Computing multiparameter persistent homology through a discrete Morse-based approach

Sara Scaramuccia[a], Federico Iuricich[b,*], Leila De Floriani[c], Claudia Landi[d]

[a] *University of Genova, Genova, Italy*
[b] *Clemson University, Clemson (SC), USA*
[c] *University of Maryland, College Park (MD), USA*
[d] *University of Modena and Reggio Emilia, Italy*

Abstract

Persistent homology allows tracking topological features like loops, holes and their higher-dimensional analogues, along a single-parameter family of nested shapes. Currently, computing descriptors for complex data characterized by multiple parameters is becoming a major challenging task in several applications, including physics, chemistry, medicine, geography, etc. *Multiparameter persistent homology* generalizes persistent homology, thus opening the way to the exploration and analysis of shapes endowed with multiple filtering functions. Still, computational constraints prevent multiparameter persistent homology to be feasible when dealing with real-sized data. In this paper, we consider *discrete Morse Theory* as a tool to reduce the computation of multiparameter persistent homology to a smaller dataset. We propose a new preprocessing algorithm, well suited for parallel and distributed implementations, and we provide the first evaluation of the impact on computations of multiparameter persistent homology.

Keywords: persistent homology, topological data analysis, multiparameter persistent homology, Morse reductions, discrete Morse theory, homotopy expansion

1. Introduction

In recent years, the increasing amount of data available has led to the development of information handling techniques beyond machine learning approaches. Topological Data Analysis in particular provides a set of new tools for retrieving, organizing and analyzing complex data by focusing on qualitative information about their shape. Recent applications of topological data analysis in neuroscience [1, 2], image processing [3, 4, 5] and astrophysics [6] to name a few, have proven its strength and versatility.

Homology [7] is one of the most relevant tools used in topological data analysis but has the drawback of being scarcely descriptive. *Persistent homology* [8] allows for multiresolution analysis of homology by means of *filtrations*. It is used in data analysis to

*Corresponding author
Email addresses: sara.scaramuccia@dibris.unige.it (Sara Scaramuccia), fiurici@clemson.edu (Federico Iuricich), deflo@umiacs.umd.edu (Leila De Floriani), claudia.landi@unimore.it (Claudia Landi)

study evolutions of qualitative features of data and it is appreciated for its computability, robustness to noise, and dimension independence.

So far, many optimization methods for computing persistent homology have been proposed. Those more tightly related to this paper refer to another relevant tool for topological data analysis, namely *discrete Morse theory* [9]. Indeed, discrete Morse theory provides an important preprocessing tool for homology computation. By defining a *discrete gradient vector field* (also called *discrete gradient*) over the input datum, the size of the input space can be reduced to the critical parts, generally few. The discrete gradient can also be built as to preserve the filtration structure, thus enhancing also persistent homology computations via a reduction procedure. Although other persistent homology optimizations outperform this Morse-based preprocessing, these no longer apply to the generalization of persistent homology, called *multiparameter persistent homology*.

Multiparameter persistent homology is an extension of persistent homology motivated by the fact that data analysis and comparisons often involve the examination of properties that are naturally described by multiple parameters (for instance, in computer vision with respect to photometric properties).

All available multiparameter persistent homology methods suffer from high computational costs and scalability problems. This prevents them to be feasible over real-sized data sets. A Morse-based preprocessing solution, generalized to the multiparameter case, has been proposed in [10, 11]. This can have, in theory, a valuable impact on multiparameter persistent homology computations. However, that preprocessing still presents limitations in scalability with real data.

In [12] we have proposed the first algorithm capable of computing a discrete gradient on simplicial real-sized multifiltered shapes and images. We have integrated the discrete gradient into a visualization tool for studying regions of correlation in a multifield dataset, i.e., a regular grid with a vector-valued function defined on its vertexes. In this work, we extend the algorithm presented in [12] to the computation of multiparameter persistent homology. Taking [11] as-the-state-of-the-art for computing a discrete gradient for multiparameter persistent homology computation, our contributions consist of:

- a new efficient and parallel algorithm for computing a discrete gradient on multiparameter filtrations;
- a detailed analysis of complexity of the algorithm and a proof of the equivalence between our approach and the one in [11];
- a comparison of complexity and computational performances of our algorithm with respect to [11];
- an evaluation of the advantages obtained by using our algorithm as a preprocessing step in multiparameter persistent homology computations.

Our approach is well suited to be used with both simplicial complexes and regular grids, it scales well when the size of the input complex increases and is well suited for a parallel implementation. Moreover, we show that the use of the discrete gradient provides an improvement of at least one order of magnitude in the computation of multipersistent homology.

The remainder of this paper is organized as follows. In Section 2, we introduce the notions at the basis of our work. Related work is reviewed in Section 3. The new

preprocessing algorithm is described in Section 4 where we also present a detailed analysis of complexity. In Section 5, we present the proof of correctness and theoretical and experimental comparison of our approach with the one presented in [11]. The results of computing multiparameter persistent homology with our approach are discussed in Section 6. In Section 7, we draw concluding remarks and we discuss future developments.

2. Background

In this section, we introduce the notions at the base of our work. After introducing simplicial complexes we will describe two topological invariants that can be computed on them, namely homology [7] and persistent homology [8]. Then, we describe multiparameter persistent homology, a multidimensional generalization of classic persistent homology [13]. We conclude with discussing discrete Morse theory [9], a combinatorial tool of great value for (multiparameter persistent) homology.

2.1. Simplicial complexes

A simplicial complex is a discrete topological structure made of simple bricks, called simplices. A *k-dimensional simplex* σ, or k-simplex for short, is the convex hull of $k+1$ affinely independent points. Often, we will write σ^k to indicate a k-simplex. A *face* τ of σ is the convex hull of any subset of points generating σ. If the dimensions of τ and σ differ by one we call τ a *facet* of σ. Dually, σ is a *coface* of τ and a *cofacet* when the two dimensions differ by one.

A *simplicial complex* S is a finite collection of simplices such that:

- every face of a simplex in S is also in S,
- the intersection of any two simplices in S is either empty or a single simplex in S (*intersection property*).

We will denote by S_k the set of k-simplices in S. An element in S_0 is also called a *vertex*. A simplicial complex S of dimension d is a simplicial complex having the maximum of the dimensions of its simplices equal to d.

2.2. Persistent homology

Homology is a topological invariant used in data analysis to qualitatively describe shapes. The homology of a simplicial complex S detects independent k-dimensional cycles of S, i.e., connected components (0-cycles), tunnels (1-cycles), voids (2-cycles), and so on. Cycles are formally captured by linear combinations of simplices whose boundary vanishes. In this work, we focus on linear combinations over $\mathbb{F}_2$, i.e., the field with only the two elements 0 and 1.

The *chain complex* $\mathcal{C}(S) = (\mathcal{C}_*(S), \partial_*)$ associated with a simplicial complex S consists of the family $\mathcal{C}_*(S) = \{\mathcal{C}_k(S)\}_{k\in\mathbb{Z}}$ of $\mathbb{F}_2$-vector spaces along with the collection of linear maps $\partial_* = \{\partial_k : \mathcal{C}_k(S) \longrightarrow \mathcal{C}_{k-1}(S)\}_{k\in\mathbb{Z}}$ defined as follows:

- $\mathcal{C}_k(S)$ is the $\mathbb{F}_2$-vector space generated by S_k, and its elements are called *k-chains*,

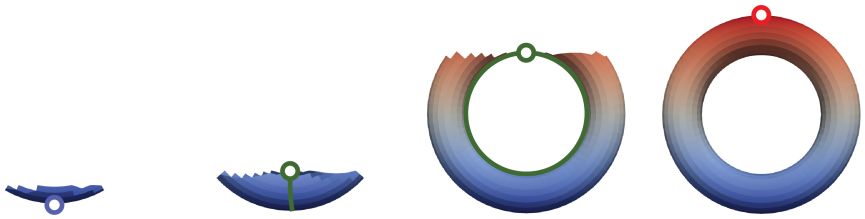


Figure 1: A filtering function defined on a torus. Blue, green, and red dots indicate the presence of components, tunnels, and voids, respectively.

- ∂_k is called *boundary map* and defined by linear extension from the images of each k-simplex τ:

$$\partial_k(\tau) = \sum_{\sigma \in S} \kappa(\tau, \sigma)\sigma,$$

with $\kappa(\tau, \sigma) = 1$ if and only if σ is a facet of τ. Elements in the kernel of ∂_k are called *k-cycles*, while elements in the image of ∂_{k+1} are called *k-boundaries*. The *k-homology* of a simplicial complex S is defined as the quotient vector space of k-cycles over k-boundaries:

$$H_k(S) = \ker \partial_k \Big/ \operatorname{im} \partial_{k+1}.$$

Persistent homology describes the homological changes occurring along an increasing sequence of simplicial complexes, called a *filtration*. We consider $\mathbb{R}$ with its usual order, and we call *grades* its elements. A *filtration* $\mathcal{S}$ of a simplicial complex S is a finite collection of simplicial subcomplexes S^u in S, indexed by $u \in \mathbb{R}$, such that for all grades $u \leqslant v$, S^u is a simplicial subcomplex in S^v. Here, we are interested in filtrations derived by the sublevel sets of functions defined on S: a *filtering function* is a function $f : S \longrightarrow \mathbb{R}$ such that, if σ is a face of τ, then $f(\sigma) \leqslant f(\tau)$.

Given a filtering function f, $\mathcal{S}(f)$ defined by setting $S^u = \{\sigma \in S | f(\sigma) \leqslant u\}$ is a filtration of S. For each pair of grades $u \leqslant v$ of the filtration $\mathcal{S}(f)$, we have that $S^u(f)$ is closed in $S^v(f)$. This means that for all $\tau \in S^u(f)$, if condition $\kappa(\tau, \sigma) \neq 0$ holds for some $\sigma \in S^v(f)$, then $\sigma \in S^u(f)$.

In Figure 1 we show a torus filtered by using the height function as filtering function. Each image illustrates a change in homology of the sublevel sets. In Figure 1(a) a new component is introduced. Two loops are created, in Figure 1(b) and (c), respectively. A void is created in Figure 1(d). For each homological class, we have a representative k-cycle appearing in the filtration. The marked blue dot is the representative 0-cycle for the new component. The two green 1-cycles are representative for the two tunnels. The set of triangles forming the entire torus surface corresponds to a 2-cycle.

The inclusion of simplicial complexes in a filtration preserves cycles and boundaries, that is, for all grades $u \leqslant v$, we get a linear map $\iota_k^{u,v} : H_k(S^u) \longrightarrow H_k(S^v)$ induced at homology level, not necessarily injective since cycles can possibly become boundaries by adding cells.

The *persistent k^{th}-homology* relative to the the grades $u \leqslant v$ is the image of $\iota_k^{u,v}$ as a subspace in $H_k(S^v)$, that is the space of all the homology classes of $H_k(S^u)$ which persist

in $H_k(S^v)$. The global persistent homology information, for all possible grades $u \leqslant v$, is encoded in the *persistence module*.

The k^{th} *persistence module* $H_k(\mathcal{S})$ of the filtered complex $\mathcal{S}$ consists of:

- the collection of $\mathbb{F}_2$-vector spaces $H_k(S^u)$ varying the grade $u \in \mathbb{R}$,
- the collection of all inclusion-induced linear maps $\iota_k^{u,v} : H_k(S^u) \longrightarrow H_k(S^v)$, varying the pairs of grades in I satisfying $u \leqslant v$ in $\mathbb{R}$.

In this work, we are interested in a generalization of persistent homology obtained by considering multiple filtrations at once. Such tool is called multiparameter persistent homology. So, from now on, we will refer to classic persistent homology as one-parameter persistent homology.

2.3. Multiparameter persistent homology

Multiparameter persistent homology analyzes the changes in homology for a *multiparameter filtration* (or *multifiltration* for short). Instead of a total order on $\mathbb{R}$, we consider a partial ordered set $(\mathbb{R}^n, \preceq)$ such that, for any $u = (u_1, \ldots, u_n), v = (v_1, \ldots, v_n) \in \mathbb{R}^n$, $u \preceq v$ if and only if $u_i \leqslant v_i$, for all $i \in \{1, \ldots, n\}$. We call *grades* the elements of the partial order set. If $u \preceq v$ and $u \neq v$, we shortly write $u \precneqq v$.

