Graphlet Counting for Topological Data Analysis

Candidate: Marco Guerra

16th July 2018
## Contents

1 Introduction ........................................ 3

2 Background ........................................... 6
   2.1 Simplicial Complexes ............................ 6
   2.5 Complexes with a Metric .......................... 9
   2.11 The Free Group and the Fundamental Group ... 12
   2.18 Simplicial Homology ............................ 17
   2.30 Persistent Homology ............................. 22

3 Shortest Homology Basis .............................. 24
   3.1 Prior Work ....................................... 26
   3.3 Cohomology and Simplex Annotation. Homology Basis .... 28
   3.6 Methodology ..................................... 31
   3.8 Implementation .................................. 33

4 Results and Applications ............................ 42
   4.1 Testing .......................................... 42
   4.2 C. Elegans ....................................... 54

5 Appendix - Code ...................................... 65
1 Introduction

Abstract

The field of Topological Data Analysis is a recent approach to the task of extracting information from especially troublesome datasets. The rationale of applying topological methods lies in the essentiality of the concept of topological shape, which only accounts for the effective structure of data and is blind to the dimensionality and particular metric, features that in some cases convey more noise than information. The main tool to have been employed in this endeavor is persistent homology, a concept in algebraic topology which extends the homological group representation of a space $X$ to a discrete filtration of subspaces of $X$. In particular, we assume to have a chain of gradually more refined simplicial complexes over a given set of data points, with the corresponding inclusion maps between them: homology provides a computable method to count the holes, while persistence is applied to analyze their evolution as connectivity grows [12].

In the present work we introduce a minimality constraint and focus on computing the shortest possible basis of the first homology group $H_1$ of a weighted simplicial complex. The task is proved to be an NP-hard problem for $H_k$ with $k > 1$ [7], but the $k=1$ case is subject of recent research. Taking the lead from the recent work of Dey [6], which borrows ideas from previous studies ([8],[9]) to improve computational complexity, I have implemented a polynomial-time algorithm for the shortest homology basis of a simplicial complex over $\mathbb{Z}_2$ coefficients and extended it to compute persistence over a family of its refinements. The final objective is to experiment with existing datasets of neuroscientific measurements, in the light of the framework proposed by [14], where a new topological object called homological scaffold is introduced to evaluate brain neuron activity correlations at a mesoscopic level, interpreting holes as inhomogeneities in the network structure. The algorithm has been tested on real-world examples concerning neural activity of a model organism, the nematode worm C. Elegans. The foreseeable developments include incorporating the method in the pipeline of the scaffold computation, as an effort to shed light on challenging questions in neuroscience.

Motivation

It is a well-established fact that the modern era of data calls for increasingly refined approaches to obtain knowledge out of the unprecedented amount of information at our disposal. Machine-gathered data is cheap and abundant, but its processing has from the beginning had to pay the price of its sheer size; besides the obvious expenses of dealing with a higher-than-ever volume of records, along with the Big Data come new challenges that are tied to the very structure (or lack thereof) of the information management process:
for example, information is not always reliable, invariably noisy, often unstructured and intractably high-dimensional, and sometimes it is even hard to compute a metric on data points. Classic approaches to the problem broadly involve some kind of dimensionality reduction, and various methods both deterministic and stochastic have been proposed to project data on smaller-dimensional subspaces. Their focus is on finding an efficient and sensible way of doing so while maintaining a sufficient sense of distance and the most possible amount of variance. Another recent approach which has gained momentum in the last decade tackles the problem in a novel and somehow radical way: geometric relations in a dataset can approximately be represented as closeness relations through manifold sampling techniques. Consequently, a purely topological method can arise, concerned solely with the continuous shape of data, which is blind altogether to the dimension of the feature space; as such it avoids the curse of dimensionality while at the same time providing robustness against noise, outliers and lack of structure.
Acknowledgments

For a brief note of gratitude: I cannot give thanks enough for the essential collaboration and help of Alessandro De Gregorio. Your craft was invaluable. And, in many ways, to Francesco Bellini.
2 Background

2.1 Simplicial Complexes

The essential topological tool we shall utilize to encode relationships between objects is a generalization of the concept of graph. A graph is usually described by a pair of sets, one of vertices and one of edges: which is, a group of objects and some binary relations between them. In general we need to describe higher-order relations between more than two entities, such as triplets and so on. This can be formally encoded by the following definition:

**Definition 2.2. Simplicial Complex**

*Let* $S$ *be a non-empty discrete set. An abstract simplicial complex is a collection* $X$ *of finite subsets of* $S$, *closed under restriction.*

In words a simplicial complex $X$ is a subset of the power set of $S$, with the additional property that every element of $X$ must be made up of other elements of $X$. These elements are called *simplices*, more precisely a $k$-simplex is an element of $X$ of cardinality $k + 1$. The subsets of a simplex are called *faces*, and are of course simplices themselves by definition.

If we consider the elements of $S$ to be geometrical points (which is not necessarily the case), we therefore see that a 0-simplex is a single point. A 1-simplex is an (undirected) edge between two points, whose 0-faces are the points. A 2-simplex is a triangle, whose faces are the 3 edges and the 3 points. A 3-simplex is a tetrahedron, subdivided in 4 triangular 2-faces, 6 linear 1-faces and 4 0-simplices, the points.

![Figure 1: An example of a simplicial complex and its graphical representation.](image)

In this frame of reference, we see that a graph is a simplicial complex.
containing at most 1-simplices, i.e. \( G = (V, E) \) where \( V \subset X \) contains the 0-simplices and \( E \subset X \) the 1-simplices.

**Example 2.3.** Consider an undirected graph, defined in the classic sense as \( G = (V, E) \), and say for example \( V = \{a, b, c\} \) and \( E = \{\{a, b\}, \{a, c\}\} \)

![Figure 2: The graph in example 2.3](image)

It is easy to interpret \( G \) as a simplicial complex \( X \) by defining

\[
X := V \cup E = \{\{a, b\}, \{a, c\}, \{a\}, \{b\}, \{c\}\}
\]

which clearly follows definition 2.2.

**Topology of Simplicial Complexes**

In order to discuss the *shape* of an object, it is necessary to at least provide it with a topology. We therefore set forth to construct a suitable open set structure. Let us begin by defining the standard k-simplex

**Definition 2.4.** We call the standard k-simplex the subset of \( \mathbb{R}^n \)

\[
\Delta^k := \{ x \in [0, +\infty)^{k+1} : \sum_{i=0}^{k} x_i = 1 \}
\]

Therefore, the standard 0-simplex is the point 1 \( \in \mathbb{R} \), the 1-simplex is a line segment connecting points (0, 1) and (1, 0) in \( \mathbb{R}^2 \), and so on. One sees that the \((k - 1)\)-faces of \( \Delta^k \) are \( k+1 \) copies of \( \Delta^{k-1} \), and in general faces of \( \Delta^k \) are standard \( j \)-simplices, with \( j < k \). The construction of a topological structure for a generic value of \( k \) is performed inductively via a procedure of *attaching maps*, which connect faces along the boundary of a higher-dimensional simplex. Define the *k-skeleton* of a complex \( X \) as the quotient

\[
X^{(k)} := \left( X^{(k-1)} \bigcup \prod_{\sigma : \dim \sigma = k} \Delta^k \right) / \sim \quad (2.4.1)
\]

where:
• $X^{(0)} = S$

• $\amalg$ represents disjoint union.

• $\sim$ is an equivalence relation that identifies the faces of a standard simplex with the corresponding subsets in the lower dimensional $j$-skeleton.

The full union $X := \bigcup_{k=0}^{\infty} X^{(k)}$ defines the simplicial complex as a topological space (note the same symbol denotes the actual set-theoretic object as well). The natural choice is to give this space the weak topology: a set $U \in X$ is open iff all of its projections onto the subspaces $X^{(k)}$ are continuous, i.e. iff $U \cap X^{(k)}$ is open $\forall k$.

For completeness, notice that other structure can arise through the recursive relation (2.4.1) modifying its base component from the standard simplex to, for example, the $k$-cube $[0,1]^k$, or modifying the attaching mechanism from the simple identification between borders of a simplex and one-step-smaller simplices, to a very general continuous map between the two. In the first case, the reasonably tame cubical complexes appear, whereas the second case gives rise to some of the most flexible (and complicated) topological spaces, known as CW complexes.
2.5 Complexes with a Metric

The set-theoretic notion of simplicial complex is especially suitable for topological applications, because it only relies onto inclusion. In most real-world application, however, topology is only obtained as a byproduct of the metric, as most phenomena are measured in some kind of metric space. It is therefore only natural that there exist ways to construct a simplicial complex from data points that lie in $\mathbb{R}^n$, defining relations on the basis of distance. Let us first provide some useful constructions, and later show how these relate to the concept of metric.

**Definition 2.6. Flag, or Clique Complex:** We define a flag complex or clique complex as the maximal simplicial complex having any given graph as its 1-skeleton.

In practise, a clique complex introduces some sort of constraint on the presence of high-dimensional simplices: requiring it to be maximal, every time the underlying graph contains three edges in the form of a triangle, the corresponding 2-simplex must belong to the complex. By the same token, every four triangles that form a tetrahedron must be "filled", and so on: a clique complex is said to fill a frame.

**Definition 2.7. Nerve of a covering:** Let $\mathcal{U} = \{U_\alpha\}$ be a family of subsets of a topological space. The nerve of $\mathcal{U}$ is a simplicial complex such that its $k$-simplices are the non-empty intersections of $k + 1$ elements of $\mathcal{U}$.

The 0-skeleton of this complex is made up of the subsets, its 1-skeleton are the pairs of subsets that intersect on nonempty sets, etc. Notice that this is indeed a well-defined simplicial complex, because if a family of sets has nonempty intersection, all its sub-families do as well. Notice that a clique complex is remarkably efficient in terms of the input one needs to define it: once the 0- and 1-skeletons are provided, the full complex can be determined uniquely. Conversely, a nerve complex, which lacks the maximality condition, must be explicitly specified in full, with a noticeable increase in complexity.

Let us now see two important classes of simplicial complexes, defined from a point cloud, i.e. a set of points in a normed vector space and therefore intrinsically metric, that implement the aforementioned schemes.
Definition 2.8. Vietoris-Rips Complex: Let $Q \subset \mathbb{R}^n$ be a discrete subset, whose elements we regard as data points, and let $\epsilon > 0$. $VR_\epsilon(Q)$ is the simplicial complex over the discrete set $Q$ whose simplices are all the finite collections of points of $Q$ whose pairwise distance is less than or equal to $\epsilon$.

In words, the Vietoris-Rips complex of radius $\epsilon$ is built by computing the pairwise distance of points and generating the 1-skeleton accordingly. Then, higher-dimensional simplices are simply built by grouping together sets of points such that all relative distances are within the $\epsilon$ threshold. This amounts to computing only $n(n-1)/2$ distances (keep in mind that, for high-dimensional spaces, computing a distance is not inexpensive). We can easily see that $VR_\epsilon$ is the metric counterpart of a clique complex: if three edges exist and form a triangle, the corresponding 2-simplex satisfies the definition and hence belongs to the complex. This is equivalent to imposing the maximality condition, and therefore the Vietoris-Rips complex is built solely on the basis of the 1-skeleton. Hence it is clique.

Let $S : VR \to \mathbb{R}^n$ be the projection map which sends singletons in $VR$ to the corresponding points in $\mathbb{R}^n$, and sends any $k$-simplex into its corresponding convex hull. We define the shadow $Sh$ of the complex as the image of projection $S$. It is easy to see that, tweaking $\epsilon$ and the cardinality of $Q$, it is possible to obtain simplices in $VR$ of dimension (potentially much) higher than $n$: this is a clear limitation if one intends to employ the VR complex to approximate an unknown geometric structure, usually a differentiable manifold, of which the point cloud is supposedly a sample. Indeed we can make this statement formal by observing that, given the suitable algebraic and topological structures, $VR$ and $Sh$ cannot be homeomorphic due to the difference in their dimensions ([23]). A well-known effort to overcome this flaw is given by a different kind of metric realization of a simplicial complex.

Definition 2.9. Čech Complex: Let $Q \subset \mathbb{R}^n$ be a discrete subset as above, we define $C_\epsilon(Q)$ as a simplicial complex whose $k$-simplices are sets of $k+1$ points of $Q$ such that the intersection of $\epsilon$-balls centered in those points is not empty.

The difference with VR lies in the fact that one does just check pairwise distances, but requires a $k+1$-fold intersection to be nonempty in order to introduce a $k$-simplex. This more stringent condition obviously implies $\tilde{C}_\epsilon(Q) \subseteq VR_\epsilon(Q)$, thus maximality is not satisfied in general and $\tilde{C}_\epsilon$ is not clique. Moreover, checking for all $k+1$-fold intersections, where
$k = 1, \ldots, |Q| - 1$ is vastly more expensive than simply computing the $1$-skeleton, and requires more memory to store.

On the other hand the nerve lemma ensures that a such complex comes with a useful property:

**Property 2.10.** $\check{C}_\epsilon(Q)$ is homotopic to the $\epsilon$-hull around $Q$.

This is due to the fact that $\check{C}_\epsilon$ is a particular case of a nerve complex: in particular, a nerve complex with the subsets $U_\alpha$ being the $\epsilon$-balls centered in points of $Q$. Therefore, the Čech complex is topologically superior in its ability to mimic the structure of the data points that it comes from.
2.11 The Free Group and the Fundamental Group

The Free Group

In order to give a proper theoretical grounding to the concept of homology to follow, let us briefly introduce the concept of free group. Given a set of groups $G = \{G_\alpha\}$, consider the set of words (strings of objects) $g_1g_2...g_m$ with finite $m$, such that all $g_i$’s belong to some group in $G$, they are not its identity, and no two adjacent letters belong to the same group. Such a word is called reduced. It is clearly always possible to obtain a reduced word from any string of $g_i$’s, by applying group operations and canceling out the identities, repeating if necessary. Since $m$ is finite this procedure terminates until a reduced word (potentially the empty string) is obtained. It is not too simple to verify that

**Property 2.12.** There exists only one reduced form of any word.

An important consequence of this fact is the following lemma

**Property 2.13.** ($x \sim y$ iff $x, y$ have the same reduced form) is an equivalence relation

So now we can consider the set of finite-length words, modulo the equivalence relation induced by reduction.

