w: {self.__weights[p_simplex]}")
        return "\n".join(output)

    def __contains__(self, item: Simplex) -> bool:
        if not isinstance(item, Simplex):
            raise ValueError(
                "Simplicial complex only contains simplex, " +
                    "contains operation of non-simplex type prohibited.")
        if item.dim not in self.__simplices:
            return False
        return item in self.__simplices[item.dim]

    def __eq__(self, other: FilteredSimplicialComplex):
        if self.__simplices != other.__simplices:
            return False
        if self.__weights != other.__weights:
            return False
        return True

    @property
```

```
37      def dim(self):
38          return max(self.__simplices.keys())
39
40      @property
41      def size(self):
42          size = 0
43          for simplices in self.__simplices.values():
44              size += len(simplices)
45          return size
46
47      def p_simplices(self, p: int) -> Generator[Simplex, None, None]:
48          if p not in self.__simplices:
49              return
50          for p_simplex in self.__simplices[p]:
51              yield p_simplex
52
53      def __add_simplex(self, simplex: Simplex, weight: float):
54          if simplex.dim not in self.__simplices:
55              self.__simplices[simplex.dim] = set()
56          self.__simplices[simplex.dim].add(simplex)
57          self.__weights[simplex] = weight
58
59      def add_simplex(self, simplex: Simplex, weight: float):
60          if simplex.dim == 0:
61              self.__add_simplex(simplex, weight)
62              return
63          if self.__check_valid:
64              for facet in simplex.facets:
65                  if simplex.dim - 1 not in self.__simplices:
66                      raise ValueError("Not all facets are in the complex, "
67                          + "cannot add simplex.")
68                  if facet not in self.__simplices[simplex.dim - 1]:
69                      raise ValueError("Not all facets are in the complex, "
70                          + "cannot add simplex.")
71                  facet_weight = self.__weights[facet]
72                  if facet_weight > weight:
73                      raise ValueError("Not all facets have a lower weight.")
74          self.__add_simplex(simplex, weight)
75
76      def get_weight(self, simplex: Simplex) -> float:
77          if simplex.dim not in self.__simplices:
78              raise ValueError("Simplex not in complex.")
79          if simplex not in self.__simplices[simplex.dim]:
80              raise ValueError("Simplex not in complex.")
81          return self.__weights[simplex]
82
83      def p_simplex_count(self, p: int) -> int:
84          if p not in self.__simplices:
85              return 0
86          return len(self.__simplices[p])
87
88      def get_edge_neighbours(self, vertex: int) -> Set[int]:
```

```
        if 1 not in self.__simplices:
            return set()
        edge_set = set()
        for edge in self.__simplices[1]:
            if vertex in edge:
                edge_set = edge_set.union(edge.get_vertices())
        edge_set.discard(vertex)
        return edge_set

    def reweight(self, simplex: Simplex, weight: float) -> None:
        if simplex.dim not in self.__simplices:
            raise ValueError("Simplex not in complex.")
        if simplex not in self.__simplices[simplex.dim]:
            raise ValueError("Simplex not in complex.")
        self.__weights[simplex] = weight

    def get_simplex_ordering(self) -> List[Simplex]:
        simplex_list = list()
        for simplices in self.__simplices.values():
            simplex_list += simplices
        simplex_list = sorted(simplex_list,
            key=lambda k: self.__weights[k])
        return simplex_list

    def get_weight_ordering(self) -> List[float]:
        return sorted(self.__weights.values())

    def cap(self, weight_limit: float) -> FilteredSimplicialComplex:
        fc = FilteredSimplicialComplex()
        for simplex in self.get_simplex_ordering():
            if self.__weights[simplex] > weight_limit:
                break
            fc.add_simplex(simplex, self.__weights[simplex])
        return fc
```

C.2 Vietoris-Rips construction library

The following is an abstract base class, for which our skeleton and expansion methods inherit from and implement the abstract methods.