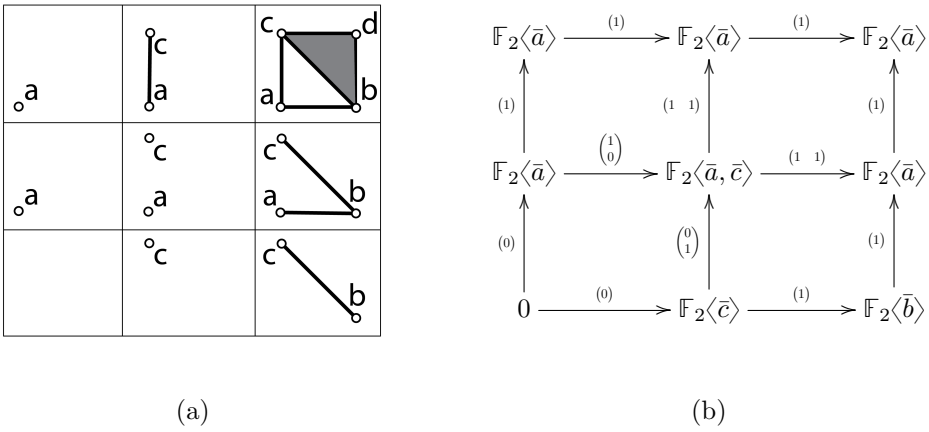
A *multiparameter filtration* $\mathcal{S}$ of a simplicial complex S is a finite collection of simplicial subcomplexes S^u indexed by $u \in \mathbb{R}^n$ such that, for all grades $u \preceq v$, S^u is a simplicial subcomplex in S^v. Multiparameter filtrations can be induced by vector-valued functions. Any function $f : S \longrightarrow \mathbb{R}^n$ satisfying $f(\sigma) \preceq f(\tau)$ for all faces σ of τ induces a multifiltration $S(f)$ by setting S^u as the set of all simplices σ satisfying $f(\sigma) \preceq u$. In this case, f is called a *(multi)filtering function* on S and S^u is called the *sublevel set* with respect to the *grade* u.

Given a multiparameter filtration of a simplicial complex S, homology construction $H_k(\cdot)$ can be applied to each sublevel set in the multifiltration. Each inclusion $S^u \subseteq S^v$ between multifiltration sublevel sets induces a linear map $\iota_k^{u,v} : H_k(S^u) \longrightarrow H_k(S^v)$.

A homology class is *persistent from grade u to grade v* if it is not trivial in $H_k(S^u)$ and still non-trivial in $H_k(S^v)$. For each homology degree k, the k^{th} *persistence module* of a multifiltration of S is the family of vector spaces $H_k(S^u)$ with $u \in \mathbb{R}^n$ along with all linear maps $\iota_k^{u,v}$ with $u \preceq v$. In Figure 2(b), we show the persistence module associated with the bifiltration depicted in Figure 2(a).

2.4. Discrete Morse theory

The algorithm we propose retrieves a combinatorial object, called a *discrete gradient*, compatible with the multiparameter filtration over a simplicial complex S. The relevance of this output has to be considered within the framework of Forman's *discrete Morse theory* [9]. A *(discrete) vector* is a pair of simplices (σ, τ) such that σ is a facet of τ. A *discrete vector field* is any collection of vectors V such that each simplex is a component of at most one vector in V. A *V-path* is a sequence of vectors (σ_i, τ_i) belonging to V, for $i = 0, \ldots, r$, such that, for all indexes $0 \leqslant i \leqslant r-1$, σ_{i+1} is a facet of τ_i and $\sigma_i \neq \sigma_{i+1}$. A V-path is said to be *closed* if $\sigma_0 = \sigma_r$, and *trivial*, if $r = 0$. A discrete vector field V is a *discrete gradient* if all of its closed V-paths are trivial. Simplices that do not belong to any vector are said to be *critical*. Given a discrete gradient V, a *separatrix* from a



(a) (b)

Figure 2: (a) A bifiltration of a 2-dimensional simplicial complex. (b) A representation of the corresponding persistence module in degree 0 (connected components). For each $\mathbb{F}_2$-vector space in the persistence module, the corresponding reference basis is made explicit, e.g., $\bar{a}$ is the homology class of the vertex a. Each matrix expresses the linear maps with respect to such bases.

critical cell τ^{k+1} to a critical cell σ^k is a V-path from any facet of τ^{k+1} to any cofacet of σ^k.

There are many ways to construct a discrete gradient on a simplicial complex. In this work, we are interested in a specific class of discrete gradients, those consistent with a multiparameter filtration. Given a multifiltration $\mathcal{S}$ of a simplicial complex S, a discrete gradient V over S is called *compatible with* $\mathcal{S}$ if, for each $(\sigma, \tau) \in V$ and each filtration grade $u \in \mathbb{R}^n$, it holds that

$$\sigma \in S^u \quad \Rightarrow \quad \tau \in S^u$$

A discrete gradient implicitly represents a cell complex, called *Morse complex* and computed by navigating the gradient V-paths. The cell of a Morse complex M of V are in one-to-one correspondence with the critical simplices of V. For any two critical simplices, the incidence between the corresponding cells in M is defined based on the following *incidence function*: given σ, τ in M, $\kappa_M(\sigma, \tau) = 1$ if and only if the number of separatrices from σ to τ is odd and $\kappa_M(\sigma, \tau) = 0$ otherwise. Analogously to the simplicial case, κ_M allows us to define a boundary map and, hence, the homology of the Morse complex.

Theorem 4.3 in [14] proves, for one-parameter filtrations, that studying the persistent homology of the Morse complex of V is equivalent to study the original simplicial complex. Corollary 3.2 in [10] proves the same result for multiparameter filtrations. From a computational point of view, a key advantage is the size of M with respect to S: M contains fewer cells than S, making the computation of the persistence module faster.

3. Related work

In this section, we review related work on the computation of persistent homology, and of multiparameter persistent homology.

3.1. Computing persistent homology

In the one-parameter case, computing the persistence module with coefficients in a field consists of reducing the *boundary matrix* via the standard algorithm [8] which has a cubic time complexity in the worst case. For this reason, new approaches have been studied to improve efficiency. We have classified such approaches into three groups: *integrated optimizations*, *annotation-based algorithms*, and *preprocessing algorithms*.

Integrated optimizations aim at improving the efficiency of the standard approach by either reducing the number of steps required for matrix reduction, or by progressively removing columns during the computation. Examples of approaches based on integrated optimizations are the *Twist* algorithm [15], the *row* algorithm [16], the approach based on sparsity presented in [17], the one based on *spectral sequences* [18], and the *chunk* algorithm [19].

Annotation-based techniques take advantage of an efficient data structure, namely the annotation matrix, to compute the persistent co-homology of a complex. An *annotation* [20] is a map which assigns a binary vector to each simplex of a simplicial complex. Each annotation provides the coordinate vectors that are used to efficiently identify the homology classes. The notion of annotation was originally introduced in [20] for computing localized homology. In [21, 22] the approach has been adapted and improved for computation of persistent homology.

Preprocessing optimizations aim at reducing the size of the input filtered complex while preserving its persistent homology. In [23], homology-preserving techniques, such as reductions, coreductions [24, 25, 26] and acyclic subspaces [27], are adapted to the case of persistent homology. Approaches rooted in discrete Morse Theory [9] compute a discrete gradient V compatible with the input filtration. The theoretical results in [14] guarantee that the Morse complex constructed from V has the same persistence module as the input complex. Many algorithms have been developed for computing a discrete gradient from a function sampled at the vertexes of a cell complex. The algorithm described in [28] is the first one to introduce a divide-and-conquer approach for computing a Forman gradient on real data. However, its main drawback is that of introducing many spurious critical simplices. Two approaches have been defined in [29, 30] for 2D and 3D images respectively. Focusing on a parallel implementation, they provide a substantial speedup in computing the discrete gradient still creating spurious critical simplices. In [31], a dimension-agnostic algorithm is proposed that processes the lower star of each vertex independently. It has been proved that up to the 3D case, the critical cells identified are in one-to-one correspondence with the topological changes in the sublevel sets, i.e. no spurious critical simplices are created. An efficient implementation of [31], focused on regular grids, is discussed in [32]. A similar approach has been developed for triangle [33] and tetrahedral meshes [34]. The first dimension independent implementation for simplicial complexes is presented in [35].

3.2. Computing multiparameter persistent homology

The first issue in computing multiparameter persistent homology is having no complete descriptors for the persistence module [13]. As a result, either the full persistence module or invariants that deliver only partial information about the multiparameter persistent homology have to be computed. The first algorithm for the persistence module retrieval is proposed in [36], where the three tasks of computing the k-boundaries,

k-cycles and their quotients at each multigrade u are translated into *submodule membership problems* in computational commutative algebra. The algorithm introduces an artifact dependency on the chosen basis, and has a time complexity of $O(|S|^4 n^3)$, where $|S|$ is the number of simplices in the complex and n is the number of independent parameters in the multifiltration. The algorithm in [37] acts on the multifiltration at the chain level rather than at homology level. First, k-cycles and k-boundaries are expressed in terms of the same basis. Then, the Smith Normal Form reduction [38, 39] is applied at each multigrade u in the multifiltration leading to a worst time complexity of $O(|S|^3 \bar{\mu}^n)$, where $\bar{\mu} := \max_{i=0,\dots,n} \mu_i$, with μ_i the number of multigrades in the multifiltration along the i^{th}-axis. The algorithm has been implemented in the *Topcat* library [40] and is distributed in the public domain.

A non-complete descriptor for multiparameter persistent homology is the *rank invariant*, introduced in [13] for each pair of multigrades $u \preceq v$ as the rank of the corresponding inclusion-induced map, that is the number of homology classes from multigrade u still persistent at multigrade v. The rank invariant value over a single pair (u, v) can be easily derived from the persistence module representation. However, computing the full rank invariant means computing its value for each possible pair (u, v) of multigrades satisfying $u \preceq v$ which multiplies the complexity by $\frac{1}{2}\mu^2$, where μ is the cardinality of all multigrades considered in the multifiltration (typically very large).

The *persistence space* [41, 42] is equivalent to the rank invariant and it enhances computational performances by avoiding to precompute the persistence module. The persistence space can be computed based on the *foliation method* [43]. With such approach, the persistence space is constructed incrementally by slicing the space of the input multiparameter filtrations and by constructing a number of one-parameter filtrations on which classic persistence homology is computed. The persistence pairs obtained on each slice form the persistence space. The first approach to computing the persistence space has been limited to the case of 0^{th}-homology [43]. An approximate version of the persistence space is proposed in [44] for two-parameter filtrations, also called *bifiltrations*: a selection of slices is performed to guarantee a fixed tolerance for the matching distance [45] among persistence spaces. This method finds applications in shape comparison in the PHOG library [46] where the authors use the approximate persistence space to deal with photometric attributes of a 3D shape.

Limitedly to bifiltrations, a visualization tool for the persistence space is RIVET [47] available online at `http://rivet.online`. The approach uses bigraded Betti numbers [48, 49] to locate multigrades where homology classes born or die. This procedure requires time $O(\mu^3\lambda)$, where λ is the product of $\lambda_x\lambda_y$ with λ_x and λ_y the number of x- and y-coordinates of such multigrades. It allows to identify an arrangement of lines such that, within a cell of the arrangement, all the filtrations have the same *barcode template*. A barcode template is constructed in $O(\mu^3\lambda + (\mu + \log\lambda)\lambda^2)$. The barcode template encode the set of bars (i.e., persistence pairs) from which to deduce that of every other filtration in the space of bifiltrations. The actual length for each bar is computed on the fly, upon request, in linear time with respect to μ.

Most optimization methods developed for classic persistent homology have not yet found a counterpart in the multiparameter case. So far, the only approach that seems still feasible is simplifying the input filtration into a new one with fewer cells and fewer grades. Limitedly to the study of 0^{th}-homology the algorithm proposed in [50] is the first approach capable of reducing the size of an input complex S without affecting its

persistence module.

The approach proposed in [10] can be seen as a Morse-based method generalizing to the multiparameter case the one proposed in [28]. The algorithm computes a discrete gradient field having the same persistence module of the input complex. Like its one-parameter counterpart [28], it suffers from introducing many spurious critical simplices. In a successive paper [11] a new approach is introduced generalizing the idea of [31] of constructing the discrete gradient locally inside the lower star of simplices of S. The resulting discrete gradient is proved to induce a Morse complex with the same persistence module, and thus the same persistence space, as the original multifiltration. However, the algorithm requires a global ordering of all the simplices of S and does not scale with the size of the data making it not feasible for real applications. In Section 5, we will further discuss this issue compared with our approach that can be seen as a divide-and-conquer generalization of [11].

4. Local computation of a discrete gradient over a multiparameter filtration

In this section, we present a new algorithm for computing a discrete gradient vector field compatible with a multiparameter filtration. For ease of exposition, we describe the method by focusing on simplicial complexes, although it is valid for any cell complex satisfying the intersection property such as *cubical complexes*.

In Section 4.1 we provide a high-level description of the algorithm workflow. A detailed description of the auxiliary functions used is provided in Section 4.2, while in Section 4.3 we discuss the algorithm complexity.

