A Space and Time Efficient Implementation for Computing Persistent Homology

Jean-Daniel Boissonnat, Tamal K. Dey, Clément Maria

A Space and Time Efficient Implementation for Computing Persistent Homology

Jean-Daniel Boissonnat, Tamal K. Dey, Clément Maria

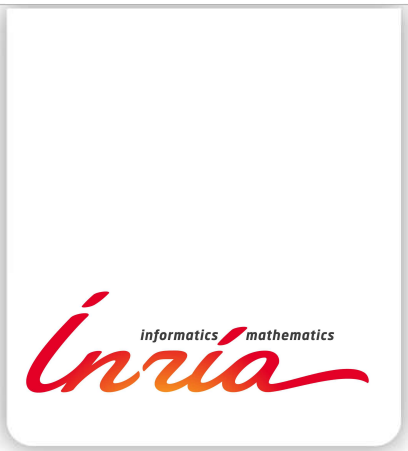
RESEARCH REPORT

N° 8195

December 2012

Project-Team GEOMETRICA

ISSN 0249-6399 ISRN INRIA/RR--8195--FR+ENG



A Space and Time Efficient Implementation for Computing Persistent Homology

Jean-Daniel Boissonnat, Tamal K. Dey, Clément Maria

Project-Team GEOMETRICA

Research Report n° 8195 — December 2012 — 17 pages

Abstract: The persistent homology with $\mathbb{Z}_2$-coefficients coincides with the same for cohomology because of duality. Recently it has been observed that the cohomology based algorithms perform much better in practice than the orginally proposed homology based persistence algorithm. We implement a cohomology based algorithm that attaches binary labels called annotations with the simplices. This algorithm fits very naturally with a recently developed data structure called simplex tree to represent simplicial complexes. By taking advantages of several practical tricks such as representing annotations compactly with memory words, using a union-find structure that eliminates duplicate annotation vectors, and a lazy evaluation, we save both space and time cost for computations. The complexity of the procedure, in practice, depends almost linearly on the size of the simplicial complex and to the variables related to the maximal dimension of the local homology groups we maintain during the computation, which remain small in practice. We provide a theoretical analysis as well as a detailed experimental study of our implementation. Experimental results show that our implementation performs several times better than the existing state-of-the-art software for computing persistent homology in terms of both time and memory requirements and can handle very large complexes efficiently (several hundred million simplices in high-dimension).

Key-words: persistent homology, persistent cohomology, implementation, annotation, simplex tree

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Une Implémentation pour le Calcul de l'Homologie Persistente Efficace en Temps et Espace

Résumé : Nous proposons une implémentation de l'homologie persistente d'une filtration d'un complexe simplicial. L'algorithme utilise le concept d'annotation et maintient une base de cohomologie durant l'insertion successive des faces du complexe simplicial, dans l'ordre de la filtration. L'implémentation utilise un simplex tree pour représentrer la filtration et les annotations sont représentées en compressant des vecteurs de bits en mots mémoire et en supprimant les doublons par le biais d'une structure union-find. Enfin, nous proposons une évaluation "paresseuse" des simplexes pour réduire la complexité des groupes de cohomologie représentés par les annotations. La complexité de la procédure dépend quasi-linéairement de la taille du complexe simplicial, et de variables dépendant de la dimension du complexe et de la dimension maximale des groupes d'homologie maintenus durant le calcul, qui reste petite en pratique.

Mots-clés : homologie persistente, cohomologie persistente, implémentation, annotation, simplex tree

1. Introduction

Persistent homology is an algebraic method for measuring the topological features of a space induced by sublevel sets of a function. Its generality and stability with regard to noise have made it a widely used tool for the study of data, where it does not need any knowledge a priori. A common approach is the study of the topological invariants of a nested family of simplicial complexes built on top of the data, seen as a set of points in a geometric space. This approach has been successfully used in various areas of science and engineering, as for example in sensor networks [8], image analysis [4], and data analysis [5], where one typically needs to deal with big data sets in high dimensions. Consequently, the demand for designing efficient algorithms and implementations to compute the persistent homology of filtered simplicial complexes has grown.
The first persistence algorithm introduced by Edelsbrunner et al. in [14] can be implemented by a reduction of a matrix of size quadratic in the number of faces, through column operations. A compact *sparse matrix representation* can be used to reduce the memory cost of the algorithm [13]. The running time is $O(|\mathcal{K}|^3)$ where $|\mathcal{K}|$ is the size of the simplicial complex $\mathcal{K}$ and, despite good performance in practice, Morozov proved that this bound is tight [21]. Other approaches based on matrix multiplication and rank computation lead to better running times [19, 6], but necessitate quadratic space. A recent approach [10] which interprets the persistent homology groups in terms of their dual, the persistent cohomology groups, has been reported to work better in practice [9]. We implement an algorithm based on this approach [12] and show that it works quite well in tandem with a recently introduced data structure for simplicial complexes [3].
Our implementation leverages on several observations that speeds up the computation. We use a recently introduced data structure called *simplex trees* [3] to represent the simplicial complex of the input filtration. Simplices in this tree are tagged with a binary vector called *annotations*. The algorithm by Dey et al. [12] that we implement maintains a cohomology basis under insertion of simplices, using these annotations. Simplices are maintained in equivalence classes using a union-find data structure where two simplices are made equivalent if they have the same dimension and they evaluate the same way on the chosen cohomology basis. The binary annotation vectors are maintained by integers which occupy only a few memory words in practice. Lastly, we use a lazy evaluation of the annotation that helps keeping their lengths smaller which in turn saves time in their updates. We provide a rigorous theoretical analysis of our implementation where we show that the method depends almost linearly on the size of the complex and to some variables that depend on the maximal dimension g_{m} of the homology groups that occur during processing the input filtration. The original persistence algorithm when implemented with matrix reduction using sparse matrices can be analyzed to take $O(|\mathcal{K}|r^2)$ time for a simplicial complex $\mathcal{K}$ where r is either the maximum index of a creator simplex or the difference between the indices of a destructor simplex and its paired simplex [13]. Thus, it depends on a quantity such as r which being more global than g_{m} generally assumes larger value. Furthermore, g_m can be lowered by a lazy evaluation as our experiments show. A straightforward implementation of the annotation based algorithm can achieve a time complexity of $O(|\mathcal{K}|^2 g_{\mathrm{m}})$. We implement the same algorithm that replaces the quadratic dependence on $|\mathcal{K}|$ with increased dependence on the local quantity g_{m}.

