Fast Computation of Zigzag Persistence

Tamal K. Dey ✉
Department of Computer Science, Purdue University, West Lafayette, IN, USA

Tao Hou ✉
School of Computing, DePaul University, Chicago, IL, USA

Abstract

Zigzag persistence is a powerful extension of the standard persistence which allows deletions of simplices besides insertions. However, computing zigzag persistence usually takes considerably more time than the standard persistence. We propose an algorithm called FASTZIGZAG which narrows this efficiency gap. Our main result is that an input simplex-wise zigzag filtration can be converted to a *cell*-wise non-zigzag filtration of a Δ-complex with the same length, where the cells are copies of the input simplices. This conversion step in FASTZIGZAG incurs very little cost. Furthermore, the barcode of the original filtration can be easily read from the barcode of the new cell-wise filtration because the conversion embodies a series of *diamond switches* known in topological data analysis. This seemingly simple observation opens up the vast possibilities for improving the computation of zigzag persistence because any efficient algorithm/software for standard persistence can now be applied to computing zigzag persistence. Our experiment shows that this indeed achieves substantial performance gain over the existing state-of-the-art softwares.

1 Introduction

Standard persistent homology defined over a growing sequence of simplicial complexes is a fundamental tool in topological data analysis (TDA). Since the advent of persistence algorithm [18] and its algebraic understanding [30], various extensions of the basic concept have been explored [6, 8, 12, 13]. Among these extensions, zigzag persistence introduced by Carlsson and de Silva [6] is an important one. It empowered TDA to deal with filtrations where both insertion and deletion of simplices are allowed. In practice, allowing deletion of simplices does make the topological tool more powerful. For example, in dynamic networks [15, 21] a sequence of graphs may not grow monotonically but can also shrink due to disappearance of vertex connections. Furthermore, zigzag persistence seems to be naturally connected with the computations involving multiparameter persistence, see e.g. [16, 17].

Zigzag persistence possesses some key differences from standard persistence. For example, unlike standard (non-zigzag) modules which decompose into only finite and infinite intervals, zigzag modules decompose into four types of intervals (see Definition 2). Existing algorithms for computing zigzag persistence from a zigzag filtration [8, 22, 23, 24] are all based on maintaining explicitly or implicitly a consistent basis throughout the filtration. This makes these algorithms for zigzag persistence more involved and hence slower in practice than

algorithms for the non-zigzag version though they have the same time complexity [25]. We sidestep the bottleneck of maintaining an explicit basis and propose an algorithm called FASTZIGZAG, which converts the input zigzag filtration to a *non-zigzag* filtration with an efficient strategy for mapping barcodes of the two bijectively. Then, we can apply any efficient algorithm for standard persistence on the resulting non-zigzag filtration to compute the barcode of the original filtration. Considering the abundance of optimizations [2, 3, 4, 5, 10, 11] of standard persistence algorithms and a recent GPU acceleration [29], the conversion in FASTZIGZAG enables zigzag persistence computation to take advantage of any existing or future improvements on standard persistence computation. Our implementation, which uses the PHAT [4] software for computing standard persistence, shows substantial performance gain over existing state-of-the-art softwares [26, 28] for computing zigzag persistence (see Section 3.5). We make our software publicly available through: `https://github.com/taohou01/fzz`.

To elaborate on the strategy of FASTZIGZAG, we first observe a special type of zigzag filtrations called *non-repetitive* zigzag filtrations in which a simplex (or more generally, a *cell*) is never added again once deleted. Such a filtration admits an *up-down* filtration as its canonical form that can be obtained by a series of *diamond switches* [6, 7, 8]. The up-down filtration can be further converted into a *non-zigzag* filtration again using diamond switches as in the Mayer-Vietoris pyramid presented in [8]. Individual switches are atomic tools that help us to show equivalence of barcodes, but we do not need to actually execute them in computation. Instead, we go straight to the final form of the filtration quite easily and efficiently. Finally, we observe that any zigzag filtration can be treated as a non-repetitive *cell-wise* filtration of a Δ-*complex* [20] consisting of multisets of input simplices. This means that each repeatedly added simplex is treated as a different cell in the Δ-complex, so that we can apply our findings for non-repetitive filtrations to arbitrary filtrations. The conversions described above are detailed in Section 3.