4.1. Overview

The proposed algorithm receives a multiparameter filtration in input and produces a compatible discrete gradient. In what follows, we describe the input multiparameter filtration as a pair (S, f), where S is a d-dimensional simplicial complex and $f : S_0 \longrightarrow \mathbb{R}^n$ is a vector-valued function defined on the vertexes of S. f is extended to all simplices of S by $f(\sigma) = \max_{v \in \sigma} f_i(v)$, for each $0 \leqslant i \leqslant n$, and induces the multifiltration that we denote as $S(f)$, as described in Section 2.3.

Without loss of generality, we require the function f to be component-wise injective. In applications, any function can be transformed into a component-wise injective one by means of simulation of simplicity [51]. The output is a pair (V, M), where V is the set of paired simplices of S and M is the set of critical (unpaired) simplices. When proving correctness, we will show that V is a discrete gradient compatible with the filtration, and that M contains the cells of its Morse complex.

The main strategy of the algorithm is that of decomposing S according to f as to compute pairings between simplices with the same multigrade, possibly in parallel.

The algorithm consists of three main steps: *vertex-based decomposition*, *multigrade grouping*, and *pairing*. A running example is depicted in Figure 3.

In the firs step we decompose S to obtain a partition of the simplices in S. In this step, we only require that simplices belonging to the same multigrade also belong to the same group in the decomposition. This is performed by algorithm `ComputeDiscreteGradient` (Algorithm 1) as follows:

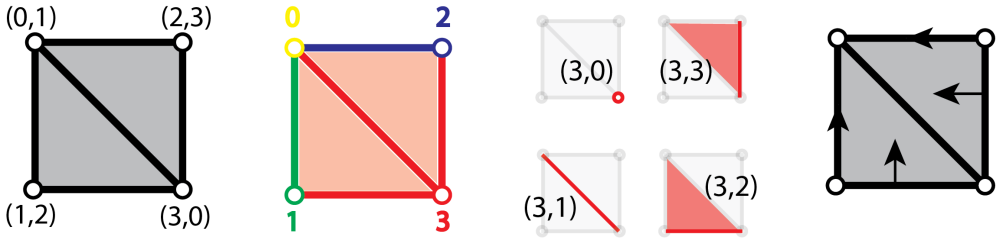


Figure 3: (a) A multiparameter filtration (S, f) where S is a triangle mesh. (b) Simplices of S are color coded accordingly to their values in the indexing schema I. (c) Within the index-based lower star of $\mathrm{Low}_I(3)$, simplices are subdivided and paired based on their value of f. (d) The final discrete gradient V.

- an indexing $I : S_0 \longrightarrow \mathbb{R}$ is computed for the vertexes of S (line 2). The indexing is extended to the other simplices $\sigma \in S$ by setting $I(\sigma) := \max_{v \in \sigma} I(v)$. The indexing used must be *well-extensible*, i.e., I must satisfy, for all simplices $\sigma, \tau \in S$, the following property:

$$f(\sigma) \preceq f(\tau) \qquad \Rightarrow \qquad I(\sigma) \leqslant I(\tau). \tag{1}$$

- S is subdivided into lower stars according to I (line 4). We recall that the star of a simplex σ, denoted by $Star(\sigma)$, is defined as the set of its cofaces. Then, the *index-based lower star* of a simplex σ, denoted as $\mathrm{Low}_I(\sigma)$, is defined as the set of its cofaces having a value of I lower or equal to σ. Formally,

$$\mathrm{Low}_I(\sigma) := \{\tau \in \mathrm{Star}(\sigma) \mid I(\tau) \leqslant I(\sigma)\}.$$

In Figure 3(b) simplices belonging to the same index-based lower star are depicted with the same color. The well-extensible indexing I and the index-based lower star Low_I satisfy two fundamental properties:

- each simplex σ belongs to the index-based lower star of exactly one vertex (see Lemma 1 in Section 5);
- if two simplices have the same value of f, then they belong to the index-based lower star of the same vertex (see Lemma 2 in Section 5).

As a consequence, a well-extensible indexing and the index-based lower stars implicitly provide a partition for the domain S. Thanks to the former properties we can guarantee that, by processing the vertexes independently, we are identifying all the valid pairings (see Section 5 for a formal proof).

The second step requires grouping the simplices in the set $L_I = \mathrm{Low}_I(v)$, with $v \in S_0$, having the same multigrade (line 5). The latter is performed by `SplitIndexLowerStar`, which organizes the simplices in a collection of sets, where each set L_u contains simplices with the same multigrade u, i.e., $L_u = \{\sigma \in \mathrm{Low}_I(v) | f(\sigma) = u\}$, $u \in \mathbb{R}^n$. In Figure 3(c), simplices belonging to the index based lower star $\mathrm{Low}_I(3)$ are subdivided based on their value of f.

Algorithm 1 ComputeDiscreteGradient(S, f)

Input: S, a simplicial complex
Input: $f : S_0 \longrightarrow \mathbb{R}^n$, a component-wise injective function
Output: V, list of simplices pairs
Output: M, list of critical simplices

1: define V and M as two empty lists
2: $I \leftarrow$ `ComputeIndexing`(S_0, f)
3: **for all** v in S_0 **do**
4: $\quad L_I \leftarrow$ `ComputeIndexLowerStar` (v, I, S)
5: $\quad$ **for all** L_u in `SplitIndexLowerStar`(f, L_I) **do**
6: $\quad\quad (V_u, M_u) \leftarrow$ `HomotopyExpansion`(S, I, L_u)
7: $\quad\quad$ add V_u to V
8: $\quad\quad$ add M_u to M
9: **return** (V, M)

In the third step (lines 6), each level set L_u is independently processed by `HomotopyExpansion` and the gradient pairs are computed. Paired and critical simplices found in the level set L_u will contribute to the final discrete gradient. Figure 3(d) shows the final discrete gradient V computed by our approach on the multiparameter filtration depicted in Figure 3(a). Since simplices are subdivided based on their multigrade, each simplex S appears in exactly one level set and it will be classified, as either paired or critical, only once. This makes the approach easily parallelizable.

4.2. Detailed description

This section provides additional information about the auxiliary functions we use in Algorithm 1 following their order of appearance in the algorithm description.

The first auxiliary function is `ComputeIndexing` which is used for computing a well-extensible indexing on the vertexes of S. There are many ways to obtain a well-extensible indexing I, here we have selected to sort all the vertexes of S according to the values of the first component of f. The total order obtained naturally generates an indexing which is guaranteed to be well-extensible as, for each pair of simplices σ and τ, $f(\sigma) \preceq f(\tau)$ implies $f_1(\sigma) \preceq f_1(\tau)$. Thus, a vertex $v \in \tau$ exists such that $f_1(v) \geqslant f_1(w)$ for every vertex $w \in \sigma$. This implies $I(v) \geqslant I(w)$, for every vertex $w \in \sigma$ and we conclude that $I(\sigma) \leqslant I(\tau)$.

`ComputeIndexLowerStar` is used for computing the index-based lower star $\mathrm{Low}_I(v)$ of a vertex v with respect to indexing I. For each vertex $v \in S_0$ the function extracts the set of simplices of $\mathrm{Low}_I(v)$ incident to v. To do so, we assume that each k-simplex σ is represented by the list of its $k+1$ vertexes $[v_0, v_1, \ldots, v_k]$ stored in decreasing order of I, i.e. $I(v_0) > I(v_1) > \cdots > I(v_k)$. Thus it is enough to take those simplices whose first vertex is v.

Each index-based lower star, $\mathrm{Low}_I(v)$ is then subdivided into level sets by `SplitIndexLowerStar` according to multigrades. This function creates an associative array, by cycling on the simplices of $\mathrm{Low}_I(v)$, that maps each multigrade u (represented as a vector of floats) to the set of simplices sharing the same multigrade.

Algorithm 2 HomotopyExpansion(S, I, L_u)

Input: S, a simplicial complex, I, a well-extensible indexing, L_u, a list of cells in S forming a level set u w.r.t. f
Output: V_u list of discrete vectors, M_u list of critical simplices

```
1: define V_u and M_u two empty lists
2: define Ord0 and Ord1 two empty ordered sets
3: define declared an array of length |L_u| with Boolean values equal to false
4: for all τ in L_u do
5:   if num_undeclared_facets(τ, L_u)= 0 then
6:     insert(τ,Ord0,I)
7:   else if num_undeclared_facets(τ, L_u)= 1 then
8:     insert(τ,Ord1,I)
9: while Ord1≠ ∅ or Ord0≠ ∅ do
10:  while Ord1≠ ∅ do
11:    τ ← delete(Ord1)
12:    if num_undeclared_facets(τ, L_u)= 0 then
13:      insert(τ,Ord0,I)
14:    else
15:      ρ ← unpaired_facet(τ, L_u)
16:      append((ρ, τ),V_u)
17:      declared[ρ] ←true , declared[τ] ←true
18:      add_cofacets(ρ,L_u,I,Ord1)
19:      add_cofacets(τ, L_u,I,Ord1)
20:  if Ord0≠ ∅ then
21:    τ ← delete(Ord0)
22:    append(τ,M_u)
23:    declared[τ] ← true
24:    add_cofacets(τ, L_u,I,Ord1)
25: return (V_u, M_u)
```

Function `HomotopyExpansion` classifies simplices with the same multigrade. We present its pseudocode in Algorithm 2. The algorithm extends the one in [31]. A k-simplex σ and a $(k+1)$-simplex τ are considered *pairable* only when σ is the only unclassified facet of τ. The main objective of `HomotopyExpansion` is that of pairing as many simplices as possible and to classify them as critical only when no pairable simplices are available.

Two ordered sets `Ord0` and `Ord1` are used to keep track of those simplices that have exactly zero unpaired facets and one unpaired facet, respectively. Intuitively, simplices in `Ord0` are candidates to be classified as critical or as tails of arrows in a discrete vector, since they have no face to be paired with. Simplices in `Ord1` are the candidate to be heads of arrows in a discrete vector. Within the two sets `Ord0` and `Ord1`, a simplex σ is represented by the sequence, in decreasing order, of the values of I on its vertexes. The two sets are organized based on the lexicographic ordering of such sequences. Auxiliary function `insert` is used to compute the corresponding sequence for any simplex σ and to insert σ in a set. Auxiliary function `delete` is used to extract

the first simplex σ in the set, according to the lexycographic order. The two sets are initialized by cycling on the simplices in the input set (lines 4 to 8 of Algorithm 2). Auxiliary function `num_undeclared_facets` is used to count the number of unclassified facets for each simplex. Array `declared` keeps track of the simplices already classified (i.e., either paired or declared critical). At the beginning, all entries of `declared` are set to `false`.

Simplices are classified within the two nested while loops (lines 9 to 24). If `Ord1` is not empty, we extract the first simplex τ and we verify if the number of unclassified facets of τ has not changed (line 12). Notice that the number of unpaired facets can only decrease. If this number is now zero (i.e., its facet has been classified), we insert τ into `Ord0`. Otherwise, we retrieve its unique unclassified facet σ (line 15), we use function `append` to add (σ,τ) to the set of pairs V_u, and we update the array `declared` accordingly. After classifying σ and τ, `add_cofacets` adds all their unclassified cofacets to either `Ord1` or `Ord0`, if they have the necessary number of unclassified facets (lines 18 and 19).

When no pairable simplex is available (i.e., `Ord1` is empty) the first simplex σ in `Ord0` is extracted and declared critical by adding σ to the set of critical simplices M_u (lines 21 to 23). All its cofacets are processed and added to `Ord1` if it is the case. The algorithm stops when both lists are empty. In Proposition 4 in [31], authors show that we exit the outer while loop when all cells have been classified.

4.3. Complexity

In this section, we discuss the computational complexity of `ComputeDiscreteGradient` and its auxiliary functions. The parameters involved in the analysis are expressed in terms of cardinality $|\cdot|$ of sets. We indicate with $\mathrm{Star}(\sigma)$ the star of a simplex $\sigma \in S$, and simply with Star any star with maximal cardinality in the simplicial complex S. Notice that, in a d-dimensional simplicial complex, $|\,\mathrm{Star}\,|$ is not bounded by a constant and is possibly as large as $|S|$. This is not the case for more regular cell complexes like, for example, cubical complexes.

To simplify the analysis and the exposition we make a few assumptions:

- for each simplex $\sigma \in S$, we assume $\mathrm{Star}(\sigma)$ to be computed and stored in the data structure. If computed on the fly, $\mathrm{Star}(\sigma)$ would require $O(|\,\mathrm{Star}(\sigma)|)$ [52].
- `Ord0`, `Ord1` are implemented as balanced binary search trees. Inserting, or removing some element from such a tree has a logarithmic cost in its size.
- For each k-simplex $\sigma \in S$, $f(\sigma)$ can be retrieved in $O(k)$ by retrieving the filtration values of the vertexes of σ. We will overestimate k by considering the dimension d of the simplicial complex S.

These assumptions are consistent with the implementation of `ComputeDiscreteGradient` used in our experimental evaluation (see Section 5.2.1).