Let us add to the set of words an operation: given two words, they can be concatenated. Then the empty word functions as the identity element. Concatenation also has another important characteristic:

**Property 2.14.** String concatenation and reduction commute.

I.e., it is the same to reduce two words and then concatenate them, or to first join them and then reduce the resulting string. Therefore, the concatenation operation can be lifted to the quotient set (modulo reduction), yielding a well-defined operation on equivalence classes of words. The class of the empty word still functions as the identity element, and it is easy to produce the general construction of the inverse: given any (reduced) word $g := g_1...g_m$, due to the group structure of the $G_\alpha$’s one can form the word $h := g_m^{-1}...g_1^{-1}$ (which is still reduced). It clearly holds that $gh = hg = 0$, where 0 identifies the identity of the quotient set, i.e. the equivalence class of the empty word.

In order to conclude that this algebraic object is indeed a group, one finally
needs to prove that concatenation is also associative. To prove it directly would be rather complicated and tedious: instead, one can build a map from the quotient set to a known associative group, and show that this map is also a homomorphism. Then we have indirectly proven associativity, and can conclude the final result.

**Property 2.15.** The set of words modulo reduction, equipped with the concatenation operation, is a group, and is called the Free Group.

Sometimes, one can specify a nonempty set $S$ to work as an alphabet, and build a set of symbols $W = S \cup S^{-1}$ made up of letters of $S$ and their inverses $S^{-1}$, instead of giving groups $G_\alpha$ explicitly. In that case, we say that $F$ is the Free Group on $S$.

**Example 2.16.** Consider the simple alphabet $S := \{1\}$. Let us, in this case, denote its inverse by $-1$. The free group $F(S)$ is, up to isomorphism and some ingenuity, the group of integers equipped with the sum $(\mathbb{Z}, +)$. Indeed, elements of $F(\{1\})$ are reduced strings of 1’s and -1’s, which is to say they are either empty or entirely made of either 1’s or -1’s. Consider the function on $F(\{1\})$ which computes the sum of the elements of the reduced representative of a word class. It holds that (indicating concatenation as $\circ$)

$$s : (F(\{1\}), \circ) \to (\mathbb{Z}, +)$$

is a group homomorphism, as

$$s(h \circ g) = s(h) + s(g)$$

Furthermore, $\forall z \in \mathbb{Z}$ there exists exactly one reduced word $g$ such that $s(g) = z$, and therefore $s$ is an isomorphism. On the other hand, it is rather involved to give a description of the free group on two generators.

Knowledge of the free group will prove handy in the construction of simplicial homology, as a source of the algebraic structure of the $k$-chains that is the core of the subject.

**The Fundamental Group** Let $(X, x_0)$ be a pointed topological space. We call a loop a closed continuous curve $\gamma : I \to X$ such that $\gamma(0) = \gamma(1) = x_0$. Consider homotopy equivalence: if there exists a continuous map of maps
Consider any two loops \( \gamma, \gamma' \). Define the \textit{concatenation} \( \gamma \circ \gamma' : I \to X = \gamma(2t)\mathbb{I}_{[0, \frac{1}{2}]} + \gamma'(2t - 1)\mathbb{I}_{[\frac{1}{2}, 1]} \). The resulting curve is still a loop with base \( x_0 \), obtained by following the two paths in succession at double speed.

Next, consider a constant curve \( e(t) = x_0 \). The products \( e \circ \gamma \) and \( \gamma \circ e \) are both strictly speaking different curves than \( \gamma \). Nevertheless, there exists a change of parameter, i.e. a bijection \( \varphi \) of \( I \) onto \( I \), such that \( \gamma = (e \circ \gamma)(\varphi) \) and \( \gamma = (e \circ \gamma)(\varphi^{-1}) \). In other words, \( e \circ \gamma \) and \( \gamma \circ e \) are both homotopical to \( \gamma \).

By the same argument, the two products of loops \( (f \circ g) \circ h \) and \( f \circ (g \circ h) \) only differ by a change of parameter, and are thus homotopical too.

It is slightly more elaborate to prove that for any loop \( \gamma(t) \), the loop \( \gamma(1-t) \) is an inverse modulo homotopy. It suffices to produce such homotopy explicitly: consider \( \forall t \in [0, 1] \) the path \( \gamma_t(s) = \gamma(s)\mathbb{I}_{s \in [0, 1-t]} + \gamma(1-t)\mathbb{I}_{s \in [1-t, 1]} \). Then defining \( \eta_t \) as the inverse path of \( \gamma_t \), a direct calculation shows that \( \gamma_t \circ \eta_t \) is a homotopy of loops connecting smoothly \( \gamma(s) \circ \gamma(1-s) \) to the constant path \( e \), and its inverse \( (\gamma_t \circ \eta_t)(1-s) \) connects smoothly \( \gamma(1-s) \circ \gamma(s) \) to \( e \). Hence we have explicitly built an inverse modulo homotopy.

It remains to prove that concatenation of loops preserves the homotopy class: indeed if \( f_t \) and \( g_t \) are homotopies connecting \( f_0 \) to \( f_1 \) and \( g_0 \) to \( g_1 \) respectively, then the concatenation of the two \( f_t \circ g_t \) (as functions of \( s \)) connects \( f_0 \circ g_0 \) to \( f_1 \circ g_1 \). So denoting by \( [f] \) the homotopy class of \( f \), it holds that \( [f] \circ [g] = [f \circ g] \) and therefore the operation lifts properly to the quotient set. By now we can safely call the concatenation of loops a product, as it is evident that it represent a group operation modulo homotopy:

**Definition 2.17.** The quotient set of loops with basepoint \( x_0 \) modulo homotopy equivalence is a group with respect to the concatenation operation, and is denoted \( \pi_1(X, x_0) \). We call it the Fundamental Group of \( X \) at basepoint \( x_0 \).

We may add as a final remark that the dependency on the choice of basepoint \( x_0 \) is not as relevant as it may appear: if the space \( X \) is path-connected, one can prove that there exists an isomorphism between \( \pi_1(X, x_0) \)
and \( \pi_1(X, x_1), \forall x_0, x_1 \in X \). Therefore, for path-connected spaces the fundamental group is unique up to isomorphism, and is sometimes denoted as simply \( \pi_1 X \).

**\( \pi_1 X \) in action**

While the construction of \( \pi_1(X) \) is relatively straightforward, it is not so simple to get a sense of its meaning. In particular, one may wonder under what circumstance a topological space has a non-trivial fundamental group: this means proving that a homotopy between two paths does not exist, or equivalently that it is necessary to break continuity to deform one path into another.

Clearly, the constant loop belongs to the fundamental group of any space: therefore, if \( X \) is path-connected \( \pi_1 X \) is trivial if and only if every loop can be smoothly deformed to the constant loop. Intuitively, this means that a space cannot have holes. It is easy to accept that for a space with a hole, loops that wind around the hole and loops that do not cannot be homotopical: in some specific sense, every hole in \( X \) adds a generator of group \( \pi_1 \). It is therefore not surprising that the fundamental group of the circle is, up to isomorphism, the group of the integers \( \mathbb{Z} \): its loops are, up to homotopy, parametrizations of the unit circumference, travelled clockwise or counterclockwise, at multiple speeds \( k \in \mathbb{Z} \). Its unit element is the constant loop at the (arbitrary) basepoint \( x_0 \), which clearly cannot be smoothly deformed into a path that winds around the circle. On the other hand, the fundamental group of the 2-ball is the trivial one, as any closed path is homotopic to the constant one.

It is useful to consider the fundamental group of some complexes (intended as topological spaces), and more importantly understand the dependency of \( \pi_1 \) from the complex structure. Since it is built onto maps from low-dimensional spaces into \( X \), namely loops \( I \to X \) and homotopies of them \( I \times I \to X \), it is not surprising that \( \pi_1 \) of a complex depends only on its low-dimensional structure. In fact, when \( X \) is a CW complex, \( \pi_1(X) \) only depends on its 2-skeleton. This is useful on the one hand for simplicity of understanding, but constitutes a serious limitation in the ability to tell higher-dimensional structures apart: for example, on the n-sphere it happens that

\[
\pi_1(S^n) = \pi_1(S^m) \quad \forall m, n \geq 2
\]

The fundamental group is essentially blind to high-dimensional features. A workaround to this is given by a higher-dimensional equivalent of the funda-
mental group: index 1 in $\pi_1$ indicates that the quotient is performed with respect to loops and homotopies of loops. One can compute it with respect to a more complex map equivalence, given by surfaces and their homotopies $I^2 \to X$ and $I^2 \times I \to X$, yielding $\pi_2(X)$, and in general the $n$-th homotopy group $\pi_n(X)$ computed through maps $I^n \to X$ and their homotopies $I^n \times I \to X$ (where $I^n$ denotes the n-cube). In this case, the above problem is solved: when $X$ is a CW complex, $\pi_n(X)$ only depends on the $n+1$ skeleton of $X$ and it holds

$$\pi_i(S^n) = 0 \quad \forall i < n , \quad \pi_n(S^n) \simeq \mathbb{Z}$$

This solution is hindered by a practical reason: computing higher-order homotopy groups is extremely complicated, even for such simple cases as the sphere. In general $\pi_i(S^n)$ for $i > n$ displays a surprisingly complex behaviour, and at present much remains unknown of these objects. This provides the main motivation for the development of the alternative concept of homology. We shall see in the following that, again, the homology group, denoted by $H_n(X)$, only depends on the $n+1$-skeleton when $X$ is a CW complex. Furthermore

$$\forall \ 1 \leq i \leq n \ H_i(S^n) \simeq \pi_i(S^n) \quad \text{but} \quad H_i(S^n) = 0 \quad \forall i > n$$
2.18 Simplicial Homology

We now come to the fundamental concept that underpins most of Topological Data Analysis. The first introduction of homology-like concepts dates back to the definition of Euler characteristic, the genus of a surface, and Poincaré’s studies of the topological properties of manifolds in dynamical systems. The modern concept of homology has been given a remarkably theoretical framing, essentially rooted in category theory, which provides a common ground for a number of different theories. All these variants, however, share the essential property that homology is a method to associate an algebraic object, sometimes a structure of algebraic objects, to other mathematical entities, and most notably topological spaces.

For the sake of the present work, we shall limit ourselves to introducing a homology theory that is especially tailored to apply to simplicial complexes, hence named simplicial homology. Consider a simplicial complex $X$, and suppose it is given an orientation. An orientation of a $k$-simplex is an equivalence class of orderings of the vertices modulo even permutations. Hence, given an ordering on the set of vertices, one can fix an orientation for every $k$-simplex in $X$, up to parity. Therefore, there exist only two orientations for each simplex.

Chain Complex

Consider an oriented simplicial complex, and the formal sum

$$\sum_i c_i \sigma_i , \quad c_i \in \mathbb{Z}$$

(2.18.1)

i.e. the space of linear combination of $k$-simplices with integer coefficients. The inverse of a simplex is intended as the simplex with the opposite orientation. The intuition behind this result is as follows: concatenating the same path in two opposite directions amounts to the identity element, that is not moving at all.

**Definition 2.19. K-Chain** We call the $k^{th}$ chain group $C_k$ of $X$ the free Abelian group on the set of its oriented $k$-simplices.

As per the insight given above, we can build ”strings” of $k$-simplices and cancel out occurrences with different orientation. Keeping in mind that the group is Abelian, we can reorder words so as to build consecutive ”paths”. Taking linear combinations of simplices as in (2.18.1) with coefficients in
the ring of the integers yields a module structure. We will show in the following that it is of paramount importance for computational reasons for this structure to be, instead, a vector space. Consider, this time, the linear combinations of the form (2.18.1) with coefficients in the finite field \( \mathbb{Z}_2 \). The field structure, namely the existence of the multiplicative inverse, entails that the \( k \)-chain with \( \mathbb{Z}_2 \) coefficients is a vector space, and as such can be given a basis from a linearly independent set of vectors.

**Boundary Operator**

Consider, again, the \( k \)-chain with integer coefficients. Let us define a boundary operator \( \partial_k \) as follows

**Definition 2.20.**

\[
\partial_k : C_k \to C_{k-1}
\]

\[
\partial_k : \sigma = [v_0, ..., v_k] \in C_k \mapsto \sum_{i=0}^{k} (-1)^i [v_0, ..., \hat{v_i}, ..., v_k]
\]

where \( \hat{v_i} \) denotes that the \( i \)-th component has been removed.