2. Background

A *simplicial complex* is a pair $\mathcal{K} = (V, S)$ where V is a finite set whose elements are called the *vertices* of $\mathcal{K}$ and S is a set of non-empty subsets of V that is required to satisfy the following two conditions : 1. $p \in V \Rightarrow \{p\} \in S$ and 2. $\sigma \in S, \tau \subseteq \sigma \Rightarrow \tau \in S$. Each element $\sigma \in S$ is called a *simplex* or a *face* of $\mathcal{K}$ and, if $\sigma \in S$ has precisely $s+1$ elements ($s \geq -1$), σ is called an s-simplex and the dimension of σ is s. The dimension of the simplicial complex $\mathcal{K}$ is the largest k such that S contains a k-simplex.
A *filtration* [13] of a simplicial complex is an order relation on its simplices which respects inclusion. Considering a simplicial complex $\mathcal{K}$ and a function $\rho : \mathcal{K} \to \mathbb{R}$. We require ρ to be monotonic in a sense that for any two simplices $\tau \subset \sigma$ in $\mathcal{K}$, ρ satisfies $\rho(\tau) \leq \rho(\sigma)$. Monotonicity implies that the sublevel set $\mathcal{K}(r) = \rho^{-1}(-\infty, r]$ is a subcomplex of $\mathcal{K}$, for every $r \in \mathbb{R}$. Let $\rho_i \in \mathbb{R}$ be minimal such that $\mathcal{K}(\rho_i)$ (noted $\mathcal{K}_i$ for short) contains at least i faces. If $\mathcal{K}$ contains m faces we have:

$$\emptyset = \mathcal{K}_0 \subseteq \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{K}_{m-1} \subseteq \mathcal{K}_m = \mathcal{K},\ \rho_0 \leq \rho_1 \leq \cdots \leq \rho_{m-1} \leq \rho_m$$

where $\sigma = \mathcal{K}_i \setminus \mathcal{K}_{i-1}$ appears at $\rho(\sigma) = \rho_i$. We will call $\rho(\sigma)$ the *time of appearance* of σ in the filtration.
Let $H_p(\mathcal{K})$ and $H^p(\mathcal{K})$ denote respectively the homology and cohomology groups of $\mathcal{K}$ in dimension p with $\mathbb{Z}_2$ coefficients. The filtration induces a sequence of homomorphisms in the homology and cohomology groups in opposite directions:

$$0 = H_p(\mathcal{K}_0) \to H_p(\mathcal{K}_1) \to \cdots \to H_p(\mathcal{K}_m) = H_p(\mathcal{K}) \tag{2.1}$$

$$0 = H^p(\mathcal{K}_0) \leftarrow H^p(\mathcal{K}_1) \leftarrow \cdots \leftarrow H^p(\mathcal{K}_m) = H^p(\mathcal{K}) \tag{2.2}$$

We refer to [15] for an introduction to the theory of (co)homology and to [13] for an introduction to persistent homology. It is known that because of duality the two sequences above provide the same persistence diagrams [10].
The original persistence algorithm [14] considers the homology sequence in 2.1 that aligns with the filtration direction. It detects when a new homology class is born and when an existing class dies as we proceed forward through the filtration. Recently, a few algorithms have considered the cohomology sequence in 2.2 which runs in the opposite direction of the filtration [10, 9, 12]. The birth of a cohomology class coincides with the death of a homology class and the death of a cohomology class coincides with the birth of a homology class. Therefore, by tracking a cohomology basis along the filtration direction and switching the notions of births and deaths, one can obtain all information about the persistent homology. The algorithm of de Silva et al. [10] computes the persistent cohomology following this principle which is reported to work better in practice than the original persistence algorithm [9]. Recently, Dey et al. [12] recognized that tracking cohomology bases provides a simple and natural extension of the persistence algorithm for filtrations connected with simplicial maps. Their algorithm is based on the notion of annotation and when restricted to only inclusions is similar to the algorithm of de Silva et al. [10]. Here we follow this annotation based algorithm.

3. Annotations

The annotation based algorithm in [12] maintains a cohomology basis under simplex insertions, where cohomology cycles are represented by the value they take on the simplices. We rephrase the description of this algorithm.

Definition 3.1. Given a simplicial complex $\mathcal{K}$, let $\mathcal{K}^p$ denote the set of p-simplices in $\mathcal{K}$. An annotation for $\mathcal{K}^p$ is an assignement $\mathsf{a}^p : \mathcal{K}^p \to \mathbb{Z}_2^g$ of a binary vector $\mathsf{a}_\sigma = \mathsf{a}^p(\sigma)$ of same length g for each p-simplex $\sigma \in \mathcal{K}$. We use a when there is no ambiguity in the dimension. We also have an induced annotation for any p-chain c_p given by $\mathsf{a}_{c_p} = \sum_{\sigma \in c_p} \mathsf{a}_\sigma$.

Definition 3.2. An annotation $\mathsf{a} : \mathcal{K}^p \to \mathbb{Z}_2^g$ is *valid* if:

(1) $g = \operatorname{rank} H_p(K)$, and
(2) two p-cycles z_1 and z_2 have $\mathsf{a}_{z_1} = \mathsf{a}_{z_2}$ iff their homology classes $[z_1]$ and $[z_2]$ are identical.

Proposition 3.3 ([12])**.** *The following two statements are equivalent*

(1) *An annotation* $\mathsf{a} : \mathcal{K}^p \to \mathbb{Z}_2^g$ *is valid*
(2) *The cochains* $\{\phi_j\}_{j=1\cdots g}$ *given by* $\phi_j(\sigma) = \mathsf{a}_\sigma[j]$ *for all* $\sigma \in \mathcal{K}^p$ *are cocycles whose cohomology classes* $\{[\phi_j]\}_{j=1\cdots g}$ *constitute a basis of* $H^p(\mathcal{K})$.

A valid annotation is thus a way to represent a cohomology basis. The algorithm for computing persistent homology consists in maintaining a valid annotation for each dimension when inserting all simplices in the order of the filtration, and deducing by duality the persistence diagram. Since we process the filtration in a direction opposite to the cohomology sequence (as in Equation 2.2), we discover the death points of cohomology classes earlier than their birth points. To avoid the confusion, we still say that a new cocycle (or its class) is born when we discover it for the first time and an existing cocycle (or its class) dies when we see it no more. It will be convenient, for the description of the algorithm, to picture a valid annotation for the p-simplices of $\mathcal{K}_i$ as a $g \times |\mathcal{K}_i^p|$ matrix of 0 and 1s, called A_i^p, with g the dimension of $H^p(\mathcal{K}_i)$ and the j^{th} column being the annotation vector of the j^{th} p-simplex and the i^{th} row being a cocycle whose cohomology class is a basis element of $H^p(\mathcal{K}_i)$.
We present the algorithm and refer to [12] for its validity. We insert simplices in the order of the filtration, and refer to $\mathcal{K}_i$ as the *current* simplicial complex, and to $H^p(\mathcal{K}_i)$ as the *current* cohomology group. Consider an elementary inclusion $\mathcal{K}_i \hookrightarrow \mathcal{K}_{i+1}$. Assume that $\mathcal{K}_i$ has a valid annotation for each dimension. We describe how to obtain a valid annotation for $\mathcal{K}_{i+1}$ from that of $\mathcal{K}_i$ after inserting the p-simplex $\sigma = \mathcal{K}_{i+1} \setminus \mathcal{K}_i$. We compute the annotation $\mathsf{a}_{\partial\sigma}$ for the boundary $\partial\sigma$ in $\mathcal{K}_i$, using A_i^{p-1}, and take actions as follows:
Case 1: If $\mathsf{a}_{\partial\sigma}$ is a zero vector, the insertion of σ creates a new cohomology class, born at time $\rho_i = \rho(\sigma)$. We add a new row at index $g+1$ and a new column at index $|\mathcal{K}_i^p|+1$, setting all their entries to 0 except $A_{i+1}^p(g+1, |\mathcal{K}_i^p|+1)$, which is set to 1. σ is assigned the new column as annotation vector, and we have created a new cohomology basis element in row $g+1$, representative of the new cohomology class.
Case 2: If $\mathsf{a}_{\partial\sigma}$ is not zero, the insertion of σ removes a cohomology class, dead at time