4.3.1. Analysis of the auxiliary functions

Here, we present the time and storage costs of the auxiliary functions introduced in Section 4.2.

For creating the well-extensible indexing with `ComputeIndexing` we sort the vertexes according to a single component of the input function. This requires $O(|S_0| \log |S_0|)$ time and $O(|S_0|)$ extra space for storing the new sorted list of vertexes.

The lower star of each vertex v is computed by `ComputeIndexLowerStar`. The lower star of a vertex is extracted from the precomputed star $\mathrm{Star}(v)$ by selecting those simplices having v as the first vertex. This requires $O(|\mathrm{Star}(v)|)$ operations.

Once a lower star is extracted, the level sets are created by means of `SplitIndexLowerStar`. The filtration value of a simplex σ is computed, for each component, in linear time in the number of vertexes of σ by $f_i(\sigma) = \max_{v \in \sigma} f_i(v)$. Searching for the set of simplices associated with a specific level set takes at most $O(\log |\mathrm{Low}_I(v)|)$. Thus, the overall cost of `SplitIndexLowerStar` is $C_{LS}(v) = O(|\mathrm{Low}_I(v)|(d + \log |\mathrm{Low}_I(v)|))$.

In the third step, `HomotopyExpansion` classifies the simplices in each level set L_u. For each simplex σ in L_u, `num_undeclared_facets` requires visiting its facets whose number is limited from above by a constant and inserting each simplex in the set takes $O(\log |L_u|)$. Then, preparing `Ord1` and `Ord0` requires $O(|L_u| \log |L_u|)$.

Within the two while loops, each simplex enters a list at most once and it is also classified once. Then, for each simplex σ:

- retrieving its facets (`num_undeclared_facets` or `unpaired_facets`) requires a constant number of operations,
- retrieving its cofacets (`add_cofacets`) takes at most $O(|L_u|)$ as the number of cofacets is not limited by any constant number,
- inserting the simplex in L_u takes $O(\log |L_u|)$.

Overall the contribution of `HomotopyExpansion` is $C_{HE}(L_u) = O(|L_u|^2 + 2|L_u| \log(|L_u|)) = O(|L_u|^2)$

4.3.2. Analysis of `ComputeDiscreteGradient` algorithm

By analyzing the complexity of the single auxiliary functions we obtain a worst-case complexity of

$$O\left(|S_0| \log |S_0| + \sum_{v \in S_0} \left(|\mathrm{Star}(v)| + C_{LS}(v) + \sum_{L_u \subseteq \mathrm{Low}_I(v)} C_{HE}(L_u) \right)\right)$$

In the internal summation, L_u can be as large as the entire index-based lower star. Thus, for any vertex v, we can estimate $O\left(\sum_{L_u \subseteq \mathrm{Low}_I(v)} C_{HE}(L_u)\right) = O(|\mathrm{Low}_I(v)|^2)$.

Each k-simplex appears in the star of its $k+1$ vertexes. Thus, we write $O\left(\sum_{v \in S_0} |\mathrm{Star}(v)|\right)$ $= O(|S|(d+1))$ by overestimating the dimension k of each simplex with the dimension of the complex d.

Each simplex appears in exactly one index-based lower star. Thus, we can estimate the worst-case complexity of `SplitIndexLowerStar` by $O\left(\sum_{v \in S_0} C_{LS}(v)\right) = O(|S|(d + \log(\max_{v \in S_0} |\mathrm{Low}_I(v)|))$. For the same reason we can estimate the worst-case complexity for `HomotopyExpansion` by $O\left(\sum_{v \in S_0} C_{HE}(v)\right) = O(\max_{v \in S_0} |\mathrm{Low}_I(v)|^2)$. Moreover, we

notice that $\max_{v \in S_0} |\operatorname{Low}_I(v)| \leqslant |\operatorname{Star}|$.

Based on these observations we can rewrite the overall worst-case complexity as

$$O(|S_0| \log |S_0| + |S|(d + |\operatorname{Star}|^2))$$

In general, the cardinality of a vertex star is not bounded from above. There are cases where in practice its size becomes negligible with respect to the total number of simplices in S, i.e., when working with low dimensional complexes such as triangle meshes or 2D and 3D images. In those cases, we can consider d and $|\operatorname{Star}|$ to be constant factors which lead to a worst case complexity of $O(|S_0| \log |S_0| + |S|)$.

5. Comparison with the `Matching` algorithm and proof of correctness

The `Matching` algorithm introduced in [53], and fully proved to be correct in [11], computes a discrete gradient on a multifiltration by using a global queue. In this Section, we compare algorithm `ComputeDiscreteGradient` to algorithm `Matching`. In Subsection 5.1 we first describe algorithm `Matching`. In Subsection 5.2 we provide a formal comparison of the two approaches and in Subsection 5.2.2, we prove their equivalence, i.e., given a multifiltration, `ComputeDiscreteGradient` and `Matching` compute the same compatible discrete gradient.

5.1. Globally computing a discrete gradient for multiparameters

In this subsection, we describe the approach applied by algorithm `Matching` [53, 11] to retrieve a discrete gradient. The `Matching` algorithm acts on a simplicial complex S and a function $f : S_0 \longrightarrow \mathbb{R}^n$ required to be component-wise injective on vertexes and extended to higher dimensional simplices as defined in Section 4. The pair (S, f) defines a multifiltration of S obtained by sublevel sets. Additionally, the `Matching` algorithm requires an *indexing* J on S, i.e., an injective map $J : S \longrightarrow \mathbb{R}$. The indexing has to be compatible both with the coface partial order among simplices and with the value ordering under f. Explicitly, J has to satisfy, the following property for every $\sigma \neq \tau \in S$,

$$\sigma \text{ is a face of } \tau \text{ or } f(\sigma) \precneqq f(\tau) \quad \Rightarrow \quad J(\sigma) < J(\tau). \tag{2}$$

The algorithm cycles on all simplices in S relying on a global queue. Simplices are processed according to increasing values of the indexing J. This makes impossible implementing the approach in parallel.

An auxiliary Boolean vector of length $|S|$, called `classified`, is initialized with all entries set to `false`. For each simplex σ, algorithm `Matching` recursively checks whether σ is classified so that only non-classified simplices are processed. We denote by P the set of all simplices $\sigma \in S$, also called *primary simplices*, that are still unclassified when `Matching` starts processing their lower star. A non-classified simplex σ is passed to an auxiliary function extracting the lower star of σ with respect to f

$$\operatorname{Low}_f(\sigma) := \{\tau \in \operatorname{Star}(\sigma) \mid f(\tau) \preceq f(\sigma)\}.$$

To do so, the auxiliary function visits $\operatorname{Star}(\sigma)$ to select each simplex τ satisfying condition $f(\tau) \preceq f(\sigma)$. Afterwards, an auxiliary function equivalent to the one in the algorithm `HomotopyExpansion` (pseudocode reported in Algorithm 2) is run with input $(\operatorname{Low}_f(\sigma), J)$.

Algorithm `HomotopyExpansion` returns a pair of lists $(V_{\mathrm{Low}_f(\sigma)}, M_{\mathrm{Low}_f(\sigma)})$ and all entries in the auxiliary vector `classified` corresponding to the simplices in the two lists are set to `true`. The global output is given by the independent contributions of all pairs $(V_{\mathrm{Low}_f(\sigma)}, M_{\mathrm{Low}_f(\sigma)})$.

Theorem 9 in [11] proves that the union of all the discrete gradients, computed locally to the lower star $\mathrm{Low}_f(\sigma)$ of each simplex σ, forms a discrete gradient. Moreover, Proposition 8 in [11] proves that the same gradient is compatible with the input multifiltration (S, f). In particular, Proposition 15 in [11] directly implies that lower stars $\mathrm{Low}_f(\sigma)$ for primary simplices $\sigma \in P$ form a partition of S.

5.2. Comparison of algorithms `Matching` and `ComputeDiscreteGradient`

In this section, we compare `ComputeDiscreteGradient` to `Matching` from three different standpoints: complexity, performance, and quality of output. We recall that P indicates the primary simplices in `Matching`, i.e., those simplices S that were still unclassified when `Matching` starts processing their lower star. Both `ComputeDiscreteGradient` and `Matching` build a discrete gradient by running `HomotopyExpansion` on a partition of the input simplicial complex S:

- `Matching` finds the discrete gradient over each lower star $\mathrm{Low}_f(\sigma)$ with σ a primary simplex of P,
- `ComputeDiscreteGradient` finds the discrete gradient independently over each level set L_u in a index-based lower star $\mathrm{Low}_I(v)$ with $v \in S_0$.

The two algorithms differ in the number of stars they compute and visit. `ComputeDiscreteGradient` computes a lower star for each vertex in the input complex. This means that, in our case, the exact number of star visited by `ComputeIndexLowerStar` is $|S_0|$.

On the countrary, `Matching` computes a lower star for each primary simplex in P. In the best case, the primary simplices P are exactly the vertices of S, i.e., $|P| = |S_0|$. Then, the two algorithms compute the same number of lower stars. In the worst case, the number of primary simplices $|P|$ is equal to the total number of simplices $|S|$ which greatly affects the performances of `Matching` as shown in Subsection 5.2.1.

5.2.1. Performance comparison

We recall that the input of both algorithms is described as a pair (S, f), where S is a simplicial complex and $f : S_0 \longrightarrow \mathbb{R}^n$ is a component-wise injective function. Here, we focus on the case where S is a triangle mesh embedded in the Euclidean 3D space and f assigns to each vertex its x and y coordinates (i.e., for $v = (x, y, z)$, $f(v) = (x, y)$). The algorithm in [11] is defined to work in combination with a data structure that encodes all the simplices of a simplicial complex S. This approach does not scale when the size and the dimension of the simplicial complex S grows. The only way to guarantee scalability is that of encoding S with an indexed-based representation based on its top simplices [35]. For a fair comparison, both algorithms have been implemented by using the *FG_Multi library* [54] which provides a compact encoding for the triangle mesh as well as for the discrete gradient.

Simplicial complex representation. A triangle mesh S is a simplicial complex of dimension two formed by vertexes, edges, and triangles. The FG_Multi library implements an incidence-based data structure for compactly encoding the relations among these simplices. Vertexes and triangles are the only simplices which are explicitly encoded for a total of $|S_0| + |S_2|$ entities. Each vertex encodes the list of triangles incident, while each triangle encodes a reference to its three vertexes. Notice that, for each triangle σ referencing a vertex v, we also have that v references σ. Thus, since each triangle references three vertexes, the triangle-vertex relation costs $3|S_2|$ while the vertex-triangle relation doubles this cost leading to a total of $7|S_2| + |S_0|$. The filtering function f is stored for each vertex by encoding a vector of floating point values, one value for each component of f.

Discrete gradient representation. The discrete gradient V is encoded by adopting the representation described in [33]. The latter focuses on encoding all the gradient pairs locally to each triangle. The encoding uses the following rationale. Since each triangle σ can be paired with at most three edges and each edge can be paired with two vertexes, locally for each triangle we have 9 possible pairs. If we consider also the possible pairs between an edge and an adjacent triangle we get 12 possible gradient pairs and thus $2^{12} = 4096$ possible combinations. However, a discrete gradient imposes certain restrictions, i.e. that each simplex can be involved in at most one pairing. As a consequence, we have only 97 valid cases for a triangle. These cases can be encoded using only 1 byte per triangle and, thus, encoding the gradient only requires $|S_2|$ bytes. Notice that the latter approach has been generalized to tetrahedral meshes [34] and d-dimensional simplicial complexes [35].

The datasets used in this comparison are originated by three triangle meshes. The experiments have been performed on a dual Intel Xeon E5-2630 v4 CPU at 2.20 GHz with 64GB of RAM. For each mesh, we obtain two refined versions of the latter by recursively applying Catmull-Clark algorithm [55] to it. The nine triangle meshes are described in Table 1. Column *Original* indicates the number of simplices composing the mesh. Column *Critical* indicates the number of unpaired (critical) simplices identified by both reduction approaches, while the resulting compression factor is reported in column *Original/Critical*.

Dataset	Parameters	Simplices		Compression factor
		Original	Critical	Original/Critical
Torus	2	1.3M	0.035M	37.9
		5.3M	0.11M	45.3
		21.5M	0.77M	27.7
Sphere	2	2.9M	0.28M	10.2
		11.7M	0.11M	10.2
		47.1M	0.46M	10.1
Gorilla	2	3.8M	0.4M	9.5
		15.2M	1.6M	9.4
		60.9M	6.4M	9.4

Table 1: Datasets used for the experiments. For each of them, we indicate the number of independent parameters in the multifiltration (column *Parameters*), the number of simplices in the original dataset (column *Original*), number of critical simplices retrieved by `ComputeDiscreteGradient` and `Macthing` (column *Critical*) and the compression factor (column *Original/Critical*).