It is easy to see that \( \partial_k \) is a group homomorphism between \( C_k \) and \( C_{k-1} \). To understand why it is called boundary operator, let us first notice that it associates to elements of dimension \( k \) elements of the previous dimension, obtained by removing one vertex at a time. As such, its image is a linear combination of faces of \( \sigma \). Its action on low-dimensional simplices is illuminating:

**Example 2.21.** Consider simplices \( \{a, b\} \) and \( \{a, b, c\} \), oriented according to alphabetical order. The boundary of an edge is

\[
\partial_1 (\{a, b\}) = +\{b\} - \{a\}
\]

The boundary of a triangle, instead, is

\[
\partial_2 (\{a, b, c\}) = +\{b, c\} - \{a, c\} + \{a, b\}
\]

That is, edges \( ab \) and edge \( bc \) forward, and edge \( ac \) backwards, yielding the closed sequence \( a \to b \to c \to a \)
It is worth noting that
\[ \partial_1 \partial_2 (\{a, b, c\}) = \partial_1 \{b, c\} - \partial_1 \{a, c\} + \partial_1 \{a, b\} = \]
\[ = \{c\} - \{b\} - \{c\} + \{a\} + \{b\} - \{a\} = 0 \]

This is true in general

**Property 2.22.** The boundary operator is nilpotent

\[ \partial_k \partial_{k+1} = 0 \quad \forall k \quad (2.22.1) \]

This lemma has a rather deep meaning: the image of the boundary operator on a \( k \)-simplex is a closed cycle of \((k - 1)\)-simplices. These closed cycles are, in particular, borders of higher-dimensional simplices. Borders themselves form a subgroup of \( C_{k-1} \)

\[ B_k := \text{Im} \ \partial_{k+1} \subseteq C_k \]

Another statement of property (2.22.1) is

**Property 2.23.**

\[ \text{Im} \ \partial_{k+1} \subseteq \ker \partial_k \]

But what is \( \ker \partial_k \)? As we have just seen, the kernel certainly contains the borders of all the \((k + 1)\)-simplices in \( X \). But not all closed concatenations of \( k \)-simplices come from borders of \((k + 1)\)-simplices. Indeed, one can concatenate three edges that form a triangle, even if the corresponding triangular 2-simplex does not belong to \( X \). In general, \( \ker \partial_k \) is the subgroup of \( C_k \) formed of all possible closed cycles of \( k \)-simplices, be they borders or not.

\[ Z_k := \ker \partial_k \subseteq C_k \]

Hence we have a group inclusion

\[ B_k \subseteq Z_k \subseteq C_k \]

**Remark** Notice that the exact same line of reasoning works if we consider \( k \)-chains with coefficients in \( \mathbb{Z}_2 \), if we take care to redefine the border operator as
Definition 2.24.

\[ \partial_k : C_k \to C_{k-1} \]

\[ \partial_k : \sigma = [v_0, ..., v_k] \in C_k \mapsto \sum_{i=0}^{k} [v_0, ..., \hat{v}_i, .., v_k] \]

Indeed, in \( \mathbb{Z}_2 \) sum and subtraction coincide, \( + \sigma = -\sigma \) and \( 2\sigma = 0 \), so only simplices that are taken odd times are not zero. We can now proceed to the main definition of the section

**Definition 2.25. k-th Homology Group**

We call the \( k \)-th Homology Group \( H_k \) the quotient

\[ H_k := \frac{\ker \partial_k}{\text{Im} \partial_{k+1}} = \frac{Z_k}{B_k} \]

The concept of homology classes is the following: we identify with the null element every cycle that is a border of a higher-dimensional simplex, or that can be built as a linear combination of borders of simplices. Therefore, the only non-zero elements (more precisely, the only non-trivial homology classes) are represented by those cycles which do not come from borders. Two cycles are homologous (i.e. they belong to the same class in \( H_1 \), i.e. they are equal modulo borders) if they differ by a sum of null cycles, namely in dimension 1 if they differ by a triangulated path.

**Graded modules, vector spaces and basis of \( H_1 \)**

As we have introduced above, the \( k \)-chains groups, equipped with a linear combination with coefficients in an Abelian unitary ring (usually \( \mathbb{Z} \)), form a module structure. It is well-known that, if the ring is also a field, the module is in fact a vector space. So it holds that the group \( H_1 \) with finite field \( \mathbb{Z}_2 \) coefficient, quotient of two vector spaces, is itself a vector space; conversely, as a quotient of two modules, it is a module.

The structure of these algebraic objects can be made clearer by employing a known result.

**Definition 2.26. Graded Ring**

We call a graded ring \( R \) a ring for which there exists a direct sum decomposition into groups \( R_i \)

\[ R = \bigoplus_i R_i \]
which respects multiplication, i.e.

\[ R_h R_k \subseteq R_{h+k} \]

The typical example of a graded ring is the ring of polynomials, where the grading is given by the degree. Specifically, over \( R[x] \) it holds that \( R[x] = \oplus_n R \cdot x^n \), namely each polynomial over \( R \) can be expressed as a sum of homogeneous terms on degree \( n \) for \( n \in \mathbb{N} \).

The concept extends naturally to a module

**Definition 2.27. Graded Module**

We call a graded module \( M \) a module over a graded ring \( R \) such that

\[ M = \bigoplus_i M_i \]

as well and multiplication follows

\[ R_h M_k \subseteq M_{h+k} \]

Let us state the standard structure theorem for graded modules over a principal ideal domain.

**Property 2.28. Structure Theorem**

Let \( M \) be a graded module over a principal ideal domain \( A \)

\[
M \simeq \left( \bigoplus_{i=1}^{n} \Sigma a_i \right) \oplus \left( \bigoplus_{j=1}^{m} \Sigma \gamma_j \right) A
\]

The first term represents the free part, which is a vector space and is made of the generators of \( M \) that can yield and infinite number of elements. The second term is called the torsional part, and represents generators that can only generate a finite number of elements. Notice that, if the module is itself a vector space as in the case of the coefficient set being a field, the second term vanishes and the decomposition becomes simply a direct sum of orthogonal one-dimensional vector spaces, with as many entries as the dimension of \( M \).

**Definition 2.29. Betti numbers**

We call the \( k \)th Betti number \( \beta_k \) the rank of the free part of \( H_k \).

Notice that, if \( H_k \) is torsion-free, specifically if it is taken with coefficients in a field, \( \beta_k \) is simply the vector space dimension of \( H_k \).
2.30 Persistent Homology

Filtration of Simplicial Complexes

Let $X$ be a simplicial complex. We say $Y$ is a subcomplex of $X$ if it is a subset of $X$ which is still a simplicial complex.

**Definition 2.31. Filtration**

*Given a simplicial complex $X$, a filtration of a complex is a family $\mathcal{X}$ of subcomplexes*

$$0 = X^0 \subseteq X^1 \subseteq ... \subseteq X^m = X$$

From a metric point of view, one can build a filtration of complexes from a point cloud by applying a scheme such as Vietoris-Rips repeatedly onto the same data points, while increasing the threshold parameter $\epsilon$. This yields a nested family of simplicial complexes with growing connectivity, which entails a growing number of cycles as well as a growing number of borders.

**Property 2.32.** Let $\{\epsilon_i, 1 \leq i \leq m, \epsilon_i > 0\}$ be an increasing sequence of positive real numbers, $Q \subset \mathbb{R}^n$ a discrete set. The sequence

$$\{ VR_{\epsilon_i}(Q) \}_{i=1}^m$$

is a filtration of simplicial complex.

Persistence

Given a filtration of complexes $X^i$, the associated groups and operators are denoted as $\partial_k^i, Z_k^i, B_k^i, C_k^i$ and $H_k^i$. In this case, intuitively, beside having a ”vertical” mapping between decreasing-dimension chains of $k$-simplices, we also have a ”horizontal” inclusion between more and more refined complexes.

**Definition 2.33.** We call the $p$-persistent $k$th homology group of $X^i$ the group

$$H_k^{i,p} = \frac{Z_k^i}{B_k^{i+p} \cap Z_k^i}$$

In this case, intuitively, we quotient with respect to borders that remain such in the more refined complexes $p$ steps ahead of $i$. Hence the name persistence.

We define the *persistent Betti numbers* $\beta_k^{i,p}$ as the rank of the free subgroup of $H_k^{i,p}$. Again, if this is a vector space one can simply consider the rank of
the whole group. Obviously, since $B^i_k \subseteq Z^i_k$, if we set $p = 0$ we recover the usual homology group

$$H^{i,0}_k(X) = H_k(X^i)$$

In our setting, the structure of persistence is represented by the diagram in Figure 3, where we move horizontally through inclusion maps $\varphi^i_k$, and vertically via the border operator $\partial_k$. It is a known, but important, result, that every square in the diagram of Figure 3 commutes, i.e. it is the same to compute $\varphi^i_{k-1} \circ \partial_k$ or $\partial_k \circ \varphi^i_k$ ([4]).
3 Shortest Homology Basis

The Problem

Having given an overview of what homology is, we now proceed to stating the problem we wish to address ([6]). Assuming we place ourselves in the context of $H_1$ homology with $\mathbb{Z}_2$ coefficients, we know that all groups $Z_1$, $B_1$, $C_1$ and $H_1$ are finite-dimensional vector spaces, and therefore can be given a basis. We denote by $Z := \dim Z_1$ the dimension of the cycle group, $B := \dim B_1$ the dimension of the borders, and $g := \dim H_1$ is the first Betti number of a fixed simplicial complex $X$.

Notice that, for any simplicial complex $X$, there exists a bijection between the set of its $k$-simplices $X_k$ and its group of $k$-chains. So if we call $n_k = |X_k|$ the number of $k$-simplices in $X$, then $C_k$ is isomorphic to $(\mathbb{Z}_2)^{n_k}$, the vector space of column vectors such that the $j$th entry is 1 if the $j$th simplex belongs to the chain, and zero otherwise. Naturally a basis of $C_k$ is the canonical basis of $(\mathbb{Z}_2)^{n_k}$, where each one-hot vector identifies a $k$-simplex.

Since $Z_1$ is a vector space of dimension $Z$, there exist set(s)

$$\{c_1, ..., c_Z \} \ , \ c_i \in Z_1$$

which form a basis of $Z_1$. We call such a set a cycle basis.

Recalling the definition of homology, we call $[c]$ the homology class of cycle $c$. It is formed by all cycles in $Z_1$ which differ the one from the other by elements of $B_1$. In the light of what stated above, there must exist

$$\{[c_1], ..., [c_g] \} \ , \ [c_i] \in H_1$$

which form a basis of $H_1$. We call such a set a homology basis.

Assume now there exists a function that assigns a non-negative weight to each cycle $c$

$$\mu : Z_1 \rightarrow [0, +\infty)$$

One way to build a such function, and the one that will be employed in the applications, is to think of the 0- and 1-skeleton of $X$ as a weighted graph with non negative weights, $(V, E)$. Weights can be thought of as lengths. Then a 1-cycle can be represented as a vector of $\mathbb{Z}_2^{\lvert E \rvert}$, which is a basis of $Z_1$ as stated above. The function that assigns to each cycle the sum of the weights of the edges that belong to it is our weighting function for the following analysis.

We can now state our problem: given a simplicial complex $X$, find a set of
cycles \( \{c_1, ..., c_g\} \) such that

\[
\{c_1, ..., c_g\} = \arg\min \sum_{i=1}^{g} \mu(c_i)
\]

subject to \( \{[c_1], ..., [c_g]\} \) being a \textit{homology basis} of \( H_1(X) \).

In words, we aim at finding cycles of the shortest possible length, which generate the whole homology group.
3.1 Prior Work

In this section we examine known results from past works which turn out useful in our analysis. Let us begin by stating that the choice to restrict ourselves to the group $H_1$ is somehow unavoidable, as the problem of computing a minimal homology basis for dimension higher than one is proven to be NP-hard (Chen and Freedman, [7]).

**Property 3.2.** [7], Corollary 6.1

For homology of dimension 2 or higher, the minimal homology basis is NP-hard to approximate within any constant factor.

The meaning of this statement is two-fold: on the one hand, for $k > 1$ computing exactly a minimal basis of $H_k$ is NP-hard (we recall that a problem $Q$ is NP-hard if any problem in NP can be reduced to $Q$ in polynomial time, hence the intuition that $Q$ is at least as hard as any problem in NP), and even a polynomial-time algorithm that computes an approximate solution which differs from the exact one by a constant multiplicative factor does not exist. The class of problems with polynomial-time, constant factor approximation is usually called APX. Then an alternative statement of the above theorem is that computing a minimal homology basis in dimension 2 or higher does not belong to APX.

As a consequence, we can shift our focus onto simplicial complexes of dimension at most two, so as to distinguish 1-cycles which are borders of triangles from those which are not. It therefore appears only natural to examine graph theory results, as our base structure is essentially a decorated graph, possessing some cycles which are declared to be null, namely those coming from borders of the 2-skeleton, and other which effectively contribute with a non-trivial generator to the homology group.

The first algorithm to find a minimal cycle basis (MCB) of a weighted graph dates back to 1987, with the work of Horton [18], who provided a $O(m^3 n)$ running-time procedure for a graph with $m$ nodes and $n$ edges. His approach involved generating a superset of cycles guaranteed to contain a MCB, by computing for every possible pair of source node and edge a cycle generated by shortest path, and then reducing the obtained set by Gaussian elimination.

More recently, de Pina in [8] improved this result by applying a different approach, which yields a $O(m^3 + mn^2 \log n)$ complexity. In his case, a spanning tree $T$ is computed over graph $G$, and the set of edges in $G \setminus T$ is employed
to represent each cycle. Then the MCB is obtained by representing cycles as vectors of $\{0, 1\}^\mathbb{Z}$, maintaining a basis of the orthogonal subspace to the set of MCB, and at each step computing the shortest cycle which is linearly independent of the previous ones by enforcing a non-zero scalar product with the orthogonal basis.

Both these algorithms were later improved by applying fast matrix multiplication, the most recent incarnation of which is described in [20], [21], which computes the product of two $n \times n$ matrices in $O(n^\omega)$ steps with $\omega < 2.376$. Yet more recently, Mehlhorn et al. ([9]) found a further improvement of de Pina’s scheme by imposing a relaxed version of the maintenance of a basis of the orthogonal complement while the cycle basis is extended. Their method presents a recursive approach which extends both the cycles and the support vectors at the same time while maintaining relative orthogonality. It results in a time complexity of $O(m^2 n + mn^2 \log n)$.

The computation of the MCB had yet to be lifted to the context of simplicial homology, where one has to deal with the added complexity of having to generate a basis of a smaller space than $\mathbb{Z}_1$, as a consequence of different cycles being identified by the border equivalence. On the other hand, if exploited correctly, the smaller dimension can represent a computational advantage. The recent work of Dey et al. ([5], [6]) moves in this direction, combining the recursive approach of [9] with a cohomological technique called simplex annotation [10] which allows for efficient computation of the homology basis.
3.3 Cohomology and Simplex Annotation. Homology Basis.