1.1 Related works

Zigzag persistence is essentially an A_n-*type quiver* [14] in mathematics which is first introduced to the TDA community by Carlsson and de Silva [6]. In their paper [6], Carlsson and de Silva also study the Mayer-Vietoris diamond used in this paper and propose an algorithm for computing zigzag barcodes from zigzag modules (i.e., an input is a sequence of vector spaces connected by linear maps encoded as matrices). Carlsson et al. [8] then propose an $O(mn^2)$ algorithm for computing zigzag barcodes from zigzag filtrations using a structure called right filtration. In their paper [8], Carlsson et al. also extend the classical sublevelset filtrations for functions on topological spaces by proposing levelset zigzag filtrations and show the equivalence of levelset zigzag with the extended persistence proposed by Cohen-Steiner et al. [12]. Maria and Outdot [22, 23] propose an alternative algorithm for computing zigzag barcodes by attaching a reversed standard filtration to the end of the partial zigzag filtration being scanned. Their algorithm maintains the barcode over the Surjective and Transposition Diamond on the constructed zigzag filtration [22, 23]. Maria and Schreiber [24] propose a Morse reduction preprocessing for zigzag filtrations which speeds up the zigzag barcode computation. Carlsson et al. [9] discuss some matrix factorization techniques for computing zigzag barcodes from zigzag modules, which, combined with a divide-and-conquer strategy, lead to a parallel algorithm for computing zigzag persistence. Almost all algorithms reviewed so far have a cubic time complexity. Milosavljević et al. [25] establish an $O(m^\omega)$ theoretical complexity for computing zigzag persistence from filtrations, where $\omega < 2.37286$ is the matrix multiplication exponent [1]. Recently, Dey and Hou [15] propose near-linear algorithms for computing zigzag persistence from the special cases of graph filtrations, with the help of representatives defined for the intervals and some dynamic graph data structures.
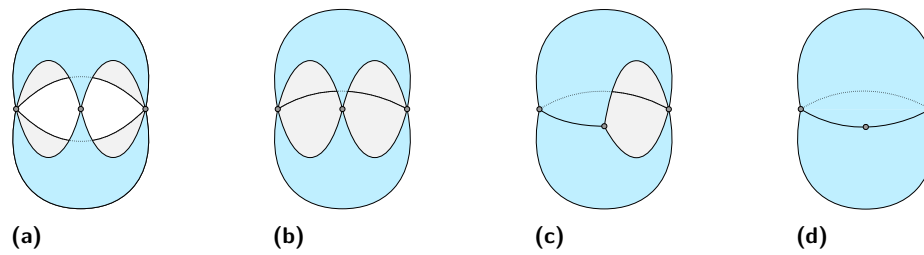
Figure 1 Examples of Δ-complexes with two triangles sharing 0, 1, 2, or 3 edges on their boundaries.

2   Preliminaries

Δ-complex.   In this paper, we build filtrations on Δ-*complexes* which are extensions of simplicial complexes described in Hatcher [20]. These Δ-complexes are derived from a set of standard simplices by identifying the boundary of each simplex with other simplices while preserving the vertex orders. For distinction, building blocks of Δ-complexes (i.e., standard simplices) are called *cells*. Motivated by a construction from the input simplicial complex described in Algorithm 3.1, we use a *more restricted* version of Δ-complexes, where boundary cells of each p-cell are identified with *distinct* $(p-1)$-cells. Notice that this makes each p-cell combinatorially equivalent to a p-simplex. Hence, the difference of the Δ-complexes in this paper from the standard simplicial complexes is that common faces of two cells in the Δ-complexes can have more relaxed forms. For example, in Figure 1, two "triangles" (2-cells) in a Δ-complex having the same set of vertices can either share 0, 1, 2, or 3 edges in their boundaries; note that the two triangles in Figure 1d form a 2-cycle.

Formally, we define Δ-complexes recursively similar to the classical definition of *CW-complexes* [20] though it need not be as general; see Hatcher's book [20] for a more general definition. Note that simplicial complexes are trivially Δ-complexes and therefore most definitions in this section target Δ-complexes.