```
class VRBase(ABC):
    _points: List[ndarray]
    _complex: FilteredSimplicialComplex | None
    _metric: Callable[[ndarray[float, ...], ndarray[float, ...]], float]
    _epsilon: float
    _is_skeleton_constructed: bool

    def __init__(self, points: List[ndarray], epsilon: float,
                 metric: Callable[[ndarray[float, ...],
                 ndarray[float, ...]], float]):
```

```
        self._points = points
        self._complex = None
        self._metric = metric
        self._epsilon = epsilon
        self._is_skeleton_constructed = False

    def get_complex(self) -> FilteredSimplicialComplex:
        return self._complex

    @abstractmethod
    def compute_skeleton(self):
        pass

    @abstractmethod
    def compute_expansion(self, dim: int):
        pass
```

C.2.1 Skeleton method: brute-force

The following implements the brute-force skeleton method.

```
class SkeletonBruteForce(VRBase, ABC):
    def __init__(self, points: List[ndarray], epsilon: float,
                 metric: Callable[[ndarray[float, ...],
                 ndarray[float, ...]], float]):
        super().__init__(points, epsilon, metric)

    def compute_skeleton(self):
        self._complex = FilteredSimplicialComplex()
        for i, _ in enumerate(self._points):
            self._complex.add_simplex(Simplex({i}), 0)
        for (i, x), (j, y) in combinations(enumerate(self._points), 2):
            dist = self._metric(x, y)
            if dist < self._epsilon:
                self._complex.add_simplex(Simplex({i, j}), dist)
        self._is_skeleton_constructed = True
```

C.2.2 Skeleton method: sklearn

The following implements the sklearn skeleton method.

```
class Sklearn(VRBase, ABC):
    def compute_skeleton(self):
        self._complex = FilteredSimplicialComplex()
        for i, _ in enumerate(self._points):
            self._complex.add_simplex(Simplex({i}), 0)

        points = array(self._points)
        neighbours = neighbors.NearestNeighbors(
```

```
            radius=self._epsilon).fit(points)
        distances_list, indices_list = neighbours.radius_neighbors(points)
        for i, (distances, indices) in enumerate(
            zip(distances_list, indices_list)):
            for distance, j in zip(distances[1:], indices[1:]):
                self._complex.add_simplex(Simplex({i, j}), distance)
        self._is_skeleton_constructed = True
```

C.2.3 Expansion method: brute-force

The following implements the brute-force expansion method.

```
class ExpansionBruteForce(VRBase, ABC):
    def __init__(self, points: List[ndarray], epsilon: float,
                 metric: Callable[[ndarray[float, ...], ndarray[float, ...]], float]):
        super().__init__(points, epsilon, metric)

    def compute_expansion(self, dim: int):
        if not self._is_skeleton_constructed:
            raise ReferenceError("Skeleton not constructed.")
        for simplex_dim in range(2, dim + 1):
            for elem in combinations(self._complex.p_simplices(simplex_dim - 1),
                simplex_dim + 1):
                flag = True
                max_dist = 0
                for (x, y) in combinations(elem, 2):
                    if x.intersect(y).dim == -1:
                        flag = False
                        break
                    max_dist = max([max_dist, self._complex.get_weight(x),
                        self._complex.get_weight(y)])
                if flag:
                    union = elem[0].union(*elem[1:])
                    if union.dim == simplex_dim:
                        self._complex.add_simplex(union, max_dist)
            if self._complex.p_simplex_count(simplex_dim) < simplex_dim + 1:
                break
```

C.2.4 Expansion method: incremental

The following implements the incremental expansion method.