As we mentioned above, the problem we face has a different character than those addressed in the previous section in that we not only search for a cycle basis, but require the minimal set to form a homology basis, hence taking into account the border equivalence. Notice that in general it is not straightforward to obtain a solution of our problem from a solution of MCB, since the quotient alters the structure of the vector space, modifying linear dependence between cycles. In particular, our solution is in general not a subset of the MCB.

A different approach to the problem is proposed in [10] and relies on a concept arising from cohomology theory, called simplex annotation. Let us define, for the homology group \( H_k(X) \) of simplicial complex \( X \), with rank \( g \)

**Definition 3.4. Simplex Annotation [10]**

An annotation for a \( k \)-simplex is a function from the \( k \)-skeleton into 0-1 vectors of length \( g \)

\[
a : X_k \rightarrow (\mathbb{Z}_2)^g
\]

such that any two \( k \)-cycles \( c_1 \) and \( c_2 \) are homologous if and only if

\[
\sum_{\sigma \in c_1} a(\sigma) = \sum_{\sigma \in c_2} a(\sigma)
\]

The above sum is referred to as the annotation of the cycle \( c_i \).

Restricting ourselves to the one-dimensional case, we can compute the annotation of the edges in \( X_1 \) so that two cycles are homologous iff their annotation vectors coincide. As such, given any cycle, its annotation provides a coordinate representation of its homology class in a given basis. Different annotating function refer to different choices of a basis on \( H_1(X) \).

Regarding the existence of such a function, it is worth noting what follows ([10]): if we denote by \( \{\phi_j\}_{j=1}^g \) the cocycles whose equivalence classes are generators of the cohomology group \( H^1(X) \), then an annotation can be obtained by assigning

\[
a : \sigma \mapsto (\phi_1(\sigma), ..., \phi_g(\sigma))
\]

The procedure to obtain this coordinate representation of cycles is roughly as follows ([10]):
• First, a basis of the cycles group is obtained.

• Then, the cycle basis is reduced to obtain a homology basis. This step naturally requires knowledge of the border structure, which shall be expressed as a matrix representation of the border operator $\partial_2$ in the given basis.

• Finally, every cycle in $Z_1$ can be assigned to its homology class, which has a natural vector representation in the computed basis.

Step one relies on the computation of a spanning tree of graph $G = (X_0, X_1)$, which we call $T$. Consider the difference between the edges of $G$ and those of the spanning tree. These are called sentinel edges. The cycle obtained in the spanning tree $T$ by adding each sentinel edge is called a sentinel cycle.

**Property 3.5.** [10] The set of sentinel cycles is known to form a basis of $Z_1$.

Hence we can express any cycle in $Z_1$ as a linear superposition of sentinel cycles, each identified by its sentinel edge. Recalling we deal with $\mathbb{Z}_2$ coefficients, any $z \in Z_1$ coincides with the sum of the sentinel cycles for every sentinel edge that belongs to $z$ (notice that, for $z$ to be a cycle, at least one of its edges must be a sentinel).

For step two, we need as mentioned a description of the border structure. Recall that the border operator $\partial_2$ is a group homomorphism between $C_2$ and $C_1$ that associates 2-simplices (triangles) to their border 1-cycle. As such, given the vector description of 1-cycles induced by the pair spanning tree - sentinel edges, operator $\partial_2$ can be cast in matrix form by writing side by side the vectors describing the boundary cycles, for each 2-simplex in $X$. The resulting matrix has as many rows as the dimension of $Z_1$, and as many columns as the cardinality of $X_2$.

To its right, we add all column vectors of the cycles of $Z_1$, as represented in the sentinel basis (let us call this submatrix $S$). Recall that all said vectors have $\mathbb{Z}_2$ elements. We obtain a matrix of the form

$$[\partial_2 \mid S]$$

The next step allows us to obtain a suitable homology basis: we compute the so-called earliest basis of the above matrix, i.e. we proceed by columns, from left to right, eliminating any column that can be expressed as a linear
combination of those that precede it. The resulting matrix has full rank, and is of the form

\[
[B \mid H]
\]

where submatrix \( B \) of columns which derived from \( \partial_2 \) is a basis of the border group \( B_1 := \text{Im} \partial_2 \) and submatrix \( H \) is a basis of \( H_1 \). In particular the number of columns of the two submatrices give us \( \text{dim} \ B_1 \) and \( g \), respectively. Finally, we have to express each sentinel cycle in the basis of \( [B \mid H] \). This notoriously amounts to solving a linear system, in our case in \( \mathbb{Z}_2 \). If we solve for \( \tilde{z} \)

\[
[B \mid H] \tilde{z} = z
\]

where \( z \) is the vector representation of a cycle in the sentinel basis, its last \( g \) entries form the annotation of cycle \( z \), i.e. the homology representation \( z_h \) without the cyclic components deriving from borders.

If we consider the full matrix of cycles \( Z \), one can globally solve system

\[
[B \mid H] \tilde{Z} = Z
\]

The last \( g \) rows of \( \tilde{Z} \) form the annotation vectors of the sentinel cycles. Then each sentinel edge can be given the annotation of the sentinel cycle it generates. All other edges are given annotation \( 0 \in (\mathbb{Z}_2)^g \).
3.6 Methodology

In the light of the results mentioned in the previous sections, we now set forth to tackle the optimization part of our problem: i.e., finding a minimal basis. In the recent work of Dey [6], the approach of Mehlhorn [9], which itself improved on de Pina’s [8], was combined with the annotation technique [10] to yield a $O(n^3)$ worst-case complexity in the cardinality of the complex. The method proceeds as follows: first, a candidate set of cycles is generated by the shortest path tree method. For each node $p$, we compute the SPT rooted in $p$ and keep track of the non-tree edges. Each non-tree edge identifies a cycle with respect to vertex $p$. The union of all cycles for all vertices $p \in X_0$, which we call $G$, is guaranteed to contain a shortest homology basis ([5], [6], [19]).

Next, we move on to the annotation of edges to obtain a homology basis. As described above, we must generate a basis of cycles by means of a spanning tree and its sentinel edges. Notice that the SPT computed at the previous step can be employed for the case, taking care to consider a root node for each connected component of $X$, hence speeding the computation. Given our vector representation of cycles in the sentinel edge basis, we can form the matrix representation of the border operator $\partial_2$ by placing side by side the vectors describing borders, for each 2-simplex in $X$. Next to it, we can place our cycle basis, and operate the left-to-right reduction, as described above ([10]). From this step, we obtain the Betti number of $X$, and solving the following linear system, a basis of $H_1(X)$ and the annotation vectors.

Finally, we proceed to produce the $j$-th minimal basis vector of $H_1$ for $j = 1, ..., g$ as follows:

1. Initialize a basis of $(\mathbb{Z}_2)^g$, for example with the canonical basis. We call these support vectors $S_i$.

2. At step $j$, generate the set of cycles, subset of $G$, whose annotation has $\mathbb{Z}_2$-scalar product equal to 1 with $S_j$. These are cycles which are not orthogonal to $S_j$. Of these, pick the shortest one in length, $c_j$. Add this vector to the minimal basis.

3. Update the support vectors $S_{j+1}, ..., S_g$ so as to guarantee that they still form a basis of $\text{Span}\{c_1, ..., c_j\}^\perp$, and that $c_j$ and $S_{j+1}$ are orthogonal.

Repeat steps 2 and 3 for $j = 1, ..., g$. Then
Property 3.7. [6],[5]

The resulting set of cycles \{c_1, ..., c_g\} is the minimal homology basis of \(H_1(X)\).

Node Labeling

The computation of the subset of cycles which are non-orthogonal requires to compute a possibly large number of scalar products. There exists a technique called node labelling to expedite the task at its best possible performance ([6]). It is in essence an application of linearity of both scalar product and annotation: for each support vector \(S_i\) and calling \(m\) the cardinality of \(X_0\), one can build a \(m \times m\) matrix of labels of nodes such that the entry \((h, k)\) is the \(\mathbb{Z}_2\) sum of scalar products of \(S_i\) with the annotation of the edges that connect \(h\) to \(k\), in the shortest path tree rooted in \(h\). Hence, one can follow the SPT from the root to the leaves, computing a scalar product for every new edge that one encounters and summing it to the label of its parent node. Therefore, all it takes to check orthogonality with respect to \(S_i\) of a cycle described by a pair \((p, (v, w))\) (where \(p\) is the root node and \((v, w)\) are the extremes of a sentinel edge) is to sum the labels of index \((p, v)\) and \((p, w)\) with the scalar product between \(S_i\) and the annotation of edge \((v, w)\).
3.8 Implementation

We now give a brief description of the Python 3 code we have implemented for the algorithm described above.

Geometry

For testing applications, point clouds were usually random-sampled. Then the 1-skeleton and its consequent clique simplicial complex were generated by computing pairwise distances. In real-world applications, data is usually given as weighted adjacency matrices, in the form of lists of triplets (node, node, weight). This gives the 1-skeleton in a precomputed manner, and one is left with the task of identifying triangles to tile in order to generate the $\partial_2$ operator matrix (2.8).

```python
def getTriangles(A):
    '''
    returns set of triangles in a graph given its adjacency matrix
    '''
    n = len(A)
    tri = []
    for vertex in range(n):
        vList = np.nonzero(A[vertex, vertex:])
        vList = [x for x in vList[1]]
        vList = [i + vertex for i in vList] # list of vertices adjacent to vertex
        for i in range(len(vList) -1):
            for j in range(i+1, len(vList)):
                if A[vList[i], vList[j]] > 0:
                    tri.append([vertex, vList[i], vList[j]])
        return tri

def getD2(A, edgesList):
    '''
    returns 2-boundary matrix of the clique complex, given the adjacency matrix of the graph
    '''
    triangles = getTriangles(A)
    # print(triangles)

    d2 = []
    n = len(edgesList)
    # print(n)
```
for row in triangles:
    newTriangle = [0 for i in range(n)]
    newTriangle[edgesList.index((row[0],row[1]))] = 1
    newTriangle[edgesList.index((row[0],row[2]))] = 1
    newTriangle[edgesList.index((row[1],row[2]))] = 1
    d2.append(newTriangle)

return d2

def points2adj(P, epsilon):
    '''
given a set of points and a threshold returns the weight
and adjacency matrix of the corresponding graph
points is a list where each element is the coordinates of
the point
    '''

    def dist(p1,p2):
        '''
euclidean distance
p1 and p2 need to be list of the same length
        '''
        return math.sqrt(sum([(p1[i]-p2[i])**2 for i in range(len(p1))]))

    n = len(P)
    W=np.zeros((n,n))
    for i in range(n):
        for j in range(i,n):
            W[i,j]= dist(P[i],P[j])
    W = W + W.T # weight matrix
    W[W >= epsilon] = 0
    A = (W >0).astype(int)
    return W , A

def getEL(A):
    edgesList = []
    for i in range(len(A)):
        for j in range(i,len(A)):
            if A[i,j] > 0:
                edgesList.append([i,j])
    return edgesList
Listing 1: Code for the generation of the simplicial structure

In the applications, as is standard procedure in topological data analysis, we have built filtrations of simplicial complexes to analyze how the homological structure changes at different length scales. This requires computing subsets of the full complex $X$, by restricting its 1-skeleton to those edges having length smaller than a given threshold $\epsilon$, and tiling the 2-skeleton accordingly via the definition of clique complex (2.6, 2.32).

```python
def filterMatrix(W, epsilon):
    Wb = np.matrix(W)
    Wb[Wb >= epsilon] = 0
    return Wb
```

Listing 2: Code for the filtration of the 1-skeleton

**Shortest Path Tree**

The following describes the computation of the shortest path tree. Its generation is necessary to obtain a candidate set of cycles among which to find the optimal ones ([6], Proposition 3.1).

```python
def computeShortestPath(self, withDistances=False):
    (self.dist, self.pred) = csgraph.shortest_path(self.Weights.tocsr(), directed=False, return_predecessors=True)

    # Aggiungiamo il calcolo dei successori. Struttura di lista^3
    for source in range(self.NVert): # per ogni sorgente
        link = self.pred[source,:] # predecessori
        forward = [[] for i in range(self.NVert)] # lista di liste vuote lunga NVert
        for (foll, prev) in enumerate(link):
            if (prev != -9999): # se è connesso e non è source
                forward[prev].append(foll)
        self.Followers.append(forward)

def shortestPathTree(self, source, withD=False, justEdges=False):
    # ...
```
if (type(source) is not int or source < 0 or source >= self.NVert):
    raise Exception('Indice del vertice non valido!')

if withD:
    if (self.dist is None or self.pred is None):
        _ = self.computeShortestPath(withD)
else:
    if self.pred is None:
        _ = self.computeShortestPath(withD)

link = np.array(self.pred[source,:])

data = []
rows = []
col = []
# ...
if withD:
    data = np.array(data, dtype=float)
else:
    data = np.array(data, dtype=int)
rows = np.array(rows)
col = np.array(col)
SPT = scipy.sparse.coo_matrix((data,(rows,col)),
    shape=(self.NVert,self.NVert))

rw, cl, _ = scipy.sparse.find(self.Weights)

edgeSPT = list(zip(list(rows),list(col)))
edgeW = list(zip(list(rw),list(cl)))

# Affinchè il metodo funzioni, è necessario che il
# nodo source non abbia -9999 per distinguere da
# quelli davvero disconnessi
link[source] = source  # creiamo un "finto" self-loop
edgeW = [ x for x in edgeW if link[x[0]] != -9999 ]

NTE = list(set(edgeW) - set(edgeSPT)) # differenza
# insiemistica tra i set di edges

NTE = [ x for x in NTE if (x[0]<x[1]) ]

if justEdges:
    return NTE
else:
Listing 3: Code for the computation of the SPT

Basis of cycles
This piece of code computes a basis of the group of cycles from the sentinel edge description ([10], Proposition 2)

```python
def getCycleBase(self):
    if (self.Sentinel is None):
        _ = self.spanningTree()

    SC = [] # lista di sentinel cycles
    for e in self.Sentinel:
        SC.append(self.closeCycle(e))

    return SC
```