▶ **Definition 1.** *A Δ-complex is defined recursively with dimension:*

1. *A 0-dimensional Δ-complex K^0 is a set of points, each called a 0-**cell**.*
2. *A p-dimensional Δ-complex K^p, $p \geq 1$, is a quotient space of a $(p-1)$-dimensional Δ-complex K^{p-1} along with several standard p-simplices. The quotienting is realized by an attaching map $h : \partial(\sigma) \to K^{p-1}$ which identifies the boundary $\partial(\sigma)$ of each p-simplex σ with points in K^{p-1} so that h is a homeomorphism onto its image. We term the standard p-simplex σ with boundary identified to K^{p-1} as a p-**cell** in K^p. Furthermore, we have that the restriction of h to each proper face of σ is a homeomorphism onto a cell in K^{p-1}.*

Notice that the original (more general) Δ-complexes [20] require specifying vertex orders when identifying the cells. However, the restricted Δ-complexes defined above do not require specifying such orders because we always identify the boundaries of cells by *homeomorphisms* and hence the vertex orders for identification are implicitly derived from a vertex order of a seeding cell.

Homology.   Homology in this paper is defined on Δ-complexes, which is defined similarly as for simplicial complexes [20]. All homology groups are taken with $\mathbb{Z}_2$-coefficients and therefore vector spaces mentioned in this paper are also over $\mathbb{Z}_2$.

Zigzag filtration and barcode. A *zigzag filtration* (or simply *filtration*) is a sequence of Δ-complexes

$$\mathcal{F} : K_0 \leftrightarrow K_1 \leftrightarrow \cdots \leftrightarrow K_m,$$

in which each $K_i \leftrightarrow K_{i+1}$ is either a forward inclusion $K_i \hookrightarrow K_{i+1}$ or a backward inclusion $K_i \hookleftarrow K_{i+1}$. For computational purposes, we only consider *cell-wise* filtrations in this paper, i.e., each inclusion $K_i \leftrightarrow K_{i+1}$ is an addition or deletion of a *single* cell; such an inclusion is sometimes denoted as $K_i \xleftrightarrow{\sigma} K_{i+1}$ with σ indicating the cell being added or deleted.

We call $\mathcal{F}$ as *non-repetitive* if whenever a cell σ is deleted from $\mathcal{F}$, the cell σ is never added again. We call $\mathcal{F}$ an *up-down* filtration [8] if $\mathcal{F}$ can be separated into two parts such that the first part contains only forward inclusions and the second part contains only backward ones, i.e., $\mathcal{F}$ is of the form $\mathcal{F} : K_0 \hookrightarrow K_1 \hookrightarrow \cdots \hookrightarrow K_\ell \hookleftarrow K_{\ell+1} \hookleftarrow \cdots \hookleftarrow K_m$. Usually in this paper, filtrations start and end with empty complexes, e.g., $K_0 = K_m = \varnothing$ in $\mathcal{F}$.

Applying the p-th homology functor on $\mathcal{F}$ induces a *zigzag module*:

$$\mathsf{H}_p(\mathcal{F}) : \mathsf{H}_p(K_0) \leftrightarrow \mathsf{H}_p(K_1) \leftrightarrow \cdots \leftrightarrow \mathsf{H}_p(K_m),$$

in which each $\mathsf{H}_p(K_i) \leftrightarrow \mathsf{H}_p(K_{i+1})$ is a linear map induced by inclusion. It is known [6, 19] that $\mathsf{H}_p(\mathcal{F})$ has a decomposition of the form $\mathsf{H}_p(\mathcal{F}) \simeq \bigoplus_{k \in \Lambda} \mathcal{I}^{[b_k, d_k]}$, in which each $\mathcal{I}^{[b_k, d_k]}$ is a special type of zigzag module called *interval module* over the interval $[b_k, d_k]$. The (multi-)set of intervals denoted as $\mathsf{Pers}_p(\mathcal{F}) := \{[b_k, d_k] \mid k \in \Lambda\}$ is an invariant of $\mathcal{F}$ and is called the *p-th barcode* of $\mathcal{F}$. Each interval in $\mathsf{Pers}_p(\mathcal{F})$ is called a *p-th persistence interval* and is also said to be in dimension p. Frequently in this paper, we consider the barcode of $\mathcal{F}$ in all dimensions $\mathsf{Pers}_*(\mathcal{F}) := \bigsqcup_{p \geq 0} \mathsf{Pers}_p(\mathcal{F})$.