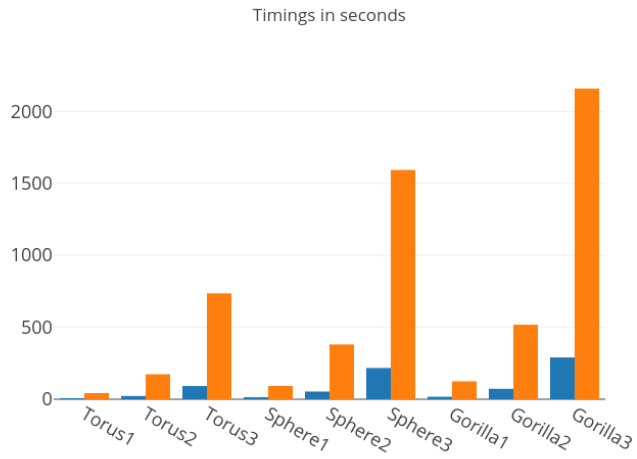


Figure 4: Timings required by `ComputeDiscreteGradient` (in blue) and `Matching` (in orange).

Timings are shown in Figure 4. `ComputeDiscreteGradient` takes between 0.89 seconds and 4.8 minutes depending on the dataset and it is generally 7 times faster than our efficient implementation of `Matching`. Time performances show the practical efficiency of the local approach compared to the global one. As seen in Section 4.3, the expected asymptotical complexity over a triangle mesh S is linear in the number of simplices for both algorithms when the number of simplices within each star is negligible. In Figure 5, we show the trends as the number of simplices increases. This confirms our argument that the number of stars to be retrieved and visited has a direct consequence on the algorithm performances. In `ComputeDiscreteGradient` we need to process each vertex star only once while `Matching` requires processing a star for each grade.

Other than time efficiency, our strategy also requires a limited use of memory. The memory consumption is shown in Figure 6, where we are reporting the maximum peak of memory used by the two algorithms. `ComputeDiscreteGradient` uses up to 10 gigabytes of memory versus more than 20 of `Matching`. The storage cost difference between the two implementations grows linearly in the number of simplices in the datasets. The advantage is due to the different memory consumption at runtime. Specifically, `Matching` stores a global queue over all the simplices in the dataset and needs to track all classified simplices. Both these steps are performed locally, over each level set, by `ComputeDiscreteGradient`.

5.2.2. Equivalence of the outputs

We now show that the discrete gradient computed by `ComputeDiscreteGradient` can be made coincide with that of `Matching` provided that the indexing I used in `ComputeDiscreteGradient` is well-extensible and the filtering function f is componentwise-injective on the vertices. These assumptions will be maintained throughout this section. Here, we first state the results required to draw such conclusions, while the detailed proofs can be found in Appendix.

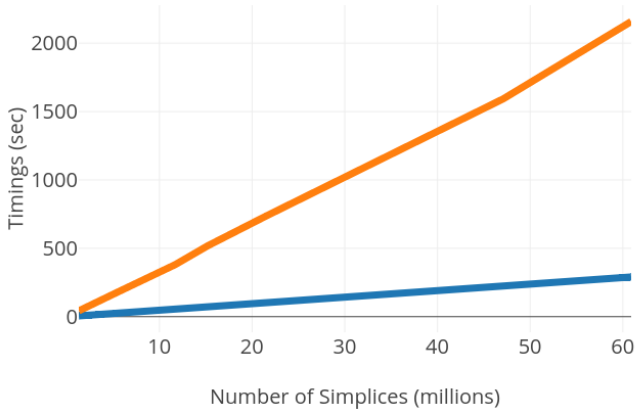


Figure 5: Trend in the timings for `ComputeDiscreteGradient` (in blue) and `Matching` (orange).

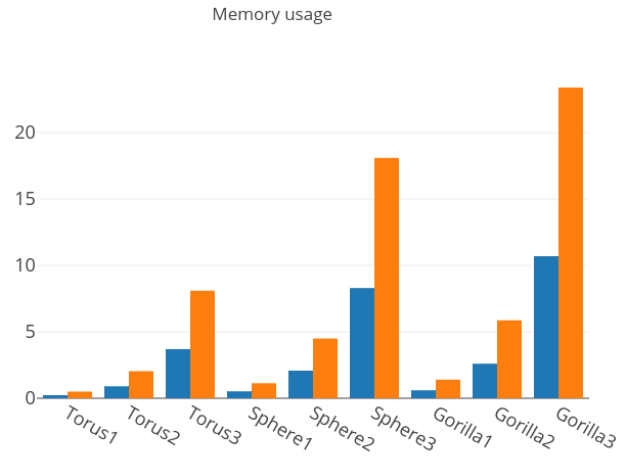


Figure 6: Maximum peaks of memory (in gigabytes) required by `ComputeDiscreteGradient` (in blue) and `Matching` (in orange).

The proof has two parts. First, we show that the two algorithms apply `HomotopyExpansion` to the same partition of the input simplicial complex S. Then we prove that, given any valid order for the simplexes used in `HomotopyExpansion` by one of the two algorithms, it is possible to construct a valid order for the other algorithm so that the output of the two is the same. This provides, as a byproduct, the correctness of `ComputeDiscreteGradient`.

We start proving that the partitioning strategies of `ComputeDiscreteGradient` and

`Matching` are the same. In `ComputeDiscreteGradient` we partition S according to the level sets L_u of f in $\mathrm{Low}_I(v)$, for all vertices v and grades u. In `Matching` S is partitioned according to $\mathrm{Low}_f(\sigma)$, with σ varying in the set of primary simplexes P, i.e. simplices that do not belong to the lower star with respect to f of any other simplex.

We start showing that the partition provided by sets $\mathrm{Low}_f(\sigma)$, with σ in P, is a refinement of the one provided by the sets $\mathrm{Low}_I(v)$ with $v \in S_0$.

Lemma 1. For every $\sigma \in S$, and, hence, in particular for $\sigma \in P$, there is exactly one vertex $v \in S_0$ such that $\mathrm{Low}_f(\sigma) \subseteq \mathrm{Low}_I(v)$.

Next, we show that, for every vertex v, any level set L_u of the filtering function f restricted to $\mathrm{Low}_I(v)$ coincides with the lower star $\mathrm{Low}_f(\sigma)$ of some primary simplex σ.

Lemma 2. Let L_u be a non-empty level set of f in $\mathrm{Low}_I(v)$ for some grade $u \in \mathbb{R}$ and some vertex $v \in S_0$. There exists a unique simplex $\sigma \in L_u$ such that $\mathrm{Low}_f(\sigma) = L_u$. Moreover, $\sigma \in P$.

Lemmas 1 and 2 allow us to conclude that both `ComputeDiscreteGradient` and `Matching` run `HomotopyExpansion` on the same subsets $L_u = \mathrm{Low}_f(\sigma)$. Now, to conclude that they give the same output, we need to show that `HomotopyExpansion` does so by processing the simplexes of these subsets in the same order.

Proposition 3. For every valid input (S, f) for `ComputeDiscreteGradient`, there exists an indexing $J : S \longrightarrow \mathbb{R}$ valid for `Matching` such that the output of `Matching`(S, f, J) and `ComputeDiscreteGradient`(S, f) coincide.

As a consequence of this result, from the correctness of `Matching`, we obtain the correctness of `ComputeDiscreteGradient`.

Corollary 4. Algorithm `ComputeDiscreteGradient` with input (S, f) returns a discrete gradient compatible with the multifiltration induced by f on S.

6. Computing multiparameter persistence homology

In this section, we evaluate the impact of our algorithm as a preprocessing method for the computation of the persistence module (Subsection 6.1) and of the persistence space (Subsection 6.2). Before presenting our results, we describe how to extract, from a discrete gradient V, the Morse complex that will be used as input of persistence computations.

Computing the Morse complex. We recall from Section 2.4 that a discrete gradient V implicitly represents a Morse complex M consisting only of the critical simplices of V with the incidence relations given by the separatrices originating and having destination in a pair of critical simplices.

In order to compute such incidence relations, we process all the critical simplices of V as in [35]. For each critical simplex σ of dimension k, a breadth-first traversal is performed as follows. From σ we extract its facets. For each $(k-1)$-simplex τ_1 we extract its paired k-simplex σ_1, if any. We apply the same rationale to σ_1 to continue the visit. As soon as we encounter a $(k-1)$-simplex τ which is unpaired (critical) we classify the two simplices σ and τ as mutually incident.

In the worst case, computing the incidences for a single critical simplex of dimension k requires visiting all k-simplices multiple times, in the order of $O(|S_k|^2)$, where $|S_k|$ is the number of k-simplices in S. Having a number of critical simplices of the same order of $|S_k|$ would bring the total worst-case complexity to be cubical in the number of simplices. In practical cases, the extraction of the Morse complex is very efficient as each k-simplex belongs to a very limited set of gradient paths, possibly zero.

6.1. Computing the persistence module

The persistence module is computed by means of the open-source library *Topcat* [40]. Due to its strong limitations in terms of time and memory costs, we used simplified datasets for our experiments. We use six triangle meshes of limited size, three representing a torus and three representing a sphere. We use a bifiltration for each mesh defined by the x and y coordinates of the vertexes. Table 2 presents a description of the input dataset. The number of simplices (column *Simplices*) and number of multigrades (column *Grades*) are reported for each simplicial complex (column *Original*) and each Morse complex (column *Morse*). Notice that for a Morse complex M, column *Critical Simplices* indicates the number of critical simplices in the discrete gradient V which is is also equivalent to the number of cells in the corresponding Morse complex M.

	Original				**Morse**				**Time**
			Persistence Module		**Critical**		**Persistence Module**		
Dataset	**Simplices**	**Grades**	**Time**	**Memory**	**Simplices**	**Grades**	**Time**	**Memory**	
Sphere	38	8x8	0.3	0.24	4	5x5	0.18	0.1	0.0264
	242	42x42	4.4	0.86	20	10x10	0.28	0.2	0.0244
	2882	482x482	-	-	278	92x89	24.3	1.5	0.0473
Torus	96	16x16	0.5	0.1	8	9x9	0.25	0.2	0.0255
	4608	768x768	-	-	128	65x66	7.96	2.4	0.0643
	7200	1200x1200	-	-	156	70x80	12.05	3.0	0.0815

Table 2: Timings (in seconds) and storage costs (in gigabytes) for the persistence module retrieval over the original (columns *Original*) and the corresponding Morse complex (columns *Morse*). Columns *Simplices* and *Grades* reports the number of simplices in the dataset and the number of level sets along each parameter, respectively. Missing entries indicate where the Topcat library runs out of memory. Column *Time* explicits the timings (in seconds) for obtaining the Morse complex using `ComputeDiscreteGradient`.

The *Topcat* library uses the boundary matrices of the complex to compute the persistence module. Since it was designed to accept only multifiltrations defined on simplicial complexes, we have modified the library for working on multifiltrations defined over more general cell complexes, like the Morse complex.

We compute the persistence module, for each homology grade, of both the original simplicial complex and the Morse complex generated by our algorithm, and we measure time and storage consumption of the *Topcat* library. We report the results obtained in Table 2, columns *Time* and *Memory*. These represent the timings (in seconds) and the memory (in Gigabytes) required for computing the persistence module. When the *Topcat* library runs out of memory, no result is reported. Where a comparison is possible, computing the persistence module on the Morse complex takes approximately half of the time than computing it on the original simplicial complex.

The memory consumption is the main bottleneck of the *Topcat* library as it is mainly affected by the number of input cells and by the number of multigrades. Using the Morse complex helps to deal with this problem by shrinking both numbers. In our

experiments, all successful executions have used a limited amount of memory, significantly below the machine limit of 64GB. This suggests a dramatic increase in memory usage in the ones where the computations have failed. For instance, the failure of the test over the Sphere dataset with 2882 cells and 482x482 multifiltration multigrades suggests that computing the persistence module on a Morse complex of the same size would fail as well. We should stress the fact that the objective of our experiment is that of evaluating the gain in performances when using our reduction approach and not that of overcoming the limitations of *Topcat*. Column *Morse Time* reports the partial timings required for computing the Morse complex. These include the contribution of running `ComputeDiscreteGradient` together with the retrieval of the boundary matrix. We point out that the reduction timings, ranging from 0.0244 to 0.0815 seconds, are negligible with respect to the time required for computing the persistence module.

6.2. Computing the persistence space

In this subsection, we evaluate the impact of the reduction method on the computation of the *persistence space* [42].