Listing 4: Code for the generation of a cycle basis

Annotation
The following piece of code computes the annotation matrix of sentinel edges ([10])

```python
def getAnnotation(self,d2,Z):
    def low(col):
        l=-1;
        for i in range(len(col)):
            if col[i]>0:
                l=i
        return l

    lowSet = {}; #dictionary with low indexes and relative rows
    if len(d2) == 0: # controlla se d2 è vuoto!
        dimB1 = 0
        i = 0
    else:
        i=0;
        while i != len(d2):
            lowRowi=low(d2[i])
```

37
```python
while lowRowi in lowSet.keys():
    d2[i] = (d2[i] + d2[lowSet[lowRowi]]) % 2
    lowRowi = low(d2[i])
if lowRowi > -1:
    lowSet[lowRowi] = i
    i = i + 1
else:
    d2 = np.delete(d2, (i), axis=0)
    dimB1 = i  # dimensione dello spazio dei bordi
if (dimB1 != 0):
    Z = np.concatenate((d2, Z), axis=0)
else:
    pass
totRow = len(Z)
reductionMatrix = np.identity(totRow, dtype=int)
Id = np.identity(totRow, dtype=int)
elementsToDelete = []

while i != totRow:
    lowRowi = low(Z[i])
    while lowRowi in lowSet.keys():
        Z[i] = (Z[i] + Z[lowSet[lowRowi]]) % 2
        reductionMatrix[i] = (reductionMatrix[i] + Id[lowSet[lowRowi]]) % 2
        lowRowi = low(Z[i])
    if lowRowi > -1:
        lowSet[lowRowi] = i
        i = i + 1
    else:
        elementsToDelete.append(i)
        i = i + 1
reductionMatrix = np.delete(reductionMatrix, elementsToDelete, axis=1)
reductionMatrix = np.delete(reductionMatrix, range(dimB1), axis=1)
A = np.delete(reductionMatrix, range(dimB1), axis=0)

self.dimB1 = dimB1
self.dimZ1 = np.shape(A)[0]
self.dimH1 = np.shape(A)[1]
self.AnnDict = {}
```
for i,e in enumerate(self.Sentinel):
    ann = self.Ann[:,i]
    self.AnnDict[e] = ann

return (self.Ann, self.dimB1, self.dimZ1, self.dimH1)

Listing 5: Code to compute the annotation

Labeling

The following piece of code computes the labels of nodes with respect to support vector Sup ([6], Base case).

def innerProd(self,S,C):
    if (len(S) != len(C)):
        raise ValueError("Dimensioni non compatibili!")

    return np.dot( np.array(S).transpose() , np.array(C) ) % 2

def computeLabels(self, Sup):
    if (self.Ann is None):
        raise ValueError("Non è stata calcolata l’annotazione degli edges!")

    for p in range(self.NVert):
        forward = self.Followers[p]
        self.Labels[p,p] = 0
        driver = []  # lista di push/pop DI TUPLE (PREVIOUS,FOLLOWER)
        add = [ (p,x) for x in forward[p] ]
        driver.extend(add)
        while ( len(driver) != 0):
            (prev,foll) = driver.pop(0)
            lab = self.Labels[p,prev]
            edge = (prev, foll) if prev<foll else (foll, prev)
            try:
                ann = self.AnnDict[edge]
            except KeyError:
                self.Labels[p,foll] = lab
            else:
                self.Labels[p,foll] = (lab + self.
                    innerProd(Sup,ann))%2
                add = [ (foll, x) for x in forward[foll] ]
                driver.extend(add)
Listing 6: Code to compute the labeling with respect to support vector Sup

**Shortest New Cycle**

The following piece of code finds the shortest cycle that is non-orthogonal to support vector Sup ([6], Claim 3.1).

```python
def findShortestNonOrtho(self, Sup, allDraws=False):
    _ = self.computeLabels(Sup)

    def checkOrt(s, e):
        try:
            ann = self.AnnDict[e]
        except KeyError:
            lab = 0
        else:
            lab = self.innerProd(Sup, ann)  # calcola il prodotto
        return (self.Labels[s, e[0]] + self.Labels[s, e[1]] + lab) % 2

    candidates = []
    candidates = [x for x in self.NTE if checkOrt(x[0], x[1]) == 1]
    candidates = [self.closeCycle(x[1], force=x[0]) for x in candidates]

    def lenCycle(C):
        return np.dot(self.WEdges, C)

    candidates = list(zip(candidates, map(lenCycle, candidates)))

    if allDraws:
        shortest = candidates[0][1]
        minList = []
        for x in candidates:
            if x[1] < shortest:
                minList = [x]
                shortest = x[1]
            elif x[1] == shortest:
                minList.append(x)
            else:
                pass
        return minList
    else:
```

40
min_len = lambda x,y : x if x[1] <= y[1] else y
minCycle = functools.reduce( min_len , candidates )
return minCycle

Listing 7: Code to find the new shortest basis vector with respect to support vector Sup

Update of support vectors
The following piece of code updates the basis of the orthogonal complement of the first basis vectors ([6], Theorem 3.1)

def updateSup(self, newC, index):
    self.SHB.append( newC ) # aggiungi il nuovo ciclo
    sup_i = self.Support[index] # support vector del nuovo ciclo
    AnnNewC = self.cycleAnnotation(newC)
    for j in range(index+1,self.dimH1):
        sup_j = self.Support[j]
        self.Support[j] = (sup_j + sup_i * self.innerProd( AnnNewC , sup_j )) % 2

Listing 8: Code to update the support vectors.
4 Results and Applications

In this section we run the algorithm we have implemented. A series of tests will be shown, after which we move on to some actual data analysis on a real-world dataset.

4.1 Testing

Erdős-Rényi

We begin by testing the code on some random-generated Erdős-Rényi graph, through the python library networkx.

```python
import numpy as np
import scipy
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

def getRandomGraph(nVert,p):
    G = nx.erdos_renyi_graph(nVert,p)
    flag = nx.is_connected(G)

    A = nx.adjacency_matrix(G).todense()
    return (G,A,flag)

(G,A,f) = getRandomGraph(10,0.15)
xn.draw(G)
```

Listing 9: Networkx Example.

Plotting the output we obtain as in figure 4:
In this case, the borders are chosen randomly among cycles. Hence the simplicial complex this graph belongs to is not clique. Indeed, we highlight in blue an arbitrary choice of cycles to form borders (Figure 5).

```
NBorders = 2
NBorders = min(NBorders, len(cycles))
indexes = random.sample(xrange(len(cycles)), NBorders)
# ...
d2 = []
for i in indexes:
    d2.append( cycles[i] )
# ...
xn.draw(Gn, pos=Gn_pos, edge_color=OrderedColorB, node_size =50)
```

Listing 10: Choice of borders.
Figure 5: The (arbitrary) borders highlighted in blue. Notice this complex is, therefore, not clique.

Let us run the algorithm on matrix $A$ representing this graph. Highlighted in yellow, the algorithm evidences that all the remaining cycles do belong to the shortest homology basis (Figure 6). On the one hand this confirms the correctness of the output for what concerns homology; on the other, this example is clearly too small to suffice to demonstrate the algorithm is able to identify the shortest possible basis.

```python
(An, B1, Z1, H1) = G.getAnnotation(d2, cycles)

for i in xrange(H1):
    (mincycle, length) = G.findShortestNonOrtho( G.Support[i] )
    G.updateSup(mincycle, i)
    print np.matrix(G.Support).transpose()

nx.draw(Gn, pos=Gn_pos, edge_color=OrderedColorH, node_size = 50)
```

Listing 11: Shortest Homology Basis
Random Point Sampling

As a second testing step, we proceed to generate a random point cloud in the $\mathbb{R}^2$ plane. Then, we compute pairwise distances between points to generate the 1-skeleton, and tile the cliques accordingly (2.8).

```python
def sampleCircle(x0, y0, r, n, noise):
    P = []
    for i in range(n):
        theta = random() * 2 * pi
        R = r + noise * random()
        xp = x0 + R * cos(theta)
        yp = y0 + R * sin(theta)
        P.append([xp, yp])
    return P

P = sampleCircle(10, 1, 8, 90, 3)
P += sampleCircle(-10, 1, 8, 90, 3)
P += sampleCircle(0, 10, 8, 90, 3)
```

# ...

G = SimplexGraph(nVert,W)
cycles = G.getCycleBase()
(An, B1, Z1, H1) = G.getAnnotation(d2,cycles)
sup = np.eye( H1, dtype = int )
Sup = []
for i in range(H1):
    Sup.append( sup[:,i] )

G.Support = Sup
for i in range(H1):
    (mincycle, length) = G.findShortestNonOrtho( G.Support[i] )
    G.updateSup(mincycle, i)

SHB = np.matrix(G.SHB).transpose()

Listing 12: Shortest Homology Basis of a random sampling of nodes.

This larger example exhibits the desired behaviour: the algorithm identifies hole-like features in the dataset, and is able to localize them tightly, yielding the shortest possible cycle as a generator for each class.

Output:

Figure 7: Example of SHB on a random sampling of 180 nodes.
Figure 8: Example of SHB on a random sampling of 270 nodes.

**Filtration of Random Samplings**

Let us now move to a typical application in Topological Data Analysis: i.e., let us repeat the previous analysis on a filtration of simplicial complexes $\mathcal{V}_c$ built from the same set of nodes, increasing the threshold that defines the 1-skeleton, and consequently the cycles (2.32).

```python
def filterMatrix(W, epsilon):
    Wb = np.matrix(W)
    Wb[Wb >= epsilon] = 0
    return Wb

# ...

P = sampleCircle(10,1,8,45,3)
P += sampleCircle(-10,1,8,45,3)
P += sampleCircle(0,10,8,45,3)

# ...

def filtration_pipeline(W, epsList):
    nVert = len(W)
    SHB = []
    Ws = []
```
maximal = SimplexGraph.getEdgeList(W)
for eps in epsList:
    Wstep = filterMatrix(W,eps)
    Ws.append(Wstep)
    G = SimplexGraph(nVert,Wstep, maximal)
    cycles = G.getCycleBase()
    d2 = getD2(Wstep, maximal[0])
    (An, B1, Z1, H1) = G.getAnnotation(d2,cycles)
    sup = np.eye(H1, dtype = int)
    Sup = []
    for i in range(H1):
        Sup.append(sup[:,i])
    G.Support = Sup
    for i in range(H1):
        (mincycle, length) = G.findShortestNonOrtho(G.Support[i])
        G.updateSup(mincycle, i)
    SHB.append(np.matrix(G.SHB).transpose())
return SHB, Ws, maximal

epsList = [3.0,4.0,5.0,6.0]
SHB,Ws,maximal = filtration_pipeline(W,epsList)

Listing 13: Shortest Homology Basis for a filtered complex.

Step 0:
Figure 9: The maximal simplicial complex, unfiltered.

Step 1, $\epsilon = 3.0$:
Figure 10: The filtered complex with threshold $\epsilon = 3.0$. We notice that the only cycles are the cliques, which are borders. Hence the homology group is trivial and has no representatives.

Step 2, $\epsilon = 4.0$:
Figure 11: The filtered complex with threshold $\epsilon = 4.0$. Longer edges begin to appear, but they are as yet unable to close macroscopic cycles; the only cycles are still cliques, and the homology group is still trivial.

Step 3, $\epsilon = 5.0$:
Figure 12: The filtered complex with threshold $\epsilon = 5.0$. At last, "big" cycles are closed and contribute to $H_1$ with non-trivial generators. We may notice that the homology basis is indeed localized, i.e. the cycles are the shortest possible.

Step 4, $\epsilon = 6.0$:
Figure 13: The filtered complex with threshold $\epsilon = 6.0$. At this step we may notice that increasing the connectivity can alter the structure in complex ways. Specifically, in this case the Betti number has remained unaltered, but the homology classes are clearly different. The smaller cycle at the previous step has been \textit{tiled out}, while a new cycle was born due to the appearance of a new edge splitting the large bottom-right hole.
4.2 C. Elegans

Finally, we have applied the algorithm to a real dataset. The data in our possession concerns time-averages of correlations between neuron firings in the nematode worm *Caenorhabditis elegans*. This organism was the first multicellular, albeit small, species whose genome was entirely sequenced, and it features another interesting peculiarity: every single individual in this species has the exact same nervous system, composed of 306 neurons. It therefore represents the model organism for the study of connectomes, i.e. the detailed mapping of connections between single neurons.

**Dataset**

The dataset consists of an edgelist describing the intensity of correlation between pairs of neurons, intended as graph nodes. In principle, this could yield a complete graph $K_{306}$. In practice, this is not the case as several correlation pairs are virtually zero, nonetheless the connectivity remains remarkably high. As a consequence, it is practically impossible to give an understandable graphical representation of the dataset; some subsets extracted from the whole aggregate are shown in Figures 15 and 16.

In order to perform a persistence analysis, it is useful to distinguish between the different values of the edge weights. As evinced in Figure 17, there exist 31 different weights for the total 2148 edges, hence to perform a single step persistence analysis we shall need to run 31 instances of the shortest homology basis finder. One must notice, however, that the edge weights represent correlations between neurons. Therefore, in order for our analysis to be meaningful (i.e. represent ”distances”) we need, as a preliminary step, to invert the value of weights.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>216 217</td>
<td>1.0</td>
</tr>
<tr>
<td>216 152</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Listing 14: An excerpt from the edge-list description of the dataset.

```python
data = np.loadtxt("celegans_weighted_undirected.edges")
v1 = data[:,0]
v1 = np.array(v1, dtype=int)
v2 = data[:,1]
v2 = np.array(v2, dtype=int)
weights = data[:,2]
wmax = max(weights)
weights = [float(wmax)/x for x in weights]
weights = np.array(weights, dtype=float)
FullEpsList = list(set(weights))
print("Min: ", min(weights), ", Max: ", max(weights), ", Total: ", len(weights), ", Different: ", len(set(weights)))
plt.hist(weights, range(int(min(weights)), int(max(weights))))
>
Numero vertici: 306
Min: 1.0 Max: 70.0 Total: 2148 Different: 31
```

Listing 15: An overview of the input dataset.
Figure 15: A subset of the input dataset.