▶ **Definition 2** (Open and closed birth/death). *For a zigzag filtration $\mathcal{F} : \varnothing = K_0 \leftrightarrow K_1 \leftrightarrow \cdots \leftrightarrow K_m = \varnothing$, the start of any interval in $\mathsf{Pers}_*(\mathcal{F})$ is called a **birth index** in $\mathcal{F}$ and the end of any interval is called a **death index**. Moreover, a birth index b is said to be **closed** if $K_{b-1} \hookrightarrow K_b$ is a forward inclusion; otherwise, b is **open**. Symmetrically, a death index d is said to be **closed** if $K_d \hookleftarrow K_{d+1}$ is a backward inclusion; otherwise, d is **open**. The types of the birth/death ends classify intervals in $\mathsf{Pers}_*(\mathcal{F})$ into four types: **closed-closed**, **closed-open**, **open-closed**, and **open-open**.*

▶ Remark 3. If $\mathcal{F}$ is a levelset zigzag filtration [8], then the open and closed ends defined above are the same as for levelset zigzag.

▶ Remark 4. An inclusion $K_i \leftrightarrow K_{i+1}$ in a cell-wise filtration either provides $i + 1$ as a birth index or provides i as a death index (but cannot provide both).

Mayer-Vietoris diamond. The algorithm in this paper draws upon the Mayer-Vietoris diamond proposed by Carlsson and de Silva [6] (see also [7, 8]), which relates barcodes of two filtrations differing by a local change:

▶ **Definition 5** (Mayer-Vietoris diamond [6]). *Two cell-wise filtrations $\mathcal{F}$ and $\mathcal{F}'$ are related by a **Mayer-Vietoris diamond** if they are of the following forms (where $\sigma \neq \tau$):*

$$
\begin{array}{c}
\mathcal{F} : \\
\\
\mathcal{F}' :
\end{array}
\qquad
\begin{array}{c}
\quad \overset{\sigma}{\nearrow} \;\; K_j \;\; \overset{\tau}{\nwarrow} \\
K_0 \longleftrightarrow \cdots \longleftrightarrow K_{j-1} \qquad\qquad K_{j+1} \longleftrightarrow \cdots \longleftrightarrow K_m \\
\quad \underset{\tau}{\nwarrow} \;\; K'_j \;\; \underset{\sigma}{\nearrow}
\end{array}
\qquad (1)
$$

*In the above diagram, $\mathcal{F}$ and $\mathcal{F}'$ differ only in the complexes at index j and $\mathcal{F}'$ is derived from $\mathcal{F}$ by switching the addition of σ and deletion of τ. We also say that $\mathcal{F}'$ is derived from $\mathcal{F}$ by an **outward** switch and $\mathcal{F}$ is derived from $\mathcal{F}'$ by an **inward** switch.*

▶ Remark 6. In Equation (1), we only provide a specific form of Mayer-Vietoris diamond which is sufficient for our purposes; see [6, 8] for a more general form. According to [6], the diamond in Equation (1) is a Mayer-Vietoris diamond because $K_j = K_{j-1} \cup K_{j+1}$ and $K'_j = K_{j-1} \cap K_{j+1}$.

We then have the following fact:

▶ **Theorem 7** (Diamond Principle [6]). *Given two cell-wise filtrations $\mathcal{F}, \mathcal{F}'$ related by a Mayer-Vietoris diamond as in Equation (1), there is a bijection from $\mathsf{Pers}_*(\mathcal{F})$ to $\mathsf{Pers}_*(\mathcal{F}')$ as follows*:

$\mathsf{Pers}_*(\mathcal{F})$		$\mathsf{Pers}_*(\mathcal{F}')$
$[b, j-1]$; $b \leq j-1$	$\mapsto$	$[b, j]$
$[b, j]$; $b \leq j-1$	$\mapsto$	$[b, j-1]$
$[j, d]$; $d \geq j+1$	$\mapsto$	$[j+1, d]$
$[j+1, d]$; $d \geq j+1$	$\mapsto$	$[j, d]$
$[j, j]$ of dimension p	$\mapsto$	$[j, j]$ of dimension $p-1$
$[b, d]$; all other cases	$\mapsto$	$[b, d]$

Note that the bijection preserves the dimension of the intervals except for $[j, j]$.

▶ Remark 8. In the above bijection, only an interval containing *some but not all* of $\{j-1, j, j+1\}$ maps to a different interval or different dimension.

3   FastZigzag algorithm

In this section, we show that computing barcodes for an arbitrary zigzag filtration of simplicial complexes can be reduced to computing barcodes for a certain *non-zigzag* filtration of Δ-complexes. The resulting algorithm called FastZigzag is more efficient considering that standard (non-zigzag) persistence admits faster algorithms [2, 3, 4, 5, 10, 11, 29] in practice. We confirm the efficiency with experiments in Section 3.5.