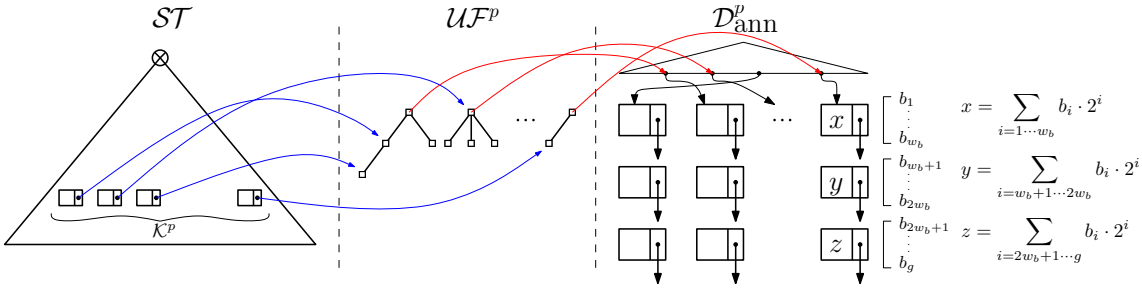


FIGURE 1. The simplicial complex is represented by a simplex tree $\mathcal{ST}$ (left). The annotation vectors, represented as sequences of intergers coded on w_{b} bits, are maintained in a dictionary $\mathcal{D}^p_{\text{ann}}$ (right). $\mathcal{UF}^p$ connects the p-simplices in $\mathcal{ST}$ to their annotation vector in $\mathcal{D}^p_{\text{ann}}$ (middle).

$\rho_{i+1} = \rho(\sigma)$. We consider the 1 in $\mathsf{a}_{\partial\sigma}$, corresponding to the youngest cohomology basis element not null in $\mathsf{a}_{\partial\sigma}$. Let i be the index of this row. We add $\mathsf{a}_{\partial\sigma}$ to all columns having a 1 at index i, and remove row i. Finally we assign σ a null vector as annotation vector. We have removed the cocycle in row i, representative of the cohomology class killed by the insertion of σ.

As with the original persistence algorithm, the pairing of simplices is derived from the creation and destruction of the cohomology basis elements.

4. IMPLEMENTATION

We divide the data structures to compute persistent homology into three parts: the first represents efficiently the simplicial complex in a simplex tree, the second stores the annotations and the third interfaces simplices and annotations using the equivalence relation defined above (see Figure 1).

4.1. Simplicial Complex and Simplex Tree.

The *simplex tree*, which has been introduced by Boissonnat and Maria in [3], constructs and represents filtered simplicial complexes. The structure is a tree whose nodes are in bijection with the faces of the simplicial complex it represents. We represent the filtered simplicial complex in a simplex tree $\mathcal{ST}$ (Figure 1, left). Each node represents a simplex of the complex, and stores additional information, like the time of appearance of the simplex. The nodes of the simplex tree are maintained in an ordered list which represents a filtration of the simplicial complex. The operation on the data structure $\mathcal{ST}$ is:

- COMPUTE-BOUNDARY(σ): computes the boundary of a simplex σ, *i.e.* finds the set of nodes in the simplex tree $\mathcal{ST}$ representing the codimension 1-subfaces of σ.

Complexity: The size of the simplex tree is linear in the number of faces of the simplicial complex: $O(|\mathcal{K}|)$. The time to compute the boundary of a simplex with COMPUTE-BOUNDARY(σ), using the algorithm described in [3], is $O(|\sigma|^2 D_m)$, where D_m is a value related to the time needed to traverse the simplex tree and is bounded by $\log(d^o_{\text{m}})$, where d^o_{m} is the maximal degree of a vertex in the simplicial complex.

4.2. **Representation of Annotations.** For each dimension p, we consider an annotation for the p-simplices of the simplicial complex $\mathsf{a} : \mathcal{K}^p \to \mathbb{Z}_2^g$, assigning $s = |\mathsf{a}(\mathcal{K}^p)|$ distinct annotation vectors of length g to the simplices of $\mathcal{K}^p$. We represent only the s distinct annotation vectors.
We compactly represent an annotation vector by packing bits into integers. Specifically, let w_b be the number of bits used to represent an integer (usually $w_b = 32$ or 64), an annotation vector of length g is represented by a sequence of $\lceil g/w_b \rceil$ integers. The set of annotation vectors for a dimension p is maintained in a dictionary $\mathcal{D}^p_{\text{ann}}$ (e.g. a red-black tree), allowing fast look-up, insertions and deletions (Figure 1, right). We store a table $\mathcal{T}_{\text{app}}$ which associates each row index to the time of appearance of the corresponding cocycle during the filtration, and a heap $\mathcal{H}_{\text{row}}$ storing the indices of unoccupied rows.
The annotation assignment changes with the successive insertions of simplices, induced by the order of the filtration. We need the following operations on $\mathcal{D}^p_{\text{ann}}$:

- SUM-ANN: computes the modulo two sum of two annotation vectors. This is performed using XOR operations between the corresponding integers of the sequences, which costs $O(\lceil g/w_{\text{b}} \rceil)$.
- SEARCH-ANN/ADD-ANN/REMOVE-ANN: searches, adds or removes an annotation vector from $\mathcal{D}^p_{\text{ann}}$. To do so, it performs dictionary operations in $\mathcal{D}^p_{\text{ann}}$, for a cost of $O(\lceil g/w_{\text{b}} \rceil \log(s))$.
- REMOVE-ROW: sets each bit to zero in a given row, and adds the index of the row to the heap $\mathcal{H}_{\text{row}}$ storing the available rows, for a cost of $O(\lceil g/w_{\text{b}} \rceil s + \log(g))$.
- ADD-ROW: pops the index of an available row from the heap $\mathcal{H}_{\text{row}}$, which costs $O(\log(g))$. If $\mathcal{H}_{\text{row}}$ is empty and the length of the annotation vectors is g, it adds a $g + 1^{\text{th}}$ row. If $g + 1 \equiv 1 \bmod [w_{\text{b}}]$, it adds an integer to each sequence representing an annotation vector, for a cost of $O(s)$.
- CHECK-ANN: traverses all bits of an annotation vector a_σ and returns, if it exists, the index j for which $\mathsf{a}_\sigma[j] = 1$ and all other indices $\ell \neq j$ satisfy that either $\mathsf{a}_\sigma[\ell] = 0$ or the cocycle corresponding to the row ℓ is older than the one corresponding to the row j. It uses $\mathcal{T}_{\text{app}}$ to determine the time of appearance of the cocycles. This process has cost $O(g)$.