Figure 16: A subset of the input dataset.

56
Figure 17: The histogram distribution of distance values. One can see there exist 31 different values for the 2148 edges.

Analysis

We now have a graph of 306 nodes and 2148 positively-weighted edges, plus a set of 31 threshold values to generate a single step filtration. Our objective is to analyze the behaviour of the homology group as the threshold increases. This example requires quite some computational effort in order to obtain a result. This is why a multithreaded approach was chosen: in order to reduce the total time requirement, different steps of the filtration were assigned to different cores of an off-the-shelf workstation. This introduces some complexity for what concerns coordinating the instances and some memory overhead to keep the outputs comparable, as is visible in the slightly more elaborate code in the listings below, but at the same time provides a substantial speed-up. The whole analysis ran for an approximate four and a half hours.

```python
def parallel_pipeline(eps):
    W = GlobalOptions['Matrix']
    nVert = len(W)
    SHB = []
    Ws = []
```
res = {}
maximal = SimplexGraph.getEdgeList(W)
stats = []
Wstep = filterMatrix(W,eps)
Ws.append(Wstep)

G = SimplexGraph(nVert,Wstep, maximal)
cycles = G.getCycleBase()
stats.append('NVert = ' + str(G.NVert) + ' NEdges = ' + str(G.NEdges))
stats.append('Filtration Eps = ' + str(eps))
stats.append('Keep all Draws = ' + str(GlobalOptions['Draws']))
d2 = getD2(Wstep, maximal[0])
(An, B1, Z1, H1) = G.getAnnotation(d2, cycles)
stats.append('H1= ' + str(H1) + ' B1= ' + str(B1) + ' Z1= ' + str(Z1))
sup = np.eye(H1, dtype=int)
Sup = []
for i in range(H1):
    Sup.append(sup[:,i])
G.Support = Sup
G.fixNTE()
t1 = time.time()
for i in range(H1):
    if GlobalOptions['Draws']:
        listMin = G.findShortestNonOrtho(G.Support[i], allDraws=True)
        mincycle = listMin[0][0]
        SHB.append(listMin)
    else:
        (mincycle, length) = G.findShortestNonOrtho(G.Support[i], allDraws=False)
        G.updateSup(mincycle, i)
if GlobalOptions['Draws']:
    res['SHB'] = SHB
else:
    res['SHB'] = np.matrix(G.SHB).transpose()
res['Draws'] = GlobalOptions['Draws']
res['Filtration Eps'] = eps
if GlobalOptions['ReturnMatrix']:
    res['Filtered Matrix'] = Ws
else:
    res['Filtered Matrix'] = None
if GlobalOptions["ReturnMaximal"]:
    res["Max"] = maximal
else:
    res["Max"] = None
    res["stats"] = stats

return res

Listing 16: The parallel pipeline to analyze the filtration.

file_stream = open(filename,'wb')
FullThreshList = [ x + 0.05 for x in FullEpsList ]
epsList = FullThreshList
cut = 306
Wn = W[0:cut,0:cut]
GlobalOptions["Matrix"] = Wn

pool = Pool(processes=4)
job = pool.map_async(parallel_pipeline, epsList)
result = job.get()

cPickle.dump(result, file_stream)
file_stream.close()

Listing 17: The multithreaded instance of the C. Elegans analysis.

As mentioned above, it is hardly useful to give a graphical representation of the resulting homology basis. The reason for this is twofold:

- The size and the sheer connectivity of the graph make it nearly impossible to get any sense of the underlying structure.

- The input data is intrinsically non-geometrical: as said, edge weights represent statistical correlations, and as such do not induce a metric on the graph, in that they do not obey any triangular inequality. Therefore, it is in general impossible to obtain a geometric realization of data points which respect the pairwise distance relations defined by the edge weights. At best one can dispose nodes according to a force-directed layout, giving up on the concept of distance while still obtaining a cluttered result (see Figure 18).
Figure 18: The minimal homology basis (in red) of a small subset of the data.

**Saturation of the complex**

It could instead be interesting to analyze the relationship between the filtration threshold and the rank of the resulting homology group. As evidenced in Figure 19, a known behaviour occurs: when the scale is very low, few edges exist and therefore very few cycles. Hence the Betti number is zero or small. As soon as the connectivity scale grows, a number of holes appear rapidly and the curve spikes up. Then, when the scale crosses a certain critical value, the number of cycles that become tiled (borders) exceeds the number of cycles that are created, and therefore the rank of $H_1$ decreases. Obviously, there exists a value $\epsilon_M := \max_e w_e$ where $w_e$ is the weight of edge $e$ such that $\forall \epsilon \geq \epsilon_M$ any clique complex $VR_\epsilon$ is entirely tiled, and therefore its homology group is trivial and the Betti number zero. Hence the $\beta/\epsilon$ curve is definitively zero.

```python
file_stream = open('CEFullNoDraws.dat', 'rb')
restore = cPickle.load(file_stream)
file_stream.close()

epsList = []
HList = []
for i in restore:
    epsList.append(i['Filtration Eps'])
    if i['SHB'].shape[0] != 0:
```

60
HList.append(i['SHB'].shape[1])
else:
    HList.append(0)
plt.plot(epsList, HList, 'r')

Listing 18: Post-processing of the C. Elegans analysis.
Figure 19: The rank of $H_1(\text{VR}_\epsilon(X_0))$ as a function of the threshold $\epsilon$.

**Edge relevance**

Let us lastly give a heuristic measure of the importance of an edge in the simplicial structure. The graph in Figure 20 is obtained by summing, for each edge from 1 to 2148, how many times it takes part into a shortest homology basis, along the filtration. One can immediately tell that there exists a very small number of edges that are very often part of a minimal cycle, while a large majority is rarely involved in the SHB structure. This gives us some valuable insight as to which links in the graph are the most critical.

```python
V = np.zeros((2148,1), dtype=int)
nonnull = [ x for x in restore if x["SHB"][0] != 0 ]
tot = 0

for i in nonnull:
    tot += i["SHB"][1]
    Sum = i["SHB"][0].sum(axis=1)
    V += Sum

print(tot)
V = Vect/tot
V = np.sort(V, axis=0)
V = np.flip(V, axis=0)
plt.plot(V,"b")
```

Listing 19: Edge hits distribution.
Figure 20: The descending-order normalized plot of how many times each edge from 1 to 2148 is part of a shortest homology basis. In this case, the total number of cycles in the SHB’s of the 31 simplicial complexes (sum of the Betti numbers at each filtration step) is 1096.
5 Appendix - Code

Repository available at https://github.com/marcoguerra192/PHScaffold ([25])

```python
# -*- coding : utf-8 -*-
### TESI: CLASSE per il problema SHORTEST PATH TREE

import numpy as np
from scipy.sparse import csgraph
import scipy.sparse
import functools
# from line_profiler import LineProfiler

class SimplexGraph:
    """ Classe che implementa un grafo come matrice di adiacenza pesata, e fornisce metodi per ottenere lo shortest path tree a partire da un nodo arbitrario, più l'elenco dei non-tree edges. Input costruttore: Il numero di vertici, o il numero di vertici e la matrice dei pesi """
    def __init__(self, numVertices, weights, maximal = None):
        self.NVert = numVertices
        self.Weights = scipy.sparse.coo_matrix(weights)
        ## coo_matrix può essere istanziata a partire da una matrice densa di tipo np.ndarray
        self.Weights = self.Weights.tocsr()
        # al momento ci serve che sia csr per accedere ai pesi

        self.dist = None
        self.pred = None
```

65
self.Followers = []
self.Sentinel = None
self.CCC = None
self.CCV = None
self.Origins = None
self.Ann = None
self.dimH1 = None
self.dimB1 = None
self.dimZ1 = None
self.Labels = -1 * np.ones((self.NVert, self.NVert), dtype=int)
self.Support = [] # vettori di supporto
self.SHB = [] # base minima di omologia
self.NTE = None # array di liste di NTE
self.closedCycles = {} # dizionario di cicli già chiusi
# la chiave è la terna (source, v1, v2)

# diversifichiamo il caso in cui abbiamo la
# filtrazione dal caso
# singolo step
if (maximal == None):
    rw, cl, _ = scipy.sparse.find(self.Weights)
    # righe e colonne sparse
    self.Edges = list(zip(list(rw), list(cl)))
    # lista di coppie di vertici
    self.Edges = [x for x in self.Edges if x[0]<x[1]]
    # scelgo solo quelli da un
    # vertice più basso ad uno più alto
    # funziona perché non ci sono self loops

def orderEdges( edgeList ):
    """ Funzione che ordina la lista di
    edge secondo qualche criterio.
    Per ora il criterio è prima
    rispetto al primo vertice,
    poi rispetto al secondo.
    """
    return sorted(edgeList, key = lambda
t: (t[0], t[1])

self.Edges = orderEdges(self.Edges)  
ora è una lista ordinata di edges,  
senza ripetizioni!

# può essere usata per tenere traccia  
dell’ordine degli edges
self.NEdges = len(self.Edges)  
# numero  
di edges IN SENSO INDIRETTO
self.WEdges = []  
# lista delle lunghezze (pesi) degli edges
for e in self.Edges:
    self.WEdges.append(self.Weights[e[0] , e[1]]

self.WEdges = np.array(self.WEdges)  
# trasformo in array
else:  
    # se ho definito la matrice più fine  
della filtrazione
    # maximal è una tupla = (edgelist,  
    weightlist)
    # la edgelist deve già essere ordinata
    self.Edges = maximal[0]  
    self.NEdges = len(self.Edges)  
    self.WEdges = maximal[1]

def getEdgeList(Weights):
    """ Metodo Statico per ottenere l’edgelist  
da una matrice massimale  
In input solo la matrice dei pesi  
"""
    rw, cl, _ = scipy.sparse.find(Weights)  
    # righe e colonne sparse
    edgelist = list(zip(list(rw), list(cl)))  
    # lista di coppie di vertici
    edgelist = [x for x in edgelist if x[0]<x [1]]  
    # scelgo solo quelli da un vertice  
più basso ad uno più alto  
# funziona perché non ci sono self loops

67
def orderEdges(edgeList):
    """ Funzione che ordina la lista di edge secondo l'ordine lessicografico """
    return sorted(edgeList, key = lambda t: (t[0],t[1]))

edgelist = orderEdges(edgelist) # ora è una lista ordinata di edges, senza ripetizioni!
WEdges = [] # lista delle lunghezze (pesi) degli edges
for e in edgelist:
    WEdges.append(Weights[e[0], e[1]])
WEdges = np.array(WEdges) # trasformo in array

return (edgelist,WEdges)

def getEdgeVector(self): # ordinamento degli archi
    return self.Edges

def getHeatMap(self, edgeList):
    """ Prende in input una lista di coppie di vertici (v1, v2), li ordina in modo che v1 < v2, e restituisce un vettore 0-1 lungo quanto la dimensione dello spazio vettoriale delle 1-catene, che descrive la 1-catena corrispondente. Se l'input contiene un edge inesistente o un self-loop, solleva ValueError """
    edgeList = [ x if (x[0] < x[1] ) else tuple(reversed(x)) for x in edgeList ] # impone v1 <= v2
    # e se v1 == v2 ?
    if ( any( [ x[0] == x[1] for x in edgeList ] ) ) :
raise ValueError('La lista contiene un self-loop!')  # eccezione

dgeList = list(set(edgeList))  # elimina i duplicati

heatMap = np.zeros(self.NEdges, dtype=int)  # vettore di zeri

for val in edgeList:
    try:
        i = self.Edges.index(val)
    except:
        errstr = "L'edge ", val, " non esiste!"
        raise ValueError(errstr)
    heatMap[i] = 1

return heatMap

def set_weights_by_coo(self, data, rows, col):
    """ Istanzia la matrice dei pesi nel formato sparso attraverso tre array data, rows e col"""
    self.Weights = scipy.sparse.coo_matrix((data,(rows,col)), shape=(self.NVert,self.NVert))

def computeShortestPath(self, withDistances=False):
    """ Restituisce la matrice dei predecessori nello Shortest Path Tree. Ogni riga si riferisce ad un nodo di origine dello SPT. Se l’argomento opzionale è vero ritorna anche la matrice delle distanze """

    # va convertito in un formato su cui può fare i calcoli

69
(self.dist, self.pred) = csgraph.
shortest_path(self.Weights.tocsr(),
directed=False, return_predecessors=True)

# Aggiungiamo il calcolo dei successori.
Utile da fare una volta invece di n
# Viene utile una struttura di lista
for source in range(self.NVert): # per ogni sorgente
    link = self.pred[source,:] # predecessori
    forward = [ [[] for i in range(self.NVert) ] ] # lista di liste vuote lunga NVert
    for (foll, prev) in enumerate(link):
        if (prev != -9999): # se è connesso e non è source
            forward[prev].append(foll)
    self.Followers.append(forward)
# non c'è nessun bisogno di restituire una matrice densa
# if withDistances:
#   return (self.pred, self.dist)
# else:
#   return self.pred

def shortestPathTree(self, source, withD=False, justEdges=False):
    """ Restituisce la matrice dello shortest path tree centrata nel vertice source. Se withD è vero
la matrice è pesata, altrimenti è solo 0-1. Se source non è un vertice valido solleva un'eccezione
I VERTICI VANO DA 0 A N-1!!
Se justEdges == True restituisce solo i non tree edges
Input: L'indice del nodo sorgente,"""
Opzionale un bool per le distanze
Output: La matrice sparsa del grafo SPT

if (type(source) is not int or source<0 or source>=self.NVert ):  
    raise Exception('Indice del vertice non valido!')

if withD:
    if (self.dist is None or self.pred is None): 
        _ = self.computeShortestPath(withD)
    else:
        if self.pred is None: 
            _ = self.computeShortestPath(withD)