The foliation method. The persistence space of an n-parameter filtration is a subset of $\mathbb{R}^n \times \mathbb{R}^n$ that can be computed via the *foliation method* introduced in [43]. This amounts to considering all possible lines ℓ in $\mathbb{R}^n$ through two grades $u \prec v$. Restricting the n-parameter filtration to the grades belonging to ℓ, we obtain a one-parameter filtration, called a *slice*. On each slice, any computational technique for one-parameter persistent homology computation can be applied to obtain the corresponding persistence diagram [56]. A *persistence diagram* is a representation, on the Cartesian plane, of a one-parameter persistence module. Each point in the persistence diagram represents a discontinuity in the rank invariant created by either a new-born or a vanishing homology class. The union of the persistence diagrams for all possible lines ℓ mapped back to $\mathbb{R}^n \times \mathbb{R}^n$ gives the persistence space. In practice, it is possible to compute only a sampling of the persistence space by selecting a finite number of lines. The number of lines to consider varies based on the application at hand. As a rule of thumb, the more slices we consider, the more accurate is the approximation of the resulting persistence space. Next, we describe our implementation of the foliation method.

We applied the foliation method on bifiltrations induced by some function $f : S \to \mathbb{R}^2$. The first step consists in selecting ω^2 lines ℓ with a non-negative slope in $\mathbb{R}^2$. To do so, we use the following procedure. Because each line ℓ with positive slope in $\mathbb{R}^2$ is determined by the angle λ of the line with the x axis, and a base point b on ℓ, we have selected ω values for both λ and b. Values of λ are uniformly taken from interval $[0, \frac{\pi}{2}]$. Points b are uniformly taken from the segment whose endpoints are the projections of points (c_1, C_2) and (C_1, c_2), with $C_i := \max_{x \in S} f_i(x)$ and $c_i := \min_{x \in S} f_i(x)$ for $i = 1, 2$, onto the bisector of the second and fourth quadrant along the direction $m = (\cos(\lambda), \sin(\lambda))$. For each choice of λ and b, we thus determine the line ℓ passing through b with direction $m = (\cos(\lambda), \sin(\lambda))$. By varying the values of λ and b, we obtained the desired ω^2 possible lines. The second step requires creating a new one-parameter filtration on S for each line ℓ. If ℓ has parameters λ, b, for each simplex $\sigma \in S$, the grade of σ in the new filtration is given by $\Phi^\ell(\sigma) := \min_{i=1,2} m_i \cdot \max_{i=1,2} \frac{f_i(\sigma) - b_i}{m_i}$. The last step consists of computing classic persistent homology for the obtained one-parameter filtration by sublevel sets of

Φ^ℓ corresponding to each choice of λ and b.

After choosing how to sample slices, the foliation method still requires choosing the number of slices. In what follows, we will present results providing insights on both choices, either by varying the number of slices (between 2 and 100), or by varying the method for computing persistent homology (taken from the PHAT library [57]).

Here we present results for evaluating the impact of our reduction approach when computing the persistence space by using a variable number of slices (between 2 and 100). Datasets considered are from the Princeton Shape Benchmark [58]. Table 3 describes the datasets and the results obtained when computing the persistence space by using 100 slices and by using the standard algorithm implemented in PHAT [57]. For each dataset reported in Table 3, the first row reports data regarding the original mesh, while the second row describes the corresponding Morse complex computed by using our reduction method. For each input complex, we show the number of simplices (column *Simplices*) and the average number of persistence pairs found per slice (column *Pairs*).

Timings are reported separately for the computation of the Morse complex (column *Morse time*), for the extraction of slices (column *Line Extraction*) and for the actual computation of the persistence space (column *Foliation Time*). The latter is formerly subdivided into three partial timings accounting for the construction of the boundary matrix (column *Building Pers. input*), for the computation of persistent homology (column *Computing Persistence*), and for reindexing the persistence pairs according to the multifiltration (column *Reindexing Pers. output*). Column *Foliation Total* shows the sum of the partial timings.

						Foliations Time			
Dataset	**Simplices**	**Pairs**		**Morse Time**	**Line Extraction**	**Building Pers. input**	**Computing Persistence**	**Reindexing Pers. output**	**Foliations Total**
Shark	9491	4744	(81.4)	0.11	0.86	9.04	1.91	1.45	12.42
	1111	554				1.15	0.21	0.84	2.22
Turtle	10861	5426	(8.8)	0.12	0.63	10.21	2.11	1.53	13.87
	1197	594				1.22	0.22	0.84	2.29
Gun	27826	13873	(10.2)	0.28	0.65	27.49	5.65	2.69	35.85
	3144	1532				3.18	0.60	0.99	4.77
Piano	119081	59349	(79.5)	1.14	0.85	118.14	26.56	10.33	155.91
	10955	5286				11.09	2.26	1.65	15.01

Table 3: Timings (in seconds) required for computing the persistence pairs on 100 uniformly sampled slices. Datasets are reported by rows. For each triangle mesh, the first row is for the original dataset and the second one for the Morse complex considered over the same 100 slices. Column *Simplices* reports the number of simplices in the multifiltration. Column *Pairs* reports the average number of persistence pairs found per slice. In parantheses, the number of pairs with positive persistence (equal for original and reduced datasets).

We notice that, by reducing the number of simplices of approximately one order, we get a one-order reduction on all timings. Looking at column *Line Extraction* we notice that the extraction of the lines are almost the same across all the triangle meshes. This happens because in this case, we are always considering the same number of slices. For the partial timings, the highest contribution is shown in column *Building Pers. Input*. This is the part where the cells are sorted by increasing values under Φ^ℓ and reindexed according to these values. Both this phase and the following one (i.e., the actual computation of persistent homology) are affected by the number of input simplices, and the results for the Morse complex reflect the one-order reduction in the number of

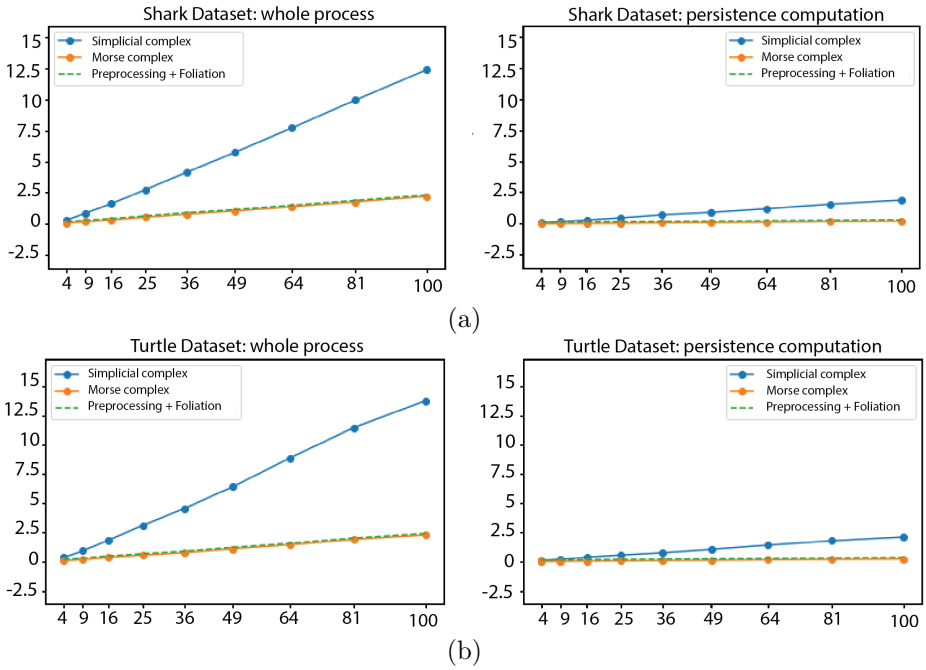


Figure 7: Time performances plotted with respect to a number of slices varying form 4 to 100 over the same dataset. Datasets considered are triangle meshes: (a) Shark and (b) Turtle. In all the figures performances are indicated in blue for the original dataset and in orange for the corresponding reduced dataset. On the left we show timings required by the foliation method. On the right, we show timings for computing persistence homology over a single slice.

simplices. Results are shown in column *Reducing Pers. Output*. The difference in the results obtained with the original triangle mesh and the corresponding Morse complex suggests that our reduction step let us avoid computation on many spurious persistence pairs. Column *Foliation Total* indicates timings for computing the persistence space as a whole. The total timings required by a Morse complex range from a minimum of 2.22 seconds (Shark triangle mesh) to a maximum of 15.01 seconds (Piano triangle mesh), whereas, the original datasets require from 12.42 (Shark triangle mesh) to 155.91 seconds (Piano triangle mesh).

In Figure 7 and Figure 8, we compare the time performances achieved by the foliation method using a number of slices ranging from 4 to 100. The one-parameter persistence over each slice is computed by the standard algorithm implemented in PHAT. For each dataset, we show, on the left, the global timings for the foliation phase and, on the right, the partial timings required by the computation of persistent homology.

Blue lines indicate results obtained for the triangle meshes, green dotted lines present results obtained with the Morse complexes accounting for both the reduction algorithm and the foliation step. Orange lines indicate results obtained with the Morse complexes exclusively for the foliation phase. As we can see, orange and green lines almost overlap indicating that the preprocessing step used for computing the Morse complex is almost negligible with respect to the computation of the persistence space.

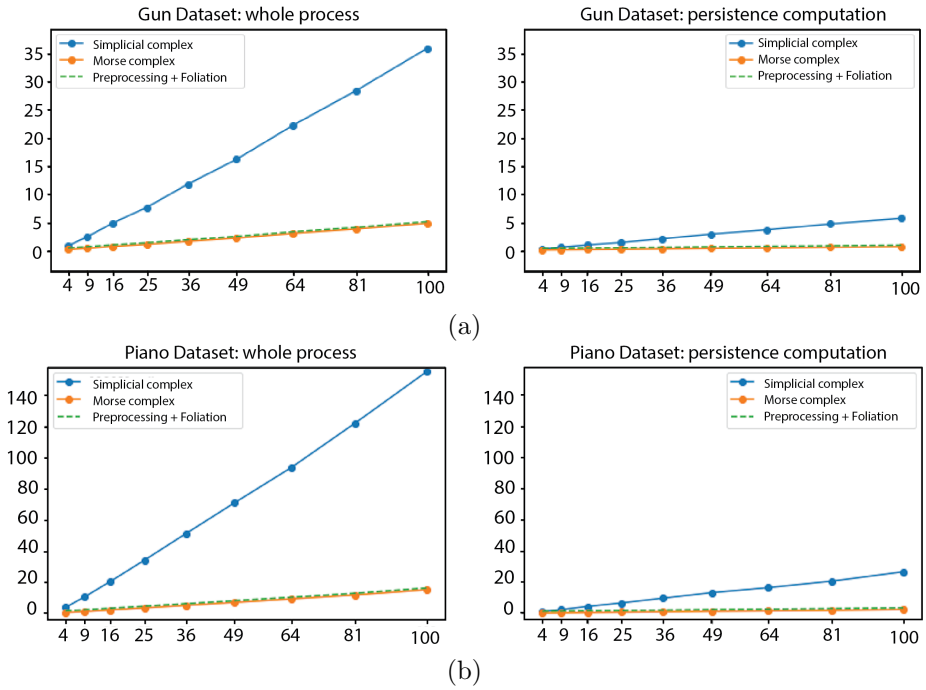


Figure 8: Time performances plotted with respect to a number of slices varying form 4 to 100 over the same dataset. Datasets considered are triangle meshes: (a) Gun and (b) Piano. In all the figures performances are indicated in blue for the original dataset and in orange for the corresponding reduced dataset. On the left we show timings required by the foliation method. On the right, we show timings for computing persistence homology over a single slice.

We also notice the linear dependency of the process on the number of slices. For Morse complexes (orange line), the slope coefficient is smaller than for the original simplicial complex (blue line). This is more evident for global timings suggesting that a preprocessing reduction is preferable independently of the number of slices considered. Notice that, only for the computation of persistent homology when using 4 slices, we get the blue line just below the green dashed line. This is the only exception where the preprocessing step could be avoided.

Our tests confirm that the complexity of the foliation method primarily depends on the number of slices considered. Our reduction approach impacts the performances by simply reducing the number of cells to be processed. Moreover, our tests show that the proposed preprocessing is effective also for a small number of slices.

7. Concluding remarks

We have proposed a new preprocessing algorithm for multiparameter persistent homology suitable for applications to real-sized data sets. We have highlighted the local characteristics of our approach as opposed to the global characteristics of the equivalent existing approach in [11]. With real data, the use of the Morse complex allowed us to successfully compute multiparameter persistent homology. We increased about 50 times

the size of the input complex that can be treated, and up to about 250 times the size of the filtration that can be treated. Our local preprocessing has shown its advantages, especially when computing the persistence space through the foliation method. We found that, in all considered datasets, the Morse complex outperforms the corresponding original filtration, regardless of the number of slices used in the foliation method. By using the Morse complex for computing the persistence module we still have experienced efficiency problems, even with rather small datasets. This suggests that our preprocessing is not powerful enough to make the persistence module computation feasible and that optimizations of current algorithms require further improvements.