4.3. **Interface Simplicial Complex-Annotations.** We say that two simplices σ and τ of $\mathcal{K}$ are equivalent iff they have the same dimension p and the same annotation vector $\mathsf{a}_\sigma = \mathsf{a}_\tau$. For a dimension p, we represent the classes of equivalent simplices in a *union-find* [7] data structure $\mathcal{UF}^p$ (Figure 1, middle). A union-find is encoded as a forest where each tree contains the elements of an equivalent class, the root being the representative of the class. The trees of $\mathcal{UF}^p$ are in bijection with the different annotation vectors stored in the annotation dictionary $\mathcal{D}^p_{\text{ann}}$ and only the root of a $\mathcal{UF}^p$-tree maintains a pointer to the corresponding annotation vector in $\mathcal{D}^p_{\text{ann}}$. Each node representing a p-dimensional simplex σ in the simplex tree $\mathcal{ST}$ stores a pointer to an element of the $\mathcal{UF}^p$-tree associated to its annotation vector a_σ. Finding the annotation vector of σ consists in getting the element it points to in a $\mathcal{UF}^p$-tree and in finding the root of the $\mathcal{UF}^p$-tree which points to a_σ in $\mathcal{D}^p_{\text{ann}}$. We avail the following operations $\mathcal{UF}^p$:

- CREATE-CLASS: creates a new tree containing one element.
- FIND-ROOT: finds the root of a tree, given an element in the tree.

- UNION-CLASSES: merges two trees.

For a dimension p, the number of elements maintained in $\mathcal{UF}^p$ is at most the number of faces of dimension p, i.e. $|\mathcal{K}^p|$. The operations on $\mathcal{UF}^p$ can be computed in amortized time $O(\alpha(|\mathcal{K}^p|))$, where $\alpha(n)$ is the very slowly growing inverse Ackermann function (constant less than 4 in practice).

4.4. **Implementation of the Algorithm.** We consider all simplices in the filtration order and update the data structures during the successive insertions. The simplicial complex is stored in a simplex tree $\mathcal{ST}$ and we maintain, for each dimension p, a structure $\mathcal{UF}^p$ and a dictionary $\mathcal{D}^p_{\text{ann}}$. Let $\sigma = \mathcal{K}_i \setminus \mathcal{K}_{i-1}$ be a p-simplex we insert. We compute a_σ by searching for the boundary of σ in the simplex tree $\mathcal{ST}$ with COMPUTE-BOUNDARY. We find the annotation vector a_τ for each of the $p+1$ faces τ in $\partial\sigma$ by FIND-ROOT in $\mathcal{UF}^p$. We obtain a_σ by issuing $p+1$ calls to SUM-ANN. We check the annotation vector $\mathsf{a}_{\partial\sigma}$ with CHECK-ANN and take action as follows:
Case 1: If $\mathsf{a}_\sigma = 0$, we create a new row in $\mathcal{D}^p_{\text{ann}}$ with ADD-ROW at index j picked from the heap $\mathcal{H}_{\text{row}}$, and insert a new annotation vector for σ in $\mathcal{D}^p_{\text{ann}}$ using ADD-ANN, with a 1 at index j. We create a new class of equivalence in $\mathcal{UF}^p$ using CREATE-CLASS, pointing to the new annotation vector a_σ.
Case 2: If $\mathsf{a}_\sigma \neq 0$, CHECK-ANN returns the index j in a_σ which corresponds to the youngest cocycle for which $\mathsf{a}_\sigma[j] = 1$. We iterate through all the annotation vectors $\mathsf{a}\tau$ of $\mathcal{D}^p_{\text{ann}}$ to find the ones for which $\mathsf{a}_\tau[j] = 1$. If so, we proceed to compute $\mathsf{a}_\tau \leftarrow \mathsf{a}_\tau + \mathsf{a}_\sigma$ using ADD-ANN. We search $\mathcal{D}^p_{\text{ann}}$ to determine if the new value of a_τ is in duplicate in $\mathcal{D}^p_{\text{ann}}$, using SEARCH-ANN. If so, we remove a_{tau} from $\mathcal{D}^p_{\text{ann}}$ and merge the two $\mathcal{UF}^p$-trees having the same annotation vector using UNION-CLASSES.

The algorithm returns the intervals of the persistence diagram.

Remark: in the 0-dimensional case, vertices are also cycles. Consequently, by definition of a *valid* annotation, two vertices are equivalent iff they are homologous as cycles, or equivalently, if they belong to the same connected component. During the computation of persistent homology, the cocycles $\{\phi_j\}_{j=1\cdots g}$ we maintain with the annotations are the cocycles satisfying $\phi_j(v_\ell) = 1$ iff $j = \ell$, for a set of vertices $\{v_1, \cdots, v_g\}$ whose cohomology classes form a basis of $H^0(\mathcal{K})$. We do not represent the annotation for the vertices as we can deduce automatically a valid annotation. We only maintain a union-find structure $\mathcal{UF}^0$, and this implementation is equivalent to the one of Delfinado and Edelsbrunner for computing 0-homology in [11].

4.5. **Complexity Analysis.** Consider the cost of inserting a p-dimensional face $\sigma = \mathcal{K}_{i+1} \setminus \mathcal{K}_i$ where, at time i, we update an annotation $\mathsf{a} : \mathcal{K}_i^p \to \mathbb{Z}_2^g$ assigning $s = |\mathsf{a}(\mathcal{K}_i^p)|$ distinct annotation vectors of length g to the p-simplices of $\mathcal{K}_i$.
To obtain the annotation vector a_σ from the node representing σ in the simplex tree, we perform a call to COMPUTE-BOUNDARY, $p+1$ calls to FIND-ROOT, $p+1$ calls to SUM-ANN and a call to CHECK-ANN for a total cost of $O(p^2 D_{\text{m}} + p\alpha(|\mathcal{K}^p|) + p\lceil g/w_{\text{b}}\rceil + g)$. If $\mathsf{a}_{\partial\sigma} = 0$, we perform a call to ADD-ROW, a call to ADD-ANN and a call to CREATE-CLASS for a total cost of $O((\log(g) + s) + \lceil g/w_{\text{b}}\rceil \log(s) + \alpha(|\mathcal{K}_i^p|))$. If $\mathsf{a}_{\partial\sigma} \neq 0$, we iterate through all annotation vectors in $\mathcal{D}^p_{\text{ann}}$ and, in the worst case, remove all of them from $\mathcal{D}^p_{\text{ann}}$ and merge all $\mathcal{UF}^p$-trees together, performing a call to REMOVE-ROW and s calls to ADD-ANN, SEARCH-ANN, REMOVE-ANN and UNION-CLASSES for a total cost of

$O(\lceil g/w_{\mathrm{b}} \rceil s + s \times (\lceil g/w_{\mathrm{b}} \rceil + 2 \cdot \lceil g/w_{\mathrm{b}} \rceil \log(s) + \alpha(|\mathcal{K}_i^p|)))$.
Let g_{m} be the maximal dimension of a current homology group over all dimensions that we maintain during the computation, and s_{m} be the maximal number of distinct annotation vectors an annotation assignment may have. For a simplicial complex $\mathcal{K}$ of dimension k, the total complexity of the entire computation is then:

$$O\left(|\mathcal{K}|\left[k\left(\alpha(|\mathcal{K}|) + \left\lceil \frac{g_{\mathrm{m}}}{w_{\mathrm{b}}} \right\rceil + kD_{\mathrm{m}}\right) + s_{\mathrm{m}}\left(\left\lceil \frac{g_{\mathrm{m}}}{w_{\mathrm{b}}} \right\rceil \log(s_{\mathrm{m}}) + \alpha(|\mathcal{K}|)\right)\right]\right)$$

The memory complexity of the data structures is linear in the number of faces of $\mathcal{K}$ for the simplex tree $\mathcal{ST}$ and $\mathcal{UF}^p$. $\mathcal{D}^p_{\mathrm{ann}}$ maintains at most s_{m} annotation vectors of size $O(\lceil g_{\mathrm{m}}/w_{\mathrm{b}} \rceil)$ each. The total memory cost of the full computation is thus:

$$O\left(|\mathcal{K}| + k\frac{g_{\mathrm{m}} s_{\mathrm{m}}}{w_b}\right)$$

The time to insert a simplex σ depends only on its dimension and the quantities g_{m} and s_{m}, which are related to the properties of the current homology groups. The memory cost is the cost for storing the input filtration, plus an additonal term depending only on g_{m} and s_{m}.
The quantity s_{m} is bounded by $2^{g_{\mathrm{m}}}$ and $|\mathcal{K}|$. In practice, we observe that s_{m} is of the same order as g_{m}.
Consequently, in practice, and for a fixed dimension, the complexity of the computation is linear in the number of faces $|\mathcal{K}|$ of the complex, times a factor depending only on the dimension of the homology groups we maintain.