# definiamo una matrice sparsa della giusta dimensione che andrà in output
# SPT = scipy.sparse.coo_matrix((self.NVert, 
#                            self.NVert)) # NO, SPRECA SOLO TEMPO!
# estraiamo dalla matrice dei precedenti la riga che ci serve
link = np.array(self.pred[source,:])
# predisponiamo i vettori data, rows e col per creare la matrice del SPT
data = []
rows = []
col = [] # man mano faremo append dei dati in queste liste
# scorriamo il vettore e creiamo i link
for i in range(len(link)):
    prev = link[i]
    if (prev == -9999): # non appartiene alla stessa componente connessa => non ha predecessori
        pass
    else:
rows.append(prev)
col.append(i)  # aggiungi un edge dal predecessore del nodo i-esimo al nodo i-esimo
rows.append(i)  # devo aggiungere anche il simmetrico!
col.append(prev)
if withD:  # se devo aggiungere le distanze le cerco in self.dist
data.append(self.dist[prev,i])
data.append(self.dist[prev,i])  # aggiungo anche il simmetrico (stessa distanza)
else:  # se no inserisco solo 1 per indicare l'edge fra prev e i
data.append(1)
data.append(1)

if withD:
    data = np.array(data, dtype=float)
else:
    data = np.array(data, dtype=int)
rows = np.array(rows)
col = np.array(col)
SPT = scipy.sparse.coo_matrix((data, (rows, col)), shape=(self.NVert, self.NVert))  # crea matrice del grafo
# Per trovare i non-tree edge: trovare gli elementi di Weights che NON SONO in SPT
rw, cl, _ = scipy.sparse.find(self.Weights)  # restituisce vettori di righe, colonne e valori non nulli
# ignoro il vettore di dati, che non ci interessa
edgeSPT = list(zip(list(rows), list(col)))  # creo una lista di edge per SPT
dewe = list(zip(list(rw), list(cl)))  # stessa cosa per il grafo originale
# MA CONTIENE ANCHE TUTTI GLI EDGES NELLE COMPONENTI NON CONNESSE!
# Affinché il metodo funzioni, è necessario che il nodo source non abbia -9999 per distinguergli da quelli davvero disconnessi
link[source] = source  # creiamo un "finto"
self-loop
dgeW = [ x for x in edgeW if link[x[0]] != -9999 ]
# limito a solo gli edge fra nodi che sono raggiungibili da source
NTE = list(set(edgeW) - set(edgeSPT))  # differenza insiemistica tra i set di edges
# contiene ancora entrambe le versioni dell’ edge (v1,v2 e v2,v1). Imponiamo solo quella (v1,v2) con v1<v2
NTE = [ x for x in NTE if (x[0]<x[1]) ]

if justEdges:
    return NTE
else:
    return (SPT, NTE)

def spanningTree(self):
    """ Calcola uno spanning tree a partire dalla stessa matrice dei predecesori di SPT. Questa volta, oltre a lavorare su tutte le componenti connesse, restituisce tutti i sentinel edges. """

    if self.pred is None:
        _ = self.computeShortestPath(True)
        # prepariamo le liste per la matrice del grafo
        rows = []
col = []
data = []
self.CCV = list(np.zeros(self.NVert, dtype=int))  # lista indice delle componenti connesse
self.CCC = 0  # contatore delle componenti connesse
self.Origins = []  # lista lunga come il numero di CC. In posizione k contiene l’origine della componente connessa k+1-esima

# che sarà necessario utilizzare per calcolare i sentinel edges!
while 0 in self.CCV:  # finché non ho assegnato ogni vertice ad una componente
    self.CCC += 1
    vert = self.CCV.index(0)  # trova il primo vertice non assegnato ad alcuna componente
    # Questo vertice diventerà l’origine della CC!! Sarà necessario usare lui per calcolare i sentinel edges!
    self.Origins.append(vert)  # questa CC origina da vert
    self.CCV[vert] = self.CCC  # assegna questo vertice alla nuova componente
    link = np.array(self.pred[vert,:])  # estraì i predecessori connessi a vert
    for i in range(len(link)):
        prev = link[i]
        if (prev == -9999):  # se è disconnesso fai nulla
            pass
        else:
            self.CCV[i] = self.CCC  # se è connesso assegna lo stesso indice di componente
            rows.append(prev)
col.append(i) # aggiungi un edge
dal predecessore del nodo i-
esimo al nodo i-esimo
rows.append(i) # devo aggiungere
anche il simmetrico!
col.append(prev)
data.append(1)
data.append(1)
data = np.array(data, dtype=int)
rows = np.array(rows)
col = np.array(col)
ST = scipy.sparse.coo_matrix((data,(rows,col)), shape=(self.NVert,self.NVert))
# matrice del grafo Spanning Tree
# per trovare i sentinel edges. Questo
spanning tree è già generato a partire
dall’origine di ogni CC!
rw,cl,_ = scipy.sparse.find(self.Weights) #
tutti gli edge del grafo originale
dgeW = list(zip(list(rw),list(cl))) #
lista edge grafo originale
dgeST = list(zip(list(rows),list(col))) #
lista edge spanning tree
self.Sentinel = list(set(edgeW) - set(edgeST)) # differenza insiemistica
# contiene ancora entrambe le versioni dell’
edge (v1,v2 e v2,v1). Imponiamo solo
quella (v1,v2) con v1<v2
self.Sentinel = [ x for x in self.Sentinel
if (x[0]<x[1]) ]
return (ST,self.Sentinel,self.CCV)

def closeCycle(self, E, force=None):
    ""
    Funzione che prende in input un sentinel
    edge E, e sfruttando la matrice
    self.pred e l’ordinamento self.Edges,
    restituisce un vettore dello spazio
delle 1-catene che descrive il ciclo identificato da E nello Shortest Path Tree.
Lo SPT è quello radicato nel nodo ORIGINE DELLA CC DI E!
Se l’input non è quello atteso, solleva ValueError.
SE E NON È UN SENTINEL EDGE NON SO BENE COSA SUCCEDA!
Se è definito force, forza il nodo sorgente ad essere quello. Utile per generare il candidate set.
Sfrutta la struttura self.closedCycles per verificare se una certa chiusura è già stata calcolata

if (force is not None): # se è fissata l’origine
    key = (force, E[0], E[1]) # la chiave è (source, v1,v2)
    try: # trova se è già stato calcolato
        cycle = self.closedCycles[ key ]
        return cycle
    except:
        pass

try:
    cycle = self.getHeatMap([E]) # trova l’heatmap del sentinel edge. Gestisce l’errore se E non è giusto
except Exception as ex:
    raise ex
# ora cycle contiene l’heatmap del solo sentinel edge.
# Proviamo a fare un controllino che sia un sentinel edge. Se self.Sentinel non esiste può comunque far danni
if (self.Sentinel is not None and force is None): # solo se non è definito force

    # (in quel caso ci sta che non siano sentinel edges)
    if (E not in self.Sentinel and tuple(reversed(E)) not in self.Sentinel):
        raise ValueError("E non è un sentinel edge!")

    # fissiamo i bordi del sentinel edge
    v1 = E[0]
    v2 = E[1]

    if (force is not None):
        if (force<0 or force>=self.NVert):
            raise ValueError("Parametro force non valido!")
else:
    source = force
else:
    # troviamo il NODO ORIGINE della CC di E
    source = self.Origins[ self.CCV[ v1 ] -1 ] # -1 perché le CC partono da 1 ma i vettori da 0. Sarebbe uguale usare
    v2
    key = (source, E[0], E[1]) # calcoliamo
    la chiave del ciclo da chiudere
    # Calcoliamo i pred, se non è già stato
    fatto
    if (self.pred is None):
        self.pred = self.computeShortestPath( False)

    link = self.pred[source,:] # vettore dei
    precedenti rispetto a source
    pr = link[v2]
    if (pr == -9999 and v2 != source): #
        significa che v2 è irraggiungibile da
        source, ci deve essere stato un errore
        raise ValueError("Critical Error! Source
        e v2 dovrebbero essere per
        costruzione nella stessa CC")

    follow = v2
    while (pr != -9999): # segue all'indietro i
        precedenti verso source (che ha link
        -9999)
        cycle += self.getHeatMap( [ (pr, follow
        ) ] )
        follow = pr
        pr = link[pr]

    # Ora stessa cosa per v1, ma gli edge in
    comune fra i due path vanno eliminati =>
    SOMMA % 2
pr = link[v1]
if (pr == -9999 and source != v1): #
    significa che v1 è irraggiungibile da
    source, sono in due componenti connesse
diverse
    raise ValueError('Critical Error! Source
    e v1 dovrebbero essere per
costruzione nella stessa CC')
follow = v1
while (pr != -9999): # finchè non torno a
    source
    cycle += self.getHeatMap([ (pr , follow
    ) ])
    follow = pr
    pr = link[pr]
# ora alcuni cicli saranno stati contati 2
    volte. Quelli in comune vanno eliminati!
cycle = cycle % 2

# ora cycle contiene la descrizione
    vettoriale del ciclo identificato dal
# sentinel edge E, rispetto all’ordinamento
    self.Edges

# aggiorniamo il dizionario dei cicli chiusi
    self.closedCycles[ key ] = cycle

return cycle

def getCycleBase(self):
    
    Calcola una base dei cicli a partire dai
    sentinel edges. Per ottenere l’
    annotazione
    
    # se ancora non l’ha fatto calcola i
    sentinel edges
    if (self.Sentinel is None):
        _ = self.spanningTree()
SC = []  # lista di sentinel cycles
for e in self.Sentinel:
    SC.append( self.closeCycle(e) )

return SC

def getAnnotation(self,d2,Z):
    """
d2: border matrix of the set of faces of the simplicial complex, borders are written as rows.
Z: basis of 1-dim-cycles. Each row is a cycle obtained from a spanning tree and a sentinel edge.
returns an annotation of sentinel edges. Non sentinel edges have annotation equal to 0.
"""

def low(col):
    """
col: 1-dimensional array.
gets the index of the "lowest" element in col different from 0.
if col=0 then low = -1
"""
    l=-1;
    for i in range(len(col)):
        if col[i]>0:
            l=i
    return l

lowSet = {};  #dictionary with low indexes and relative rows

if len(d2) == 0:  # controlla se d2 è vuoto!
dimB1 = 0

80
i = 0
else:
  i=0;
while i != len(d2):
  lowRowi=low(d2[i])
  while lowRowi in lowSet.keys():
    d2[i]=(d2[i]+d2[lowSet[lowRowi]])%2
    lowRowi=low(d2[i])
  if lowRowi > -1 :
    lowSet[lowRowi]=i
    i=i+1
else:
  d2=np.delete(d2,(i),axis=0)
dimB1=i # dimensione dello spazio dei bordi

if (dimB1 != 0):
  Z=np.concatenate((d2,Z),axis=0)
else:
  pass # lasciamo Z
#we start the reduction from row dimB1
totRow=len(Z)
reductionMatrix=np.identity(totRow,dtype=int)
Id=np.identity(totRow,dtype=int)
elementsToDelete=[]

while i != totRow:
  lowRowi=low(Z[i])
  while lowRowi in lowSet.keys():
    Z[i]=(Z[i]+Z[lowSet[lowRowi]])%2
    reductionMatrix[i]=(reductionMatrix[i]+Id[lowSet[lowRowi]])%2
    lowRowi=low(Z[i])
  if lowRowi > -1:
    lowSet[lowRowi]=i
i=i+1
else:
    elementsToDelete.append(i)
    i=i+1

# eliminate coordinates of cycles that are borders:
reductionMatrix=np.delete(reductionMatrix, elementsToDelete, axis=1);
range(dimB1),axis=1);
A=np.delete(reductionMatrix,range(dimB1),
axis=0);
""" observation: the number of rows of A is the dimension of the 1-dim-cycle group; the number of columns is the dimension of the 1st homology group """

self.dimB1 = dimB1
self.dimZ1 = np.shape(A)[0]
self.dimH1 = np.shape(A)[1]

# OTTIMIZZAZIONE!
# Qui verrebbe bene creare un dizionario di coppie edge-annotazione per velocizzare la ricerca negli step successivi. Edge nel formato (v1, v2), con v1<v2
self.AnnDict = {}
for i,e in enumerate(self.Sentinel):
    ann = self.Ann[:,i]
    self.AnnDict[e] = ann

return (self.Ann, self.dimB1, self.dimZ1, self.dimH1)

def innerProd(self,S,C):
Scalar Product over Z2 of support vector S and the ANNOTATION of cycle C

```python
if (len(S) != len(C)):
    raise ValueError("Dimensioni non compatibili!")

return np.dot( np.array(S).transpose() , np.array(C) ) % 2
```

def computeLabels(self, Sup):
    """ Calcola i labels rispetto al support vector S_i Sup """

    if (self.Ann is None):
        raise ValueError("Non è stata calcolata l'annotazione degli edges!")

    for p in range(self.NVert):  # per ogni root
        # link = self.pred[p, :]
        forward = self.Followers[p]

        """ Utilizziamo un metodo tipo push pop: ad ogni nodo aggiungiamo la lista dei successori in coda, poi pop di uno e avanti così """

        self.Labels[p,p] = 0  # il label di source è 0
        driver = []  # lista di push/pop DI TUPLE (PREVIOUS, FOLLOWER)

        # iniziamo dai successori di source
        add = [ (p,x) for x in forward[p] ]
        driver.extend( add )

        while ( len(driver) != 0):  # finché non ho esaurito i successori
            (prev,foll) = driver.pop(0)  # pop del primo elemento (BREADTH FIRST !)

            lab = self.Labels[p,prev]  # leggiamo il label del nodo precedente
edge = (prev, foll) if prev < foll
else (foll, prev) # scriviamo l'edge
try: # leggiamo l'annotazione dell'edge
    ann = self.AnnDict[edge]
except KeyError:
    self.Labels[p, foll] = lab # non è un sentinel e quindi non cambia
else:
    # calcola il prodotto scalare in Z2 e somma
    self.Labels[p, foll] = (lab +
        self.innerProd(Sup, ann))%2
# Ora aggiungiamo alla lista driver i successori di foll e avanti
add = [(foll, x) for x in forward[foll]]
driver.extend(add)