At the time of submitting the present paper, a new reduction algorithm has been proposed in preprint version [59] inspired by the chunk algorithm [19]. The latter works on a matrix-based representation of the simplicial complex S called boundary matrix. Each column of the *boundary matrix* represents a simplex σ in S by storing the facets of σ. The method in [59] and the Morse-based approach discussed here have been experimentally compared showing that the one in [59] is generally an order of magnitude faster than ours. On the other hand, the Morse-based approach have other valuable characteristics.

The Morse-based approach stores additional information with respect to [59] still using a comparable amount of memory at runtime. Depending on the applicative domain this information can be fundamental. Above all, with the Morse-based approach, we are able to maintain a complete mapping between the original simplicial complex and the obtained Morse complex. That is, once we have computed multiparameter persistent homology on the Morse complex, we are able to project back the results on S for studying, for example, the distribution of the critical simplices on the original complex. This is currently impossible with the reduction approach described in [59] since, once the matrix is reduced, the connection with the original simplicial complex is lost. Maintaining such information would require storing a copy of the unreduced boundary matrix, then occupying twice as much memory.

Also, the approach in [59] is based on a global representation of the facets of each simplex of the simplicial complex. The Morse-based approach applies a local approach by encoding the top simplices of S only. While this approach has its drawbacks in lower dimensions, as shown by the timings in [59], it has been proven that it provides better scalability when the dimension of the simplicial complex increases. A new dimension-independent encoding for a simplicial complex endowed with a discrete gradient V has been proposed in [35]. The scalability of such data structure has been demonstrated by comparing to state-of-the-art approaches which are all based on the encoding of all the simplices of S. Such data structure is equivalent, in 2D and 3D, to the one used in this paper.

We think that the idea of a discrete gradient compatible with a multifiltration deserves further investigations. Currently, we are expanding the set of visual features that can be extracted from a discrete gradient for data analysis and visualization by studying the relationships between the critical simplices identified by our method and the *multigraded Betti numbers* computed by RIVET [47]. This may help us constructing a bridge between the discrete notions of critical simplices and the piecewise-linear notions of Pareto sets [60] and Jacobi sets [61].

8. Acknowledgements

This work has been partially supported by the Behavioral and Social Sciences College of the University of Maryland under the 2017-2018 Deans Research Initiative Program.The authors wish to thank Michael Kerber for interesting discussion on the results.

References

[1] P. Bendich, J. S. Marron, E. Miller, A. Pieloch, S. Skwerer, Persistent homology analysis of brain artery trees, The Annals of Applied Statistics 10 (1) (2016) 198–218. doi:10.1214/15-AOAS886.

[2] C. Giusti, R. Ghrist, D. S. Bassett, Two's company, three (or more) is a simplex: Algebraic-topological tools for understanding higher-order structure in neural data, Journal of Computational Neuroscience 41 (1) (2016) 1–14. doi:10.1007/s10827-016-0608-6.

[3] G. Carlsson, T. Ishkhanov, V. de Silva, A. Zomorodian, On the Local Behavior of Spaces of Natural Images, International Journal of Computer Vision 76 (1) (2008) 1–12. doi:10.1007/s11263-007-0056-x.

[4] H. Adams, G. Carlsson, On the Nonlinear Statistics of Range Image Patches, SIAM Journal on Imaging Sciences 2 (1) (2009) 110–117. doi:10.1137/070711669.

[5] P. Wu, C. Chen, Y. Wang, S. Zhang, C. Yuan, Z. Qian, D. Metaxas, L. Axel, Optimal Topological Cycles and Their Application in Cardiac Trabeculae Restoration, in: M. Niethammer, M. Styner, S. Aylward, H. Zhu, I. Oguz, P.-T. Yap, D. Shen (Eds.), Information Processing in Medical Imaging, Vol. 10265, Springer International Publishing, Cham, 2017, pp. 80–92. doi:10.1007/978-3-319-59050-9_7.

[6] R. van de Weygaert, G. Vegter, H. Edelsbrunner, B. Jones, P. Pranav, C. Park, W. A. Hellwing, B. Eldering, N. Kruithof, P. Bos, J. Hidding, J. Feldbrugge, E. ten Have, M. van Engelen, M. Caroli, M. Teillaud, Alpha, Betti and the Megaparsec Universe: On the Topology of the Cosmic Web, in: M. L. Gavrilova, C. K. Tan, M. A. Mostafavi (Eds.), Transactions on Computational Science XIV, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 60–101.

[7] A. Hatcher, Algebraic topology, Cambridge UP, Cambridge.
URL http://www.math.cornell.edu/~hatcher/AT/ATpage.html

[8] H. Edelsbrunner, D. Letscher, A. Zomorodian, Topological persistence and simplification, Discrete and Computational Geometry 28 (4) (2002) 511–533. doi:10.1007/s00454-002-2885-2.

[9] R. Forman, Morse Theory for Cell Complexes, Advances in Mathematics 134 (1998) 90–145. doi:10.1006/aima.1997.1650.

[10] M. Allili, T. Kaczynski, C. Landi, Reducing complexes in multidimensional persistent homology theory, Journal of Symbolic Computation 78 (C) (2017) 61–75. doi:10.1016/j.jsc.2015.11.020.

[11] M. Allili, T. Kaczynski, C. Landi, F. Masoni, Acyclic partial matchings for multidimensional persistence: Algorithm and combinatorial interpretation, Journal of Mathematical Imaging and Visiondoi:10.1007/s10851-018-0843-8.
URL https://doi.org/10.1007/s10851-018-0843-8

[12] F. Iuricich, S. Scaramuccia, C. Landi, L. De Floriani, A discrete Morse-based approach to multivariate data analysis, SIGGRAPH ASIA 2016 Symposium on Visualization on - SA '16 Dec (2016) 1–8. doi:10.1145/3002151.3002166.

[13] G. Carlsson, A. Zomorodian, The theory of multidimensional persistence, in: SoCG '07 Proceedings of the twenty-third annual symposium on Computational geometry, Vol. 392, ACM New York, Gyeongju, South-Korea, 2007, pp. 184–193. doi:10.1145/1247069.1247105.

[14] K. Mischaikow, V. Nanda, Morse Theory for Filtrations and Efficient Computation of Persistent Homology, Discrete & Computational Geometry 50 (2) (2013) 330–353. doi:10.1007/s00454-013-9529-6.

[15] C. Chen, M. Kerber, Persistent homology computation with a twist, in: 27th European Workshop on Computational Geometry, Vol. 45, 2011, pp. 28–31. doi:10.1.1.224.6560.

[16] V. de Silva, D. Morozov, M. Vejdemo-Johansson, Dualities in persistent (co)homology 27 (12) (2011) 124003. doi:10.1088/0266-5611/27/12/124003.

[17] N. Milosavljević, D. Morozov, P. Skraba, Zigzag Persistent Homology in Matrix Multiplication Time, in: Proc. 27th Ann. Symp. Comput. Geom., SoCG '11, ACM, New York, NY, USA, 2011, pp. 216–225. doi:10.1145/1998196.1998229.

[18] H. Edelsbrunner, J. Harer, Persistent homology-a survey, Contemporary mathematics 453 (2008) 257–282. doi:10.1090/conm/453/08802.

[19] U. Bauer, M. Kerber, J. Reininghaus, Clear and Compress: Computing Persistent Homology in Chunks, in: arXiv preprint arXiv:1303.0477, Springer International Publishing, 2013, pp. 1–12. doi:10.1007/978-3-319-04099-8_7.

[20] O. Busaryev, S. Cabello, C. Chen, T. K. Dey, Y. Wang, Annotating simplices with a homology basis and its applications, in: F. V. Fomin, P. Kaski (Eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7357 of Lecture Notes in Computer Science, Springer, 2012, pp. 189–200. doi:10.1007/978-3-642-31155-0_17.

[21] T. K. Dey, F. Fan, Y. Wang, Computing Topological Persistence for Simplicial Maps, in: Annual Symposium on Computational Geometry - SOCG'14, ACM, 2014, pp. 345–354. doi:10.1145/2582112.2582165.

[22] J. D. Boissonnat, T. K. Dey, C. Maria, The Compressed Annotation Matrix: en Efficient Data Structure for Computing Persistent Cohomology, in: Algorithms - ESA 2013, Vol. 8125 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 695–706. doi:10.1007/978-3-642-40450-4_59.

[23] P. Dłotko, H. Wagner, Simplification of complexes for persistent homology computations, Homology, Homotopy and Applications 16 (1) (2014) 49–63. doi:10.4310/HHA.2014.v16.n1.a3.

[24] M. Mrozek, B. Batko, Coreduction Homology Algorithm, Discrete & Computational Geometry 41 (1) (2009) 96–118. doi:10.1007/s00454-008-9073-y.

[25] M. Mrozek, T. Wanner, Coreduction homology algorithm for inclusions and persistent homology, Computers & Mathematics with Applications 60 (10) (2010) 2812–2833. doi:10.1016/j.camwa.2010.09.036.

[26] P. Dłotko, T. Kaczynski, M. Mrozek, T. Wanner, Coreduction Homology Algorithm for Regular CW-Complexes, Discrete and Computational Geometry 46 (2) (2011) 361–388. doi:10.1007/s00454-010-9303-y.

[27] M. Mrozek, P. Pilarczyk, N. Zelazna, Homology algorithm based on acyclic subspace, Computers and Mathematics with Applications 55 (11) (2008) 2395–2412. doi:10.1016/j.camwa.2007.08.044.

[28] H. King, K. Knudson, N. Mramor, Generating Discrete Morse Functions from Point Data, Experimental Mathematics 14 (4) (2005) 435–444. doi:10.1080/10586458.2005.10128941.

[29] N. Shivashankar, S. Maadasamy, V. Natarajan, Parallel computation of 2D Morse-Smale complexes, IEEE Trans. Vis. Comput. Graph. 18 (10) (2012) 1757–1770. doi:10.1109/TVCG.2011.284.

[30] N. Shivashankar, V. Natarajan, Parallel computation of 3D Morse-Smale complexes, Comput. Graph. Forum 31 (3) (2012) 965–974. doi:10.1111/j.1467-8659.2012.03089.x.

[31] V. Robins, P. J. Wood, A. P. Sheppard, Theory and algorithms for constructing discrete Morse complexes from grayscale digital images, IEEE Transactions on Pattern Analysis and Machine Intelligence 33 (8) (2011) 1646–1658. doi:10.1109/TPAMI.2011.95.

[32] D. Günther, J. Reininghaus, H. Wagner, I. Hotz, Efficient computation of 3D Morse–Smale complexes and persistent homology using discrete Morse theory, The Visual Computer 28 (10) (2012) 959–969. doi:10.1007/s00371-012-0726-8.

[33] R. Fellegara, F. luricich, L. De Floriani, K. Weiss, Efficient computation and simplification of discrete Morse decompositions on triangulated terrains, in: Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '14, ACM, 2014, pp. 223–232. doi:10.1145/2666310.2666412.

[34] K. Weiss, F. Iuricich, R. Fellegara, L. De Floriani, A primal/dual representation for discrete Morse complexes on tetrahedral meshes, Computer Graphics Forum 32 (3pt3) (2013) 361–370. doi:10.1111/cgf.12123.

[35] U. Fugacci, F. Iuricich, L. De Floriani, Computing Discrete Morse Complexes from Simplicial Complexes, Graphical Models 103 (2019) 101023. doi:https://doi.org/10.1016/j.gmod.2019.101023.
URL http://www.sciencedirect.com/science/article/pii/S1524070319300141

[36] G. Carlsson, G. Singh, A. Zomorodian, Computing multidimensional persistence, in: Y. Dong, D.-Z. Du, O. Ibarra (Eds.), Lecture Notes in Computer Science, Vol. 5878 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 730–739. doi:10.1007/978-3-642-10631-6_74.

[37] O. Gäfvert, Algorithms for Multidimensional Persistence, Master thesis, KTH Royal Institute of Technology.

[38] J. R. Munkres, Elements of Algebraic Topology, Perseus Books, 1984.