3.1   Overview

Given a *simplex-wise* zigzag filtration

$$\mathcal{F} : \varnothing = K_0 \xleftrightarrow{\sigma_0} K_1 \xleftrightarrow{\sigma_1} \cdots \xleftrightarrow{\sigma_{m-1}} K_m = \varnothing$$

of simplicial complexes as input, the FastZigzag algorithm has the following main procedure:
1. Convert $\mathcal{F}$ into a *non-repetitive* zigzag filtration of Δ-complexes.
2. Convert the non-repetitive filtration to an *up-down* filtration.
3. Convert the up-down filtration to a *non-zigzag* filtration with the help of an *extended persistence* filtration.
4. Compute the standard persistence barcode, which is then converted to the barcode for the input filtration based on rules given in Proposition 15 and 19.
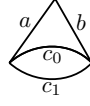
Figure 2 The Δ-complex resulting from performing an inward switch around $\hat{K}_4$ for the example shown in Figure 5.

Step 1 is achieved by simply treating each repeatedly added simplex in $\mathcal{F}$ as a new cell in the converted filtration (see also [25]). Throughout the section, we denote the converted non-repetitive, *cell-wise* filtration as

$$\hat{\mathcal{F}} : \varnothing = \hat{K}_0 \xleftrightarrow{\hat{\sigma}_0} \hat{K}_1 \xleftrightarrow{\hat{\sigma}_1} \cdots \xleftrightarrow{\hat{\sigma}_{m-1}} \hat{K}_m = \varnothing.$$

Notice that each $\hat{K}_i$ in $\hat{\mathcal{F}}$ is homeomorphic to K_i in $\mathcal{F}$, and hence $\mathsf{Pers}_*(\mathcal{F}) = \mathsf{Pers}_*(\hat{\mathcal{F}})$. However, we get an important difference between $\mathcal{F}$ and $\hat{\mathcal{F}}$ by treating the simplicial complexes as Δ-complexes. For example, in Figure 5 presented later in this section, the first addition of edge c in $\mathcal{F}$ corresponds to a cell c_0 in $\hat{\mathcal{F}}$ and its second addition in $\mathcal{F}$ corresponds to a cell c_1. Performing an inward switch around $\hat{K}_4$ (switching $\xleftarrow{c_0}$ and $\xrightarrow{c_1}$) turns $\hat{K}_4$ into a Δ-complex as shown in Figure 2. However, we cannot perform such a switch in $\mathcal{F}$ which consists of simplicial complexes, because diamond switches require the switched simplices or cells to be different (see Definition 5).

In Section 3.2 and 3.3, we provide details for Step 2 and 3 as well as propositions for converting barcodes mentioned in Step 4. We summarize the filtration converting process in Section 3.4 by providing pseudocodes (Algorithm 3.1) and examples (Figure 5 and 6).

3.2 Conversion to up-down filtration

▶ **Proposition 9.** *For the filtration $\hat{\mathcal{F}}$, there is a cell-wise* up-down *filtration*

$$\mathcal{U} : \varnothing = L_0 \hookrightarrow L_1 \hookrightarrow \cdots \hookrightarrow L_n \hookleftarrow L_{n+1} \hookleftarrow \cdots \hookleftarrow L_{2n} = \varnothing$$

derived from $\hat{\mathcal{F}}$ by a sequence of inward switches. Note that $m = 2n$

Proof. Let $\hat{K}_i \xleftarrow{\hat{\sigma}_i} \hat{K}_{i+1}$ be the first deletion in $\hat{\mathcal{F}}$ and $\hat{K}_j \xrightarrow{\hat{\sigma}_j} \hat{K}_{j+1}$ be the first addition after that. That is, $\hat{\mathcal{F}}$ is of the form

$$\hat{\mathcal{F}} : \hat{K}_0 \hookrightarrow \cdots \hookrightarrow \hat{K}_i \xleftarrow{\hat{\sigma}_i} \hat{K}_{i+1} \xleftarrow{\hat{\sigma}_{i+1}} \cdots \xleftarrow{\hat{\sigma}_{j-2}} \hat{K}_{j-1} \xleftarrow{\hat{\sigma}_{j-1}} \hat{K}_j \xrightarrow{\hat{\sigma}_j} \hat{K}_{j+1} \leftrightarrow \cdots \leftrightarrow \hat{K}_m.$$