```
class Incremental(VRBase, ABC):
    def __init__(self, points: List[ndarray], epsilon: float,
                 metric: Callable[[ndarray[float, ...], ndarray[float, ...]], float]):
        super().__init__(points, epsilon, metric)
        self.__new_complex = FilteredSimplicialComplex()

    def lower_neighbours(self, v: int) -> Set[int]:
```

```
        edge_neighbours = self._complex.get_edge_neighbours(v)
        lower_edge_neighbours = set()
        for neighbour in edge_neighbours:
            if neighbour < v:
                lower_edge_neighbours.add(neighbour)
        return lower_edge_neighbours

    def add_cofaces(self, level: int, simplex: Simplex, lower_neighbours: Set[int]):
        stack = [(simplex, lower_neighbours)]
        while stack:
            current_simplex, current_neighbours = stack.pop()
            self.__new_complex.add_simplex(current_simplex, 0)

            if current_simplex.dim < level:
                for neighbour in current_neighbours:
                    coface = current_simplex + neighbour
                    new_lower_neighbours = current_neighbours.intersection(
                        self.lower_neighbours(neighbour))
                    stack.append((coface, new_lower_neighbours))

    def compute_weights(self):
        for edge in self.__new_complex.p_simplices(1):
            (vertex_x, vertex_y) = edge.get_vertices()
            weight = self._metric(self._points[vertex_x], self._points[vertex_y])
            self.__new_complex.reweight(edge, weight)

        for p in range(2, self.__new_complex.dim + 1):
            for p_simplex in self.__new_complex.p_simplices(p):
                weight = max(self.__new_complex.get_weight(facet) \
                    for facet in p_simplex.facets)
                self.__new_complex.reweight(p_simplex, weight)

    def compute_expansion(self, dim: int):
        if self._complex.p_simplex_count(0) == 0:
            return
        for simplex in self._complex.p_simplices(0):
            (vertex,) = simplex.get_vertices()
            lower_neighbours = self.lower_neighbours(vertex)
            self.add_cofaces(dim, simplex, lower_neighbours)
        self.compute_weights()
        self._complex = self.__new_complex
```

C.2.5 Expansion method: inductive

The following implements the inductive expansion method.

```
class Inductive(VRBase, ABC):
    def __init__(self, points: List[ndarray], epsilon: float,
                 metric: Callable[[ndarray[float, ...], ndarray[float, ...]], float]):
        super().__init__(points, epsilon, metric)

```

```
    def lower_neighbours(self, v: int) -> Set[int]:
        edge_neighbours = self._complex.get_edge_neighbours(v)
        lower_edge_neighbours = set()
        for neighbour in edge_neighbours:
            if neighbour < v:
                lower_edge_neighbours.add(neighbour)
        return lower_edge_neighbours

    def compute_weights(self):
        for p in range(2, self._complex.dim + 1):
            for p_simplex in self._complex.p_simplices(p):
                weight = max(self._complex.get_weight(facet) for facet in p_simplex.facets)
                self._complex.reweight(p_simplex, weight)

    def compute_expansion(self, dim: int):
        for simplex_dim in range(2, dim + 1):
            stack = list(self._complex.p_simplices(simplex_dim - 1))
            for simplex in self._complex.p_simplices(simplex_dim - 1):
                common_lower_neighbours = None
                for vertex in simplex:
                    if common_lower_neighbours is None:
                        common_lower_neighbours = self.lower_neighbours(vertex)
                    else:
                        common_lower_neighbours = common_lower_neighbours.intersection(
                            self.lower_neighbours(vertex))
                for neighbour in common_lower_neighbours:
                    self._complex.add_simplex(simplex + neighbour, 0)
        self.compute_weights()
```

C.3 Persistent homology library

The following is an abstract base class, for which our persistent homology implements inherit and implement. We also have defined here data classes for persistence points and persistence diagrams.

```
@dataclass(frozen=True)
class PersPoint:
    born_index: int
    die_index: int
    born: float
    die: float
    dim: int


class PersDiag:
    __points: Dict[int, Counter[PersPoint]]
    __point_count: int

    def __init__(self):
        self.__points = dict()
```

```
        self.__point_count = 0

    def __repr__(self) -> str:
        return f"PersDiag with {self.__point_count} points"

    def __eq__(self, other: PersDiag):
        if self.__point_count != other.__point_count:
            return False
        if self.__points.keys() != other.__points.keys():
            return False
        for key in self.__points.keys():
            if self.__points[key] != other.__points[key]:
                return False
        return True