[39] M. K. Agoston, Computer Graphics and Geometric Modeling: Mathematics, Springer Verlag Lon-

don Ltd., 2005.
[40] Gäfvert, Oliver , TopCat: a Java library for computing invariants on multidimensional persistence modules (2016) [cited 2017-03-30].
URL https://github.com/olivergafvert/topcat
[41] A. Cerri, C. Landi, The Persistence Space in Multidimensional Persistent Homology, in: R. Gonzalez-Diaz, M.-J. Jimenez, B. Medrano (Eds.), Discrete Geometry for Computer Imagery, Vol. 7749 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 180–191. doi:10.1007/978-3-642-37067-0_16.
[42] A. Cerri, C. Landi, Hausdorff stability of persistence spaces, Foundations of Computational Mathematics 16 (2) (2016) 343–367. doi:10.1007/s10208-015-9244-1.
URL https://doi.org/10.1007/s10208-015-9244-1
[43] S. Biasotti, A. Cerri, P. Frosini, D. Giorgi, C. Landi, Multidimensional size functions for shape comparison, Journal of Mathematical Imaging and Vision 32 (2) (2008) 161–179. doi:10.1007/s10851-008-0096-z.
[44] S. Biasotti, A. Cerri, P. Frosini, D. Giorgi, A new algorithm for computing the 2-dimensional matching distance between size functions, Pattern Recognition Letters 32 (14) (2011) 1735–1746. doi:10.1016/j.patrec.2011.07.014.
[45] F. Cagliari, B. Di Fabio, M. Ferri, One-dimensional reduction of multidimensional persistent homology, Proceedings of the American Mathematical Society 138 (08) (2010) 3003–3003. doi:10.1090/S0002-9939-10-10312-8.
[46] S. Biasotti, A. Cerri, D. Giorgi, M. Spagnuolo, PHOG: Photometric and geometric functions for textured shape retrieval, Computer Graphics Forum 32 (5) (2013) 13–22. doi:10.1111/cgf.12168.
[47] M. Lesnick, M. Wright, Interactive Visualization of 2-D Persistence Modules, ArXiv preprint (2015) 1–75arXiv:1512.00180.
[48] K. P. Knudson, A refinement of multi-dimensional persistence, Homology, Homotopy and Applications 10 (1) (2008) 259–281. doi:10.4310/HHA.2008.v10.n1.a11.
[49] D. Eisenbud, The Geometry of Syzygies: A second course in Commutative Algebra and Algebraic Geometry, Springer, New York, NY, 2005. doi:10.1007/b137572.
[50] A. Cerri, P. Frosini, C. Landi, A global reduction method for multidimensional size graphs, Electronic Notes in Discrete Mathematics 26 (2006) 21–28. doi:10.1016/j.endm.2006.08.004.
[51] H. Edelsbrunner, E. P. Mücke, Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms, ACM Transactions on Graphics 9 (1) (1990) 66–104. doi:10.1145/77635.77639.
[52] D. Canino, L. D. Floriani, K. Weiss, IA*: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions, Computers & Graphics 35 (3) (2011) 747 – 753, shape Modeling International (SMI) Conference 2011. doi:https://doi.org/10.1016/j.cag.2011.03.009.
[53] M. Allili, T. Kaczynski, C. Landi, F. Masoni, Algorithmic Construction of Acyclic Partial Matchings for Multidimensional Persistence, in: Kropatsch W., Artner N., Janusch I. (Eds.), Discrete Geometry for Computer Imagery. DGCI 2017. Lecture Notes in Computer Science, vol 10502, Springer, Cham, 2017, pp. 375–387. doi:10.1007/978-3-319-66272-5_30.
[54] Federico Iuricich, MDG: a C++ library for computing discrete gradients on multivariate data (2018) [cited 2018-09-30].
URL https://github.com/IuricichF/fg_multi
[55] E. Catmull, J. Clark, Recursively generated b-spline surfaces on arbitrary topological meshes, Computer-Aided Design 10 (6) (1978) 350 – 355. doi:https://doi.org/10.1016/0010-4485(78)90110-0.
[56] H. Edelsbrunner, J. Harer, Persistent homology - a survey, in: Contemporary Mathematics, Vol. 453, Providence, RI: American Mathematical Society, 2008, pp. 257–282. doi:10.1090/conm/453/08802.
[57] U. Bauer, M. Kerber, J. Reininghaus, Persistent Homology Algorithm Toolbox (PHAT) (2013).
[58] P. Shilane, P. Min, M. Kazhdan, T. Funkhouser, The Princeton Shape Benchmark, in: Shape Modeling Applications, 2004. Proceedings, Genova, Italy, 2004, pp. 167–178. doi:10.1109/SMI.2004.1314504.
[59] U. Fugacci, M. Kerber, Chunk Reduction for Multi-Parameter Persistent Homology, in: In 39th International Symposium on Computational Geometry, 2019, 2019.
[60] L. Huettenberger, C. Heine, C. Garth, Decomposition and simplification of multivariate data using Pareto sets, IEEE Transactions on Visualization and Computer Graphics 20 (12) (2014) 2684–2693. doi:10.1109/TVCG.2014.2346447.

[61] H. Edelsbrunner, J. L. Harer, Jacobi sets, in: Foundations of Computational Mathematics: Minneapolis, 2002, Vol. 312 of London Mathematical Society Lecture Note Series, Cambridge University Press, 2004, pp. 37–57. doi:10.1017/CBO9781139106962.003.

Appendix

In this appendix, we report all the proofs of the results provided in Subsection 5.2.2.

Proof of Lemma 1. Let $\sigma \in S$. Because the sets $\mathrm{Low}_I(v)$, with $v \in S_0$, form a partition of S, there is a unique vertex v of S such that σ belongs to $\mathrm{Low}_I(v)$. By definition of lower star, v is a face of σ, and $I(\sigma) \leqslant I(v)$. Because I is extended from the vertices to other simplexes by taking the maximum over all vertexes, it also holds that $I(v) \leqslant I(\sigma)$, and hence $I(v) = I(\sigma)$.

Let τ be a simplex in $\mathrm{Low}_f(\sigma)$. Again by definition of lower star and because f is defined by max-extension as well, we similarly get that σ is a face of τ and $f(\tau) = f(\sigma)$. Since I is well-extensible, this implies that $I(\tau) = I(\sigma)$, and hence $I(\tau) = I(v)$. Because $v < \sigma < \tau$ we can thus conclude that $\tau \in \mathrm{Low}_I(v)$. Therefore, $\mathrm{Low}_f(\sigma) \subseteq \mathrm{Low}_I(v)$. □

Proof of Lemma 2. In order to prove uniqueness, assume by contradiction that there are two simplexes $\sigma, \sigma' \in S$ such that $\mathrm{Low}_f(\sigma) = L_u = \mathrm{Low}_f(\sigma')$. Hence, $\sigma' \in \mathrm{Low}_f(\sigma)$ and $\sigma \in \mathrm{Low}_f(\sigma')$, implying that σ' is a coface of σ and σ is a coface of σ'. Thus, $\sigma' = \sigma$.

In order to prove existence, we take σ to be the maximal common face of all simplexes of L_u. Since $v \in \tau$ for every $\tau \in L_u$, σ is non-empty and belongs to $\mathrm{Low}_I(v)$. In particular, σ belongs to L_u. Indeed, on one hand, for every τ in L_u, $f(\sigma) \preceq f(\tau) = u$, because f does not decrease with the coface relation. On the other hand, since f is obtained by max-extension from the vertices of a component-wise injective function, for every $i = 1, \ldots, n$ there is a unique vertex w_i such that $u_i = f_i(w_i)$. Hence, for all $\tau \in L_u$, it holds that $u_i = f_i(\tau) \geqslant f_i(w_i) = u_i$, implying that w_i is a vertex of τ for all $\tau \in L_u$. By definition of σ, w_i is also a vertex of σ. Therefore, $f_i(\sigma) \geqslant f_i(w_i) = u_i$. Because this holds for every $i = 1, \ldots, n$, we deduce that $u \preceq f(\sigma)$, which together with $f(\sigma) \preceq u$ yields $f(\sigma) = u$. In conclusion, σ belongs to L_u.

We now claim that $\mathrm{Low}_f(\sigma) = L_u$. We easily see that L_u is contained in $\mathrm{Low}_f(\sigma)$ because, by definition of σ, all simplices $\tau \in L_u$ are cofaces of σ, and by definition of level set, $f(\sigma) = f(\tau) = u$. Conversely, to prove that $\mathrm{Low}_f(\sigma) \subseteq L_u$, let τ be a simplex in $\mathrm{Low}_f(\sigma)$. Lemma 1 ensures that $\tau \in \mathrm{Low}_I(v)$, so it is sufficient to prove that $f(\tau) = u$. We have already proved that $f(\sigma) = u$, hence the claim follows by using again the fact that, for all $\tau \in \mathrm{Low}_f(\sigma)$, $f(\tau) = f(\sigma)$.

To conclude the proof, it remains to show that σ is a primary simplex for `Matching`. Recall that a simplex is primary if and only if it is not contained in a lower star, with respect to f, of some other simplex τ: $\sigma \in P$ if and only if there is no other simplex τ such that $\sigma \in \mathrm{Low}_f(\tau)$. By contradiction, let us assume such simplex τ exists. Hence $\mathrm{Low}_f(\sigma) \subseteq \mathrm{Low}_f(\tau)$. By Lemma 1, σ and τ belong to the same index-based lower star $\mathrm{Low}_I(v)$. Since $\sigma \in \mathrm{Low}_f(\tau)$, we get $u = f(\sigma) = f(\tau)$. Thus, $L_u = \mathrm{Low}_f(\sigma) \subseteq \mathrm{Low}_f(\tau) \subseteq L_u$, implying that $\mathrm{Low}_f(\sigma) = \mathrm{Low}_f(\tau)$, and hence $\sigma = \tau$, a contradiction. □

Proof of Proposition 3. To prove the claim, we show how to construct an indexing J on S valid as an input of `Matching` such that each call of `HomotopyExpansion` with simplices of $\mathrm{Low}_f(\sigma) = L_u$ taken in the order of J gives the same result as `ComputeDiscreteGradient`.

For each $\sigma \in P$, we indicate by J_σ the indexing on simplexes of $\mathrm{Low}_f(\sigma) = L_u$ built in $\texttt{HomotopyExpansion}(S, I, L_u)$ when called by `ComputeDiscreteGradient`. By construction, it is increasing with the coface relation, and consistent with the orderings of the lists `Ord0` and `Ord1`.

Let $g : P \longrightarrow \mathbb{R}$ be any injective function compatible with f, i.e., $f(\sigma) \preceq f(\tau)$ implies $g(\sigma) \leqslant g(\tau)$ (obtained, for example, by topological sorting). For every simplex $\tau \in S$, we can consider the map $Q : S \to P$ such that $Q(\tau) = \sigma$, with σ the unique primary simplex $\sigma \in P$ such that $\tau \in \mathrm{Low}_f(\sigma)$. Thus, we can extend g from P to S by taking $G = g \circ Q$. By construction, G is still compatible with f.

Therefore, for any simplex $\tau \in S$, we get a pair of real numbers $(G(\tau), J_{Q(\tau)}(\tau))$. The set of all such pairs can be lexicographically ordered, and we can finally take $J : S \longrightarrow \mathbb{R}$ to be an injective map giving a total order equivalent to such lexicographic order.

We need to show that J is a valid ordering for `Matching`, that is it satisfies condition (2). If $\sigma < \tau$, we have $f(\sigma) \preceq f(\tau)$ because f is defined by max-extension from the vertices. By compatibility of G with f, this implies that $G(\sigma) \leqslant G(\tau)$. In the case $G(\sigma) < G(\tau)$, by the equivalence of J with the lexicographic order on all the pairs $(G(\tau), J_{Q(\tau)}(\tau))$ with $\tau \in S$, we get $J(\sigma) < J(\tau)$. In the case $G(\sigma) = G(\tau)$, by injectivity of g, it follows that $Q(\sigma) = Q(\tau)$. Because $J_{Q(\sigma)}$ is increasing with the coface relation, we get $J(\sigma) < J(\tau)$. Hence, in any case, $\sigma < \tau$ implies $J(\sigma) < J(\tau)$. Let us now assume that $f(\sigma) \precnsim f(\tau)$. From $f(\sigma) = f \circ Q(\sigma)$ and $f(\tau) = f \circ Q(\tau)$, it follows that $f \circ Q(\sigma) \precnsim f \circ Q(\tau)$. Necessarily, $Q(\sigma) \neq Q(\tau)$. Thus, by injectivity of g, $g \circ Q(\sigma) < g \circ Q(\tau)$. Because $G = g \circ Q$, we conclude that $J(\sigma) < J(\tau)$ by equivalence of J with the lexicographic order on the pairs considered above. Therefore, we have proved that J satisfies condition (2) and is a valid ordering for `Matching`. In conclusion, by Lemma 2, `Matching` and `ComputeDiscreteGradient` call `HomotopyExpansion` with the same input and, provided that `Matching` uses the ordering J, they also process simplices in the same order. Hence, `Matching` and `ComputeDiscreteGradient` produce the same output. □

Proof of Corollary 4. By Proposition 3, for any valid input (S, f) for `ComputeDiscreteGradient` there is a valid input (S, f, J) for `Matching` such that $\texttt{ComputeDiscreteGradient}(S, f)$ is equal to $\texttt{Matching}(S, f, J)$. By Proposition 8 and Theorem 9 in [11], $\texttt{Matching}(S, f, J)$ returns a discrete gradient V that is compatible with the multifiltration induced by the pair (S, f). Therefore, so does $\texttt{ComputeDiscreteGradient}(S, f)$. □