Remark: we can reduce the memory cost by computing the persistence of one dimension at a time. Each simplex is parsed twice instead of once and we maintain only one dictionary of annotation vectors $\mathcal{D}^p_{\mathrm{ann}}$ and only one $\mathcal{UF}^p$ at a time, for each dimension p. The computation of the persistence diagrams for the different dimensions is also naturally parallelizable.

5. Lazy Evaluation of Annotations

The performance of the algorithm depends on our ability to keep the maximal dimension g_{m} small during the computation of the persistent homology. We propose a lazy evaluation of simplices to minimize the number of cocycles represented simultaneously. To optimize our computation, we postpone the insertion of a positive simplex σ until we consider one of its cofaces in the filtration: instead of inserting a simplex σ at time $\rho(\sigma)$ the procedure defers its actual insertion to the time $\min_{\tau \not\supseteq \sigma}\{\rho(\tau)\}$. We observe that a simplex σ that creates a cocycle acquires an annotation which does not affect any other annotation during subsequent annotation updates until a coface of σ appears. Therefore, the updates can proceed with deferring the actual insertion of σ. We prove in Appendix A that this gives the same persistence diagram. The lazy evaluation reduces the dimension of the local homology group we maintain, and thus reduces the length of the annotation vectors.

To compute the persistence homology using the lazy evaluation, we mark each simplex whose insertion has been postponed. As before, we call the insertion procedure Lazy-evaluation on each simplex in the order of the filtration. Suppose we are considering

a p-simplex σ on which we call LAZY-EVALUATION. If σ is marked, LAZY-EVALUATION directly inserts it as a creator, without computing the annotation of its boundary, and unmarks σ. If σ is not marked, LAZY-EVALUATION computes the boundary $\partial\sigma$ of σ and calls itself recursively on each marked face of $\partial\sigma$. LAZY-EVALUATION then computes $\mathtt{a}_{\partial\sigma}$ and processes as follows. If $\mathtt{a}_{\partial\sigma}$ is not null (σ kills a cocycle) LAZY-EVALUATION proceeds to the standard insertion of σ and updates the annotation. If $\mathtt{a}_{\partial\sigma}$ is null then σ creates a cocycle and LAZY-EVALUATION postpones its insertion by simply marking it. At the begin of the computation, no face is marked. The recursive call to the marked subfaces garantees that all subfaces of σ have been inserted when we compute $\mathtt{a}_{\partial\sigma}$. Note that we call LAZY-EVALUATION on a simplex σ at most twice (marking a simplex and then inserting it), but there is no additional cost for the computation because we can directly insert a marked simplex as a creator, without computing its boundary.

In section 6, we give experimental evidence that this lazy evaluation is effective at decreasing the maximal dimension of the homology groups we maintain.

6. Experiments

In this section, we report on the performance of our implementation to compute persistent (co)homology along three directions. We first compare the computation of persistent cohomology with the computation of persistent homology, and give a detailed study of why persistent cohomology is much faster in practice. Then, we compare our implementation of persistent cohomology with the existing software based on the persistent cohomology algorithm [9]. Finally, we compare the memory efficiency of our implementation with other representations.
The experimental study is done on both real and synthetic data, with comparisons to existing software. More specifically, we benchmark computation of persistent homology on the filtration of Rips complexes. Given a threshold $\rho_{\max} \geq 0$, the Rips complex of a set of point $\mathcal{P}$ in a metric space is defined as the set of simplices whose vertices have pairwise distances less than $\rho_{\max}$. Recent investigations in topological data analysis have recognized the usefulness of this complex [5, 2].
Our implementations are in `C++`. We use the `ANN` library [22] to compute the pairwise distances between points and construct the Rips complex using the simplex tree based algorithm described in [3]. All timings are measured on a Linux machine with 3.00 GHz processor and 32 GB RAM. Timings are provided by the `clock` function from the `Standard C Library` for codes in `C++`, and by the `tic` function for `Matlab`. Zero means that the measured time is below the resolution of the time function. All timings are averaged over 10 independent runs.
We use a variety of both real and synthetic datasets: **S4** is a set of points sampled from the unit 4-sphere in $\mathbb{R}^5$; **Bud** is a set of points sampled from the surface of the *Stanford Buddha* [1] in $\mathbb{R}^3$; **Bro** is a set of 5×5 *high-contrast patches* derived from natural images, interpreted as vectors in $\mathbb{R}^{25}$, from the Brown database (with parameter $k = 300$ and cut 30%) [17, 4]; **Cy8** is a set of points in $\mathbb{R}^{24}$, sampled from the space of conformations of the cyclo-octane molecule [18], which is the union of two intersecting surfaces; finally, **Kl** is a set of points sampled from the surface of the figure eight Klein Bottle embedded in $\mathbb{R}^5$.

Data	$\lvert\mathcal{P}\rvert$	D	d	r	k	$\lvert\mathcal{K}\rvert$	T_{ann}	$T_{\text{ann}}/\lvert\mathcal{K}\rvert$(s.)
S4	507	5	4	0.75	5	$191 \cdot 10^6$	$\sim$ 32min	$1.0 \cdot 10^{-5}$
Bud	1000	3	2	2.4	3	$283 \cdot 10^6$	$\sim$ 10min	$2.2 \cdot 10^{-6}$
Bro	500	25	?	0.8	16	$38 \cdot 10^6$	$\sim$ 181min	$2.8 \cdot 10^{-4}$
Cy8	500	24	2	1.3	17	$18 \cdot 10^6$	$\sim$ 9min	$3.1 \cdot 10^{-5}$
Kl	4900	5	2	0.415	5	$218 \cdot 10^6$	$\sim$ 29min	$7.9 \cdot 10^{-6}$

FIGURE 2. Data, timings and statistics

Datasets are listed in Figure 2 with details on the sets of points $\mathcal{P}$, their size $|\mathcal{P}|$, the ambient dimension D, the intrinsic dimension d of the object the sample points belong to (if known), the threshold $\rho_{\max}$, the dimension k of the simplicial complexes, the size $|\mathcal{K}|$ of the simplicial complexes, the time T_{ann} to compute their persistence diagram and the time per face $T_{\text{ann}}/|\mathcal{K}|$ spent for computing the persistent homology. We test the performance of our implementation on these datasets, and compare it to two existing software that are state-of-the-art. We compare our implementation to the JPLEX [23] library and the DIONYSUS [20] library. The first is a Java package which can be used with `Matlab` and provides an implementation of persistent homology. The second is implemented in `C++` and provides implementations for both persistent homology and cohomology. Both libraries are widely used to construct simplicial complexes and to compute their persistent (co)homology.
We also provide an experimental analysis of the memory performance of our data structure compared to other representations.
Unless mentionned otherwise, simplicial complexes are constructed up to the embedding dimension, because the homology is trivial in dimension higher.
As reported in Figure 2, we are able to compute the persistent homology of simplicial complexes of hundred of million simplices and of high dimension (more than 10).