    @property
    def points(self) -> Generator[PersPoint, None, None]:
        for p_pers_points in self.__points.values():
            for p_pers_point in p_pers_points:
                yield p_pers_point

    def p_points(self, p: int) -> Generator[PersPoint, None, None]:
        if p not in self.__points:
            return
        for pers_point in self.__points[p]:
            yield pers_point

    def add_point(self, point: PersPoint) -> None:
        if point.dim not in self.__points:
            self.__points[point.dim] = collections.Counter()
        self.__points[point.dim][point] += 1
        self.__point_count += 1


class PersHomBase(ABC):
    _pers_diag: PersDiag | None

    def __init__(self):
        self._pers_diag = None

    @abstractmethod
    def compute(self) -> None:
        pass

    def get_pers_diag(self) -> PersDiag:
        assert self._pers_diag is not None,
            "It seems like that the persistent diagram has not been constructed."
        return self._pers_diag
```

C.3.1 Standard algorithm

The following implements the standard algorithm, S.

```
class PersHom1(PersHomBase):
    __simplex_ordering: List[Simplex]
    __simplex_count: int
    __weights: List[float]
    __boundary_matrix: np.ndarray
    __working_matrix: np.ndarray | None

    def __init__(self, filtered_complex: FilteredSimplicialComplex):
        super().__init__()
        self.__simplex_ordering = filtered_complex.get_simplex_ordering()
        self.__simplex_to_order = dict()
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__simplex_to_order[simplex] = i
        self.__simplex_count = len(self.__simplex_ordering)
        self.__weights = filtered_complex.get_weight_ordering()
        self.__boundary_matrix = np.zeros((len(self.__simplex_ordering),
            len(self.__simplex_ordering)))
        for i, simplex in enumerate(self.__simplex_ordering):
            for facet in simplex.facets:
                if facet not in self.__simplex_to_order:
                    if facet.dim != -1:
                        raise Exception()
                    continue
                j = self.__simplex_to_order[facet]
                self.__boundary_matrix[j][i] = 1
        self.__working_matrix = None

    def __get_last_in_col(self, i: int) -> int | None:
        if np.all(self.__working_matrix[:, i] == 0):
            return None
        return max(j for j, val in enumerate(self.__working_matrix[:, i]) if val == 1)

    def __add_col(self, i: int, j: int) -> None:
        """
        R_i <- R_i + R_j
        """
        assert i < self.__simplex_count and j < self.__simplex_count,
            "Column index out of range."
        assert self.__working_matrix is not None, "Working matrix is not initialised."
        self.__working_matrix[:, i] = np.mod(
            self.__working_matrix[:, i] + self.__working_matrix[:, j], 2)

    def compute(self) -> None:
        low: Dict[int, int] = dict()
        self.__working_matrix = self.__boundary_matrix.copy()

        for i in range(self.__simplex_count):
            last_in_col = self.__get_last_in_col(i)
            if last_in_col is None:
```

```
                continue
            if last_in_col not in low:
                low[last_in_col] = i
                continue
            while last_in_col in low and last_in_col is not None:
                competing_col = low[last_in_col]
                self.__add_col(i, competing_col)
                last_in_col = self.__get_last_in_col(i)
            if last_in_col is not None:
                low[last_in_col] = i

        self._pers_diag = PersDiag()
        for row, col in low.items():
            pers_point = PersPoint(
                born_index=row,
                die_index=col,
                born=self.__weights[row],
                die=self.__weights[col],
                dim=self.__simplex_ordering[row].dim
            )
            self._pers_diag.add_point(pers_point)

```

C.3.2 Sparse matrix implementation

The following implements the sparse matrix algorithm, SS.