#return self.Labels # inutile restituire una struttura pesante

def fixNTE(self):
    """ Calcola un array di NTE per ogni sorgente p, e la salva in self.NTE """
    self.NTE = []
    for p in range(self.NVert):
        nte = self.shortestPathTree(p, justEdges=True)
        for e in nte:
            self.NTE.append( (p,e) )
    # ora tutti i NTE sono calcolati una volta per tutte

def resetLabels(self):

84
""" Resetta i labels per poterli calcolare rispetto al nuovo vettore di supporto
"""
self.Labels = -1 * np.ones((self.NVert, self.NVert), dtype=int)

def findShortestNonOrtho(self, Sup, allDraws=False):
    """ Funzione che genera una lista di cicli non-ortogonali (m = 1) a Sup
    I NTE seguono la convenzione lessicografica
    If allDraws ritorna una lista di tutti i cicli minimi a parità!
    """
    # bisogna controllare che sia stato generato tutto il resto
    # calcolare i labels
    #self.resetLabels() # riporta i labels a -1
    _ = self.computeLabels(Sup)

def checkOrt(s, e):
    """ In input una sorgente dello SPT e un edge scritto giusto
    """
    try:
        ann = self.AnnDict[e]  # trova l’annotazione dal dizionario
    except KeyError:  # se non appartiene ai sentinel vale 0
        lab = 0
    else:
        lab = self.innerProd(Sup, ann)  # calcola il prodotto
return (self.Labels[s,e[0]] + self.
Labels[s,e[1]] + lab ) %2

candidates = [] # lista di tuple (origine, NTE)

# selezioniamo solo quelli non ortogonali
candidates = [ x for x in self.NTE if
checkOrt(x[0],x[1]) == 1 ]

# generiamo i vettori dei cicli a partire da
(source, non-tree edge)
candidates = [ self.closeCycle(x[1], force=x [0]) for x in candidates ]

# definiamo un metodo per calcolare la
lunghezza di un ciclo
def lenCycle(C):
    """ Calcola la lunghezza di un ciclo
    come prodotto scalare tra il vettore dei pesi self.WEdges e la descrizione del ciclo nella base self.Edges
    """
    return np.dot( self.WEdges, C)

# list of tuples (ciclo, lunghezza)
candidates = list(zip( candidates, map( lenCycle, candidates) ))

if allDraws: # restituisce tutti i pareggi
    shortest = candidates[0][1]
    minList = []
    for x in candidates:
        if x[1] < shortest:
            minList = [x]
            shortest = x[1]
        elif x[1] == shortest:
            minList = minList + [x]

86
minList.append(x)
else:
    pass
return minList
else:  # trovane solo uno
    min_len = lambda x,y : x if x[1] <= y[1]
    else y
minCycle = functools.reduce( min_len , candidates )
return minCycle

def cycleAnnotation(self, C):
    """ Calcola l'annotazione del ciclo C (somma delle ann degli edges)  """
    C = map( lambda x: bool(x), C )  # rendilo bool
    edges = [ e for (e , filt) in zip(self.Edges , C) if filt ]
    ann = np.zeros( (self.dimH1,1), dtype = int )
    for e in edges:
        try:
            a = self.AnnDict[e]
            ann = (ann + a) % 2
        except KeyError:
            pass  # se non è un sentinel fai nulla
    return ann

def updateSup(self, newC, index):
    """ Funzione che aggiorna i support vector per ogni nuovo ciclo minimo della base di omologia. Fa in pratica un Graham-Schmidt in Z2 Deve calcolare l'annotazione del ciclo.  """
 Carni deve andare da 0 a g-1 

```python
# Ciclo della SHB numero (i) = index
self.SHB.append( newC ) # aggiungi il nuovo ciclo
# Il nuovo ciclo è newC
# index = len(self.SHB) # indice del nuovo ciclo
sup_i = self.Support[index] # support vector del nuovo ciclo
# Il support vector relativo è sup_i
AnnNewC = self.cycleAnnotation(newC) # annotazione del nuovo ciclo
# L'annotazione del nuovo ciclo è AnnNewC
for j in range(index+1,self.dimH1) : # per S_j da i+1 a g
    # scorri su (j) = j)
    sup_j = self.Support[j]
    # Sup_j = sup_j
    # Prodotto scalare = self.
    innerProd( AnnNewC , sup_j )
    # S_i * <C,S_j> = sup_i * self.
    innerProd( AnnNewC , sup_j )
    # S_j + S_i * <C,S_j> = (sup_j + sup_i * self.
    innerProd( AnnNewC , sup_j ) % 2
    # somma = sottrae) la proiezione
    self.Support[j] = (sup_j + sup_i * self.
    innerProd( AnnNewC , sup_j ) % 2
# Fine ciclo i = index
```

```python
def run(self):
    """ Metodo che concatena tutti i passaggi e 
```
calcola la base minima di omologia

""

pass
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines
from math import sin, cos, pi
from random import random
from SPT import SimplexGraph
import scipy.sparse
import time
from multiprocessing import Pool
from datetime import datetime
import _pickle as cPickle

filename = 'CElegansDataDump'
timestamp = str(datetime.now()).replace(" ", "h")
filename += timestamp
filename += '.dat'
print(filename)

def getTriangles(A):
    '''
    returns set of triangles in a graph given its adjacency matrix
    '''
    n = len(A)
    tri = []
    for vertex in range(n):
        vList = np.nonzero(A[vertex, vertex:])[1]  # NB: qui c'era uno zero che non faceva più funzionare l'algoritmo
        vList = [i + vertex for i in vList]  # list of vertices adjacent to vertex
        for i in range(len(vList)-1):
            for j in range(i+1, len(vList)):
                if A[vList[i], vList[j]] > 0:
                    tri.append([vertex, vList[i], vList[j]])
def getD2(A, edgesList):
    '''
    returns 2-boundary matrix of the clique complex,
    given the adjacency matrix of the graph
    '''
    triangles = getTriangles(A)
    # print(triangles)

    d2 = []
    n = len(edgesList)
    # print(n)
    for row in triangles:
        newTriangle = [0 for i in range(n)]
        newTriangle[edgesList.index((row[0],row[1]))] = 1
        newTriangle[edgesList.index((row[0],row[2]))] = 1
        newTriangle[edgesList.index((row[1],row[2]))] = 1
        d2.append(newTriangle)
    d2 = np.array(d2)
    return d2

def points2adj(P, epsilon):
    '''
    given a set of points and a threshold returns
    the weight and adjacency matrix of the
    corresponding graph
    points is a list where each element is the
    coordinates of the point
    '''

    def dist(p1,p2):
        '''
        return tri

def getD2(A, edgesList):
    '''
    returns 2-boundary matrix of the clique complex,
    given the adjacency matrix of the graph
    '''
    triangles = getTriangles(A)
    # print(triangles)

    d2 = []
    n = len(edgesList)
    # print(n)
    for row in triangles:
        newTriangle = [0 for i in range(n)]
        newTriangle[edgesList.index((row[0],row[1]))] = 1
        newTriangle[edgesList.index((row[0],row[2]))] = 1
        newTriangle[edgesList.index((row[1],row[2]))] = 1
        d2.append(newTriangle)
    d2 = np.array(d2)
    return d2

def points2adj(P, epsilon):
    '''
    given a set of points and a threshold returns
    the weight and adjacency matrix of the
    corresponding graph
    points is a list where each element is the
    coordinates of the point
    '''

    def dist(p1,p2):
euclidean distance
p1 and p2 need to be list of the same length

return math.sqrt(sum([(p1[i]-p2[i])**2 for i in range(len(p1))]))

n = len(P)
W=np.zeros((n,n))
for i in range(n):
    for j in range(i,n):
        W[i,j]= dist(P[i],P[j])
W = W + W.T  # weight matrix
W[W >= epsilon] = 0

A = (W >0).astype(int)
return W , A

def filterMatrix(W, epsilon):
    Wb = np.matrix(W)
    Wb[Wb >= epsilon] = 0
    return Wb

def plotCycle(ax, cic, edList, P):
    for i in range(len(cic)):
        if cic[i] > 0:
            edge = edList[i]
p1 = edge[0]
p2 = edge[1]
line = lines.Line2D([P[p1][0],P[p2][0]],[P[p1][1],P[p2][1]], color='r')
#line = lines.Line2D([P[edList[cic[i]][0]][0],P[edList[cic[i]][1]][0]],[P[edList[cic[i]][0]][1],P[edList[cic[i]][1]][1]],[0], color='r')
ax.add_line(line)

def sampleCircle(x0,y0,r,n, noise):
P = []
for i in range(n):
    theta = random() * 2 * pi
    R = r + noise * random()
    xp = x0 + R * cos(theta)
    yp = y0 + R * sin(theta)
    P.append([xp, yp])
return P

def getEL(A):
    edgesList = []
    for i in range(len(A)):
        for j in range(i, len(A)):
            if A[i, j] > 0:
                edgesList.append([i, j])
    return edgesList

def plot_examples(SHBi, Wi, maximal, P):
    fig, ax = plt.subplots()
    x = [i[0] for i in P]
    y = [i[1] for i in P]
    ax.scatter(x, y)
    for row in getEL(Wi):
        line = lines.Line2D([P[row[0]][0], P[row[1]][0]], [P[row[0]][1], P[row[1]][1]])
        ax.add_line(line)
    for row in SHBi.T:
        row = row.tolist()
        row=row[0]
        plotCycle(ax, row, maximal[0], P)
    plt.show()
return

def plot_filtration(SHB, Ws, maximal, data):
    for index in range(len(SHB)):
        plot_examples(SHB[index], Ws[index], maximal, P)
def createRandomScatter(N, noise):
P = []
for i in range(N):
x = noise * random()
y = noise * random()
P.append([x, y])
return P

def parallel_pipeline(eps):
    W = GlobalOptions['Matrix']
nVert = len(W)
SHB = []
Ws = []
res = {}
maximal = SimplexGraph.getEdgeList(W)
stats = []
Wstep = filterMatrix(W, eps)
Ws.append(Wstep)
G = SimplexGraph(nVert, Wstep, maximal)
cycles = G.getCycleBase()
stats.append('NVert = ' + str(G.NVert) + 'NEdges = ' + str(G.NEdges))
stats.append('Filtration Eps = ' + str(eps))
stats.append('Keep all Draws = ' + str(GlobalOptions['Draws']))
t = time.time()
d2 = getD2(Wstep, maximal[0])  # ottiene d2 da Wstep ed edgelist
(An, B1, Z1, H1) = G.getAnnotation(d2, cycles)
elapsed = time.time()-t
stats.append('H1= ' + str(H1) + ' B1= ' + str(B1) + ' Z1= ' + str(Z1))
stats.append('Annotazione in sec ' + str(elapsed))
sup = np.eye(H1, dtype = int)
Sup = []
for i in range(H1):
    Sup.append( sup[:,i] )
G.Support = Sup
G.fixNTE()  # precalcoliamo i non tree edges

for i in range(H1):
    if GlobalOptions['Draws']:
        listMin = G.findShortestNonOrtho( G.Support[i], allDraws=True)
        # scegli il primo per l'update
        mincycle = listMin[0][0]
        SHB.append(listMin)
    else:
        (mincycle, length) = G.findShortestNonOrtho( G.Support[i],
            allDraws=False )
        G.updateSup(mincycle, i)

elapsed = time.time() - t1
stats.append( 'Cicli minimi in sec ' + str(elapsed) )

if GlobalOptions['Draws']:
    res['SHB'] = SHB
else:
    res['SHB'] = np.matrix(G.SHB).transpose()

res['Draws'] = GlobalOptions['Draws']
res['Filtration Eps'] = eps

if GlobalOptions['ReturnMatrix']:
    res['Filtered Matrix'] = Ws
else:
    res['Filtered Matrix'] = None

if GlobalOptions['ReturnMaximal']:
    res['Max'] = maximal
else:
    res['Max'] = None

res['stats'] = stats
return res
data = np.loadtxt("celegans_weighted_undirected.edges")
v1 = data[:,0]
v1 = np.array(v1, dtype=int)
v2 = data[:,1]
v2 = np.array(v2, dtype=int)
weights = data[:,2]
wmax = max(weights)
weights = [float(wmax)/x for x in weights]
weights = np.array(weights, dtype=float)
FullEpsList = list(set(weights))
NVert = max(max(v1), max(v2))
print("Numero vertici:", NVert)
print("Min: ", min(weights), "Max: ", max(weights),
"Total: ", len(weights), "Different: ", len(set(weights)))
plt.hist(weights, range(int(min(weights)), int(max(weights))))
print(FullEpsList)
v1 = np.array([x-1 for x in v1])
v2 = np.array([x-1 for x in v2])
# matrice dei pesi
W = scipy.sparse.coo_matrix((weights,(v1,v2)), dtype=float)
W = W.todense()
# I dati non sono SIMMETRICI!
W = (W+W.T)

GlobalOptions = {}
GlobalOptions['Matrix'] = W
GlobalOptions['Draws'] = False
GlobalOptions['ReturnMatrix'] = True
GlobalOptions['ReturnMaximal'] = True

file_stream = open(filename,'wb')
FullThreshList = [x + 0.05 for x in FullEpsList]
epsList = FullThreshList
cut = 306
\( W_n = W[0:cut, 0:cut] \)

\[
\text{GlobalOptions['Matrix']} = W_n
\]

\[
\text{pool} = \text{Pool(4)}
\]

\[
\text{job} = \text{pool.map_async(parallel_pipeline, epsList)}
\]

\[
\text{result} = \text{job.get()}
\]

\[
\text{cPickle.dump(result, file_stream)}
\]

\[
\text{file_stream.close()}
\]

\[
\text{file_stream} = \text{open(filename, 'rb')}
\]

\[
\text{restore} = \text{cPickle.load(file_stream)}
\]

\[
\text{file_stream.close()}
\]

\[
\text{print(restore[0].keys())}
\]
References