6.1. **Time Performance Cohomology vs Homology.** We compare the time performance of our implementation of persistent cohomology with the computation of persistent homology in JPLEX and in DIONYSUS. See the left column of Figure 3. We restricted the computation to simplicial complexes of dimension less than 7, because JPLEX does not construct simplicial complexes of bigger dimension. Sometimes we had to stop the experiments because the running-time was too long, which explains why some points are missing from the plots. On the different data sets, JPLEX is faster than DIONYSUS on small simplicial complexes (small $\rho_{\max}$), but DIONYSUS shows better performance on bigger simplicial complexes. In particular, we have been able to run DIONYSUS on bigger filtrations than JPLEX.
Our implementation is faster than both libraries on every data set, at any scale, and can run on bigger simplicial complexes. Our implementation runs 5 to 135 times faster than JPLEX on **S4**, 2 to 154 faster on **Bud**, 2 to 91 times faster on **Bro**, 2 to 65 times faster on **Cy8** and 3 to 443 times faster on **Kl**. The time-ratio between the two implementations gets big when the input filtrations grow. Our implementation is 65 to 267 times faster than DIONYSUS on **S4**, 45 to 145 times faster on **Bud**, 13 to 234 times faster on **Bro**, 24 to 130 times faster on **Cy8** and 30 to 185 times faster on **Kl**.

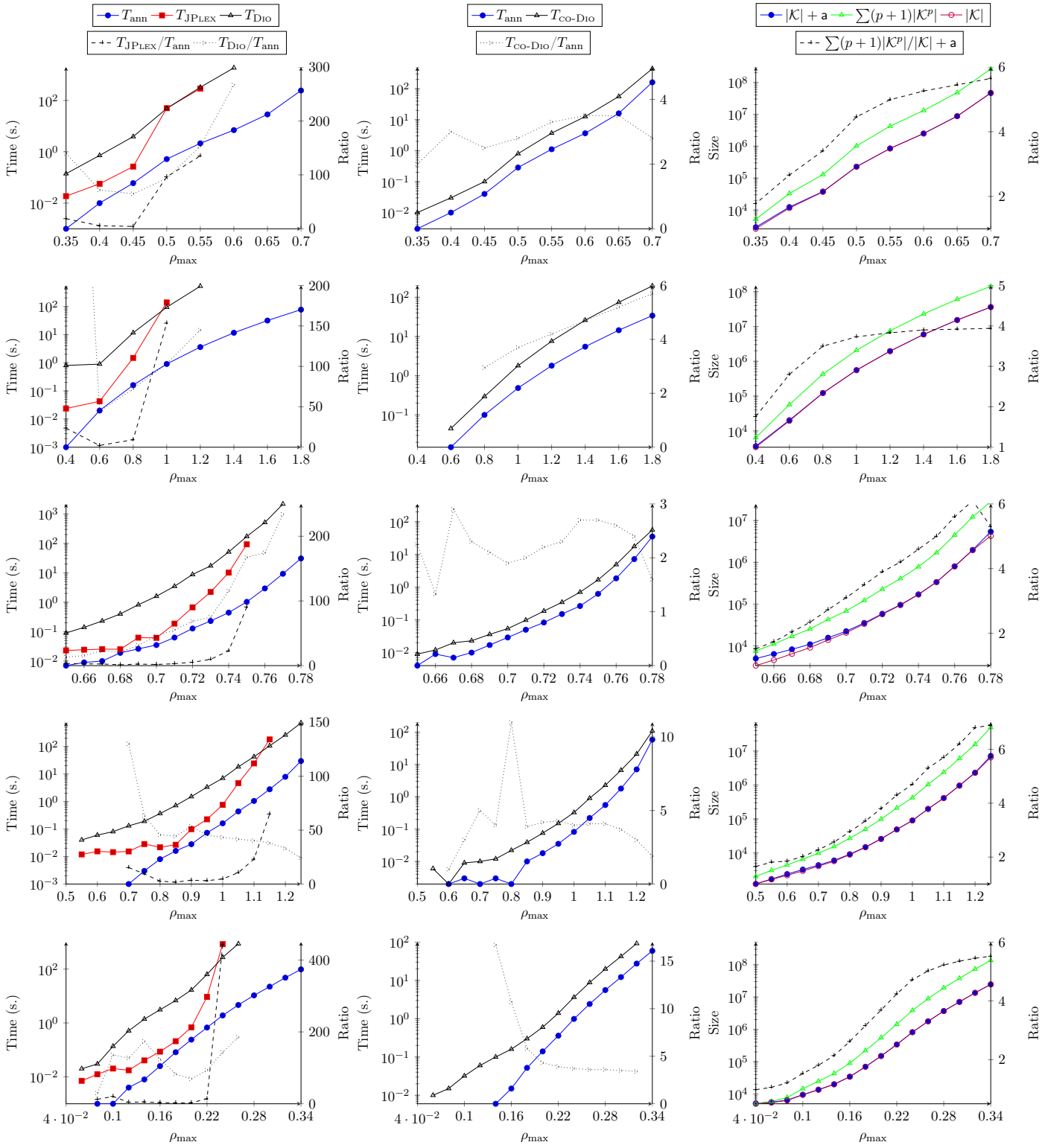


FIGURE 3. Left column: comparison of time performance between our implementation (T_{ann}), and the computation of persistent homology with JPLEX (T_{JPlex}) and DIONYSUS (T_{Dio}). Middle column: comparison of time performance of our implementation (T_{ann}) and the computation of persistent cohomology in DIONYSUS ($T_{\text{co-Dio}}$). Right column: comparison of the memory performance of our implementation $|\mathcal{K}|$ + ann with the sparse matrix representation $\sum(p+1)|\mathcal{K}^p|$. From Top to Bottom: Rips complex for 507 points from **S4**, 1000 points from **Bud**, 500 points from **Bro**, 500 points from **Cy8** and 4900 points from **Kl**.

6.2. **Time Performance of the Implementation.** We compare the performance of our implementation of persistent cohomology with the computation of persistent cohomlogy implemented in DIONYSUS [10]. See the middle column in Figure 3. In the representation of Rips complexes in DIONYSUS, a simplex stores a list of pointers to the simplices of its boundary. To compare the time performance of the computation of persistent cohomology, we augment the representation of a simplex as a node in the simplex tree with a list of pointers to the simplices of its boundary. In these experiments, we construct all simplicial complexes until the embedding dimension.
On all data sets and at all scales, our implementation is several times faster than DIONYSUS. Our implementation is 2 to 3.5 times faster on **S4**, 3 to 6 times faster on **Bud**, 2 to 3 times faster on **Bro**, 2 to 4.5 times faster on **Cy8** and 3.5 to 17 times faster on **Kl**.