```
class PersHom2(PersHomBase):
    """
    Implements the standard persistent homology algorithm, with no speed-ups.
    """
    __simplex_ordering: List[Simplex]
    __simplex_count: int
    __weights: List[float]
    __boundary_matrix: List[Set[int]]
    __working_matrix: List[Set[int]] | None

    def __init__(self, filtered_complex: FilteredSimplicialComplex):
        super().__init__()
        self.__simplex_ordering = filtered_complex.get_simplex_ordering()
        self.__simplex_to_order = dict()
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__simplex_to_order[simplex] = i
        self.__simplex_count = len(self.__simplex_ordering)
        self.__weights = filtered_complex.get_weight_ordering()
        self.__boundary_matrix = []
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__boundary_matrix.append(set())
            for facet in simplex.facets:
                if facet not in self.__simplex_to_order:
                    if facet.dim != -1:
```

```
                        raise Exception()
                    continue
                j = self.__simplex_to_order[facet]
                self.__boundary_matrix[i].add(j)
        self.__working_matrix = None

    def __get_last_in_col(self, i: int) -> int | None:
        if not self.__working_matrix[i]:
            return None
        return max(j for j in self.__working_matrix[i])

    def __add_col(self, i: int, j: int) -> None:
        """
        R_i <- R_i + R_j
        """
        for k in self.__working_matrix[j]:
            if k not in self.__working_matrix[i]:
                self.__working_matrix[i].add(k)
                continue
            self.__working_matrix[i].remove(k)

    def compute(self) -> None:
        low: Dict[int, int] = dict()
        self.__working_matrix = list()
        for i in self.__boundary_matrix:
            self.__working_matrix.append(i.copy())

        for i in range(self.__simplex_count):
            last_in_col = self.__get_last_in_col(i)
            while last_in_col in low and last_in_col is not None:
                competing_col = low[last_in_col]
                self.__add_col(i, competing_col)
                last_in_col = self.__get_last_in_col(i)
            if last_in_col is not None:
                low[last_in_col] = i

        self._pers_diag = PersDiag()
        for row, col in low.items():
            pers_point = PersPoint(
                born_index=row,
                die_index=col,
                born=self.__weights[row],
                die=self.__weights[col],
                dim=self.__simplex_ordering[row].dim
            )
            self._pers_diag.add_point(pers_point)
```

C.3.3 Reduction by killing

The following implements the reduction by killing algorithm, SSR.

```python
class PersHom3(PersHomBase):
    __simplex_ordering: List[Simplex]
    __simplex_to_order: Dict[Simplex, int]
    __simplex_count: int
    __dim: int
    __weights: List[float]
    __boundary_matrix: List[Set[int]]
    __working_matrix: List[Set[int]] | None

    def __init__(self, filtered_complex: FilteredSimplicialComplex):
        super().__init__()
        self.__simplex_ordering = filtered_complex.get_simplex_ordering()
        self.__simplex_to_order = dict()
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__simplex_to_order[simplex] = i
        self.__simplex_count = len(self.__simplex_ordering)
        self.__dim = filtered_complex.dim
        self.__weights = filtered_complex.get_weight_ordering()
        self.__boundary_matrix = []
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__boundary_matrix.append(set())
            for facet in simplex.facets:
                if facet not in self.__simplex_to_order:
                    if facet.dim != -1:
                        raise Exception()
                    continue
                j = self.__simplex_to_order[facet]
                self.__boundary_matrix[i].add(j)
        self.__working_matrix = None

    def __get_last_in_col(self, i: int) -> int | None:
        if not self.__working_matrix[i]:
            return None
        return max(j for j in self.__working_matrix[i])

    def __add_col(self, i: int, j: int) -> None:
        """
        R_i <- R_i + R_j
        """
        for k in self.__working_matrix[j]:
            if k not in self.__working_matrix[i]:
                self.__working_matrix[i].add(k)
                continue
            self.__working_matrix[i].remove(k)

    def compute(self) -> None:
        low: Dict[int, int] = dict()
        self.__working_matrix = list()
        for i in self.__boundary_matrix:
            self.__working_matrix.append(i.copy())