6.3. **Memory Performance.** In the right column of Figure 3, we report on the memory performance of our implementation. As mentionned before, our implementation requests $O(|\mathcal{K}| + kg_{\mathrm{m}}s_{\mathrm{m}}/w_{\mathrm{b}})$ space for representing the simplicial complex and the annotations. We report $|\mathcal{K}|$ + ann in our experiments, where $|\mathcal{K}|$ stands for the size of the simplex tree, up to a constant factor, and ann is the maximal number of blocks of bits stored in the annotation data structures (all dimensions together) during the computation. In the literature, the most memory efficient algorithm to compute persistent homology is based on the *sparse matrix representation* [14] of the matrix of the boundary operator, which represents only the 1s and has size $\Omega(\sum_p (p+1)|\mathcal{K}^p|)$. We report $\sum(p+1)|\mathcal{K}^p|$ in our experiments.
All along the experiments, the size of our representation remains really close to the size of the input filtration $|\mathcal{K}|$. Indeed, the extra-cost for representing the annotation represents a few percents of the size of the structures, dominated by the size of the simplicial complex. More precisely, in the case of **S4**, the structure to store the annotations represents at most 10% of the total size for small complexes, and less than 0.1% for big ones. For **Bud**, the memory cost decreases from 6.6% to 0.002% when $\rho_{\max}$ increases. The annotation structures represent between 0.6% and 20% of the memory cost for **Bro**, between 0.2% and 9.8% of the memory cost for **Cy8**, and between 3% and 0.1% of the memory cost for **Kl**. Our representation is better than the sparse matrix one, mostly on big simplicial complexes. Indeed, the two representations are equivalent when the complexes have few faces but we observe, in the different experiments, a gain of factor ranging from 4 to 6 for big simplicial complexes.

6.4. **Performance and Limitation.** We also report in Figure 2 the computing time needed per face which depends on the dimension k of the simplicial complex as indicated by the complexity analysis. For **S4**, **Bud** and **Kl**, the ambient dimension is less than 5, and so is k. For these examples, we can handle simplicial complexes with hundreds of million faces in less than 30 minutes and the limiting factor is the memory size. Differently, for **Bro** and **Cy8**, the dimension of the ambient space is big (25 and 24) and so are the dimensions of the simplicial complexes (16 and 17 respectively). The limiting factor is then the computing time. Nevertheless, we are able to compute the persistence diagram on complexes with tens of million faces on these examples in a reasonnable time.

6.5. **Dimension of the Homology Groups during the Computation.** As mentionned earlier, the time and space efficiency of our implementation depends on the

Dim:	**S4**: 1	2	3	4	**Bud**: 1	2	**Kl**: 1	2	3	4
$G_\mathrm{m}/g_\mathrm{m}$	4.9	1.2	6.2	6.1	2.1	8.4	11.8	10.3	10.3	8.9
$\lvert\mathcal{K}^p\rvert/s_\mathrm{m}$	10^2	10^3	10^4	10^3	10^3	10^5	10^3	10^4	10^4	10^4
$\lceil g_\mathrm{m}/w_b \rceil$	5	4	10	86	5	2	4	1	11	87

Dim:	**Bro**: 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$G_\mathrm{m}/g_\mathrm{m}$	2.1	1.8	5.4	5.7	5.2	5.6	4.0	4.2	3.0	3.7	3.7	2.9	3.0	3.7	3.0
$\lvert\mathcal{K}^p\rvert/s_\mathrm{m}$	10^1	10^3	10^3	10^3	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2
$\lceil g_\mathrm{m}/w_\mathrm{b} \rceil$	6	6	6	36	150	330	703	694	726	343	133	46	8	228	342

Dim:	**Cy8**: 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$G_\mathrm{m}/g_\mathrm{m}$	2.3	1.7	3.9	3.8	4.1	4.1	3.5	3.5	3.9	2.7	2.6	3.0	2.8	1.9	2.7	1
$\lvert\mathcal{K}^p\rvert/s_\mathrm{m}$	10^3	10^3	10^3	10^3	10^3	10^3	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2	10^2
$\lceil g_\mathrm{m}/w_\mathrm{b} \rceil$	3	1	3	12	33	74	147	186	160	168	93	33	10	3	1	1

FIGURE 4. Statistics about the evolution of the dimension of the homology groups we maintain during the computation.

maximal dimension g_m of the homology groups we maintain during the computation of the persistent homology and the maximal number s_m of distinct annotation vectors assigned to simplices. In Figure 4, we present statistics about these quantities, extracted from the experiments presented in Figure 2.

First, we have observed that, for all experiments and all dimensions, the quantity s_m is of the same order as g_m, never exceeding it by a factor more than 3.

For a dimension p, G_m is the maximal dimension of $H_p(\mathcal{K}_i)$ during the filtration ($i = 0 \cdots m$) and g_m is the maximal dimension of the current p-homology group we maintain, using the lazy evalutation presented in section 5. We report the ratio $G_\mathrm{m}/g_\mathrm{m}$ and notice the efficiency of the lazy evaluation, which reduces the dimension of the maintained homology group by a factor 4.3 in average (over all dimensions and all experiments).

We report the order of magnitude of the ratio $|\mathcal{K}^p|/s_\mathrm{m}$, which illustrates the interest of the union-find data structure $\mathcal{UF}^p$ to store no duplicate annotation vector.

In the implementation, we compress the annotation matrix by packing the bits into memory words of length w_b bits. We report the ratio $\lceil g_\mathrm{m}/w_b \rceil$ which represents the maximal number of rows in the matrix we represent, with $w_\mathrm{b} = 32$. Despite the big size of the simplicial complexes, the length $\lceil g_\mathrm{m}/w_b \rceil$ remains small in pratice.

CONCLUSION

We have presented an implementation of an algorithm for computing persistent homology that is fully general since it relies on a general data structure that can represent any simplicial complex. The implementation is fast and compact and can compute the persistent homology of simplicial complexes that are bigger than what was doable before. A public and fully documented version of our code will be released soon. On the theoretical side, we intend to better understand and optimize the behaviour of this algorithm under realistic conditions. We also intend to develop a parallel version of the algorithm.

Acknowledgements

This research has been partially supported by the 7th Framework Programme for Research of the European Commission, under FET-Open grant number 255827 (CGL Computational Geometry Learning).

References

[1] The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[2] Dominique Attali, André Lieutier, and David Salinas. Vietoris-rips complexes also provide topologically correct reconstructions of sampled shapes. In *Symposium on Computational Geometry*, pages 491–500, 2011.

[3] Jean-Daniel Boissonnat and Clément Maria. The simplex tree: An efficient data structure for general simplicial complexes. In Leah Epstein and Paolo Ferragina, editors, *ESA*, volume 7501 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 2012.

[4] Gunnar Carlsson, Tigran Ishkhanov, Vin Silva, and Afra Zomorodian. On the local behavior of spaces of natural images. *Int. J. Comput. Vision*, 76:1–12, January 2008.

[5] F. Chazal and S. Oudot. Towards persistence-based reconstruction in euclidean spaces. In *Proc. 24th. Annu. Sympos. Comput. Geom.*, pages 231–241, 2008.

[6] Chao Chen and Michael Kerber. An output-sensitive algorithm for persistent homology. In Hurtado and van Kreveld [16], pages 207–216.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] V. de Silva and R. Ghrist. Coverage in sensor network via persistent homology. *Algebraic & Geometric Topology*, 7:339–358, 2007.

[9] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. *Inverse Problems*, 27:124003, 2011.

[10] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Persistent cohomology and circular coordinates. *Discrete Comput. Geom.*, 45:737–759, 2011.

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

[12] Tamal K. Dey, Fengtao Fan, and Yusu Wang. Computing topological persistence for simplicial maps. *CoRR*, abs/1208.5018, 2012.

[13] Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction*. American Mathematical Society, 2010.

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

[15] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[16] Ferran Hurtado and Marc J. van Kreveld, editors. *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*. ACM, 2011.

[17] Ann B. Lee, Kim Steenstrup Pedersen, and David Mumford. The nonlinear statistics of high-contrast patches in natural images. *International Journal of Computer Vision*, 54(1-3):83–103, 2003.

[18] S. Martin, A. Thompson, E.A. Coutsias, and J. Watson. Topology of cyclo-octane energy landscape. *J Chem Phys*, 132(23):234115, 2010.

[19] Nikola Milosavljevic, Dmitriy Morozov, and Primoz Skraba. Zigzag persistent homology in matrix multiplication time. In Hurtado and van Kreveld [16], pages 216–225.

[20] Dmitriy Morozov. Dionysus. http://www.mrzv.org/software/dionysus/.

[21] Dmitriy Morozov. Persistence algorithm takes cubic time in worst case. *BioGeometry News, Dept. Comput. Sci., Duke Univ*, 2005.

[22] David M. Mount and Sunil Arya. ANN, Approximate Nearest Neighbors Library. http://www.cs.sunysb.edu/ algorith/implement/ANN/implement.shtml.

[23] Harlan Sexton and Mikael Vejdemo Johansson. JPlex, 2009. http://comptop.stanford.edu/programs/jplex/.

Appendix A. The Lazy Algorithm Computes the Same Persistence Diagram

We prove that the persistence diagram we compute with the lazy algorithm is the same as the persistence diagram we compute with the standard persistence algorithm. We first prove the lemma:

Lemma A.1. *Given a filtration $F = F_1 \cdot \sigma \cdot \tau \cdot F_2$, the pairing of simplices remains unchanged in the filtration $\widetilde{F} = F_1 \cdot \tau \cdot \sigma \cdot F_2$ if σ is a creator in F and $\sigma \not\subseteq \tau$.*

Proof. Let $\mathcal{B}_1^p$ and $\mathcal{B}_2^p$ be the sets of cocycles, maintained with the annotations in the algorithm, representing the classes forming a basis of the cohomology group of dimension p after considering, respectively, the prefix F_1 and the prefix $F_1 \cdot \sigma \cdot \tau$ of the filtration ordering F. Define similarly $\widetilde{\mathcal{B}}_1^p$ and $\widetilde{\mathcal{B}}_2^p$ for, respectively, the prefix F_1 and the prefix $F_1 \cdot \tau \cdot \sigma$ of the filtration ordering $\widetilde{F}$. We first prove that σ and τ are paired the same way in F and $\widetilde{F}$.

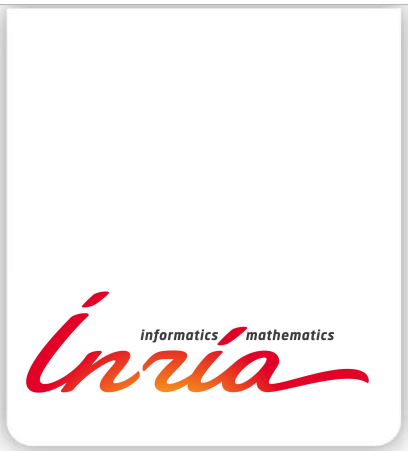
By hypothesis, σ is a creator in F thus $\mathsf{a}_{\partial\sigma} = 0$ after processing the prefix F_1 of the filtration F. We prove that $\mathsf{a}_{\partial\sigma} = 0$ after processing $F_1 \cdot \tau$ in $\widetilde{F}$. Let $\partial\sigma = \{f_1, \cdots, f_{|\sigma|}\}$ be the codimension 1 subfaces of σ. If τ is a creator or the dimension of τ is different from the one of σ, the insertion of τ does not modify the annotation vectors of the f_is and $\mathsf{a}_{\partial\sigma}$ is not changed. If τ is a killer of same dimension as σ, an annotation vector a_{f_i} may be XORed with $\mathsf{a}_{\partial\tau}$ if it contains a 1 at the index corresponding to the cocycle τ kills. The fact that $\mathsf{a}_{\partial\sigma} = \sum_{i=1\cdots|\sigma|} \mathsf{a}_{f_i} = 0$ after processing F_1 implies that for all j, $\sum_{i=1\cdots|\sigma|} \mathsf{a}_{f_i}[j] = 0$ and consequently, for a given row index j, there is an even number of 1s at index j in the annotation vectors $\mathsf{a}_{f_1} \cdots \mathsf{a}_{f_{|\sigma|}}$. Consequently, the value of $\mathsf{a}_{\partial\sigma}$ after processing $F_1 \cdot \tau$ is the value of $\mathsf{a}_{\partial\sigma}$ after processing F_1 (which is 0), XORed an even number of times with $\mathsf{a}_{\partial\tau}$, because the operation XOR is commutative. Thus $\mathsf{a}_{\partial\sigma} = 0$ after processing $F_1 \cdot \tau$ in $\widetilde{F}$. Consequently, σ is also a creator in the filtration $\widetilde{F}$. In both filtrations σ creates a cocycle ϕ^σ with time of appearence ρ_σ.
As $\sigma \not\subseteq \tau$ and σ is a creator, the annotation $\mathsf{a}_{\partial\tau}$ of the boundary of τ remains unchanged whether σ is inserted before or after τ. Consequently, τ is paired the same way in F or $\widetilde{F}$, and the cocycle it creates/kills is born/dies at the same time ρ_τ.

We finally prove that, for any p, $\mathcal{B}_1^p = \widetilde{\mathcal{B}}_1^p$ and $\mathcal{B}_2^p = \widetilde{\mathcal{B}}_2^p$. The first equality $\mathcal{B}_1^p = \widetilde{\mathcal{B}}_1^p$ is straighforward. The second equality $\mathcal{B}_2^p = \widetilde{\mathcal{B}}_2^p$ is deduced from the fact that σ and τ admits the same pairing in both filtrations, and give the same times for births and deaths of cocycles. Thus, the pairing of the simplices in F_1 and F_2 is the same in F and $\widetilde{F}$. □

We deduce:

Proposition A.2. *The lazy algorithm computes the same persistence diagram as the standard algorithm for computing persistent homology.*

Proof. The shuffle of the filtration induced by the lazy evaluation can be decomposed into successive elementary transpositions of the form $F = F_1 \cdot \sigma \cdot \tau \cdot F_2 \to \widetilde{F} = F_1 \cdot \tau \cdot \sigma \cdot F_2$, with σ creator in F and $\sigma \not\subseteq \tau$. As such a transposition does not modify the pairing nor the time of birth/death of cocycle classes (lemma A.1), the output persistence diagram is the same. □



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399