        for dim in range(self.__dim, -1, -1):
```

```
            for i in range(self.__simplex_count):
                if self.__simplex_ordering[i].dim == dim:
                    last_in_col = self.__get_last_in_col(i)
                    while last_in_col in low and last_in_col is not None:
                        competing_col = low[last_in_col]
                        self.__add_col(i, competing_col)
                        last_in_col = self.__get_last_in_col(i)
                    if last_in_col is not None:
                        low[last_in_col] = i
                        self.__working_matrix[last_in_col] = set()

        self._pers_diag = PersDiag()
        for row, col in low.items():
            pers_point = PersPoint(
                born_index=row,
                die_index=col,
                born=self.__weights[row],
                die=self.__weights[col],
                dim=self.__simplex_ordering[row].dim
            )
            self._pers_diag.add_point(pers_point)
```

C.3.4 Reducing the coboundary matrix

The following implements the persistent cohomology algorithm, SSRCoh.

```
class PersHom4(PersHomBase):
    __simplex_ordering: List[Simplex]
    __simplex_to_order: Dict[Simplex, int]
    __simplex_count: int
    __dim: int
    __weights: List[float]
    __boundary_matrix: List[Set[int]]
    __working_matrix: List[Set[int]] | None

    def __init__(self, filtered_complex: FilteredSimplicialComplex):
        super().__init__()
        self.__simplex_ordering = filtered_complex.get_simplex_ordering()
        self.__simplex_to_order = dict()
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__simplex_to_order[simplex] = i
        self.__simplex_count = len(self.__simplex_ordering)
        self.__dim = filtered_complex.dim
        self.__weights = filtered_complex.get_weight_ordering()
        self.__boundary_matrix = []
        for i, simplex in enumerate(self.__simplex_ordering):
            self.__boundary_matrix.append(set())
            for facet in simplex.facets:
                if facet not in self.__simplex_to_order:
                    if facet.dim != -1:
                        raise Exception()
```

```
                    continue
                j = self.__simplex_to_order[facet]
                self.__boundary_matrix[i].add(j)
        self.__coboundary_matrix = list()
        for i in range(self.__simplex_count):
            self.__coboundary_matrix.append(set())
        for col_i, row_is in enumerate(self.__boundary_matrix):
            for row_i in row_is:
                self.__coboundary_matrix[self.__simplex_count - 1 - row_i] \
                    .add(self.__simplex_count - 1 - col_i)
        self.__working_matrix = None

    def __get_last_in_col(self, i: int) -> int | None:
        if not self.__working_matrix[i]:
            return None
        return max(j for j in self.__working_matrix[i])

    def __add_col(self, i: int, j: int) -> None:
        """
        R_i <- R_i + R_j
        """
        for k in self.__working_matrix[j]:
            if k not in self.__working_matrix[i]:
                self.__working_matrix[i].add(k)
                continue
            self.__working_matrix[i].remove(k)

    def compute(self) -> None:
        low: Dict[int, List[int]] = dict()
        self.__working_matrix = list()
        cocycles = []
        marked = set()
        for i in range(self.__simplex_count):
            cocycles.append({i})

        self._pers_diag = PersDiag()

        for i in range(self.__simplex_count - 1, -1, -1):
            indices = set()
            for j in self.__boundary_matrix[self.__simplex_count - 1 - i]:
                j_index = self.__simplex_count - 1 - j
                if j_index in marked:
                    continue
                for k in cocycles[j_index]:
                    if k in indices:
                        indices.remove(k)
                        continue
                    indices.add(k)
            if not indices:
                continue
            marked.add(i)
            p = min(indices)
```

```
            marked.add(p)

            # print(cocycles)
            row_i = self.__simplex_count - 1 - p
            col_i = self.__simplex_count - 1 - i
            pers_point = PersPoint(
                born_index=row_i,
                die_index=col_i,
                born=self.__weights[row_i],
                die=self.__weights[col_i],
                dim=self.__simplex_ordering[row_i].dim
            )
            self._pers_diag.add_point(pers_point)

            for j in indices:
                if j == p:
                    continue
                cocycles[j].add(p)
```