Since $\hat{\mathcal{F}}$ is non-repetitive, we have $\hat{\sigma}_{j-1} \neq \hat{\sigma}_j$. So we can switch $\xleftarrow{\hat{\sigma}_{j-1}}$ and $\xrightarrow{\hat{\sigma}_j}$ (which is an inward switch) to derive a filtration

$$\hat{K}_0 \hookrightarrow \cdots \hookrightarrow \hat{K}_i \xleftarrow{\hat{\sigma}_i} \hat{K}_{i+1} \xleftarrow{\hat{\sigma}_{i+1}} \cdots \xleftarrow{\hat{\sigma}_{j-2}} \hat{K}_{j-1} \xrightarrow{\hat{\sigma}_j} \hat{K}_j' \xleftarrow{\hat{\sigma}_{j-1}} \hat{K}_{j+1} \leftrightarrow \cdots \leftrightarrow \hat{K}_m.$$

We then continue performing such inward switches (e.g., the next switch is on $\xleftarrow{\hat{\sigma}_{j-2}}$ and $\xrightarrow{\hat{\sigma}_j}$) to derive a filtration

$$\hat{\mathcal{F}}' : \hat{K}_0 \hookrightarrow \cdots \hookrightarrow \hat{K}_i \xrightarrow{\hat{\sigma}_j} \hat{K}_{i+1}' \xleftarrow{\hat{\sigma}_i} \cdots \xleftarrow{\hat{\sigma}_{j-3}} \hat{K}_{j-1}' \xleftarrow{\hat{\sigma}_{j-2}} \hat{K}_j' \xleftarrow{\hat{\sigma}_{j-1}} \hat{K}_{j+1} \leftrightarrow \cdots \leftrightarrow \hat{K}_m.$$

Note that from $\hat{\mathcal{F}}$ to $\hat{\mathcal{F}}'$, the up-down "prefix" grows longer. We can repeat the above operations on the newly derived $\hat{\mathcal{F}}'$ until the entire filtration turns into an up-down one. ◀

Throughout the section, let

$$\mathcal{U} : \varnothing = L_0 \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_{n-1}} L_n \xleftarrow{\tau_n} \cdots \xleftarrow{\tau_{2n-1}} L_{2n} = \varnothing$$

be the up-down filtration for $\hat{\mathcal{F}}$ as described in Proposition 9, where $m = 2n$. We also let $\hat{K} = L_n$.

In a cell-wise filtration, for a cell σ, let its addition (insertion) be denoted as $\downarrow\sigma$ and its deletion (removal) be denoted as $\uparrow\sigma$. From the proof of Proposition 9, we observe the following: during the transition from $\hat{\mathcal{F}}$ to $\mathcal{U}$, for any two additions $\downarrow\sigma$ and $\downarrow\sigma'$ in $\hat{\mathcal{F}}$ (and similarly for deletions), if $\downarrow\sigma$ is before $\downarrow\sigma'$ in $\hat{\mathcal{F}}$, then $\downarrow\sigma$ is also before $\downarrow\sigma'$ in $\mathcal{U}$. We then have the following fact:

▶ **Fact 10.** *Given the filtration $\hat{\mathcal{F}}$, to derive $\mathcal{U}$, one only needs to scan $\hat{\mathcal{F}}$ and list all the additions first and then the deletions, following the order in $\hat{\mathcal{F}}$.*

▶ **Remark 11.** Figure 3 gives an example of $\hat{\mathcal{F}}$ and its corresponding $\mathcal{U}$, where the additions and deletions in $\hat{\mathcal{F}}$ and $\mathcal{U}$ follow the same order.

▶ **Definition 12** (Creator and destroyer). *For any interval $[b,d] \in \mathsf{Pers}_*(\hat{\mathcal{F}})$, if $\hat{K}_{b-1} \xleftrightarrow{\hat{\sigma}_{b-1}} \hat{K}_b$ is forward (resp. backward), we call $\downarrow\hat{\sigma}_{b-1}$ (resp. $\uparrow\hat{\sigma}_{b-1}$) the **creator** of $[b,d]$. Similarly, if $\hat{K}_d \xleftrightarrow{\hat{\sigma}_d} \hat{K}_{d+1}$ is forward (resp. backward), we call $\downarrow\hat{\sigma}_d$ (resp. $\uparrow\hat{\sigma}_d$) the **destroyer** of $[b,d]$.*

By inspecting the interval mapping in the Diamond Principle, we have the following fact:

▶ **Proposition 13.** *For two cell-wise filtrations $\mathcal{L}, \mathcal{L}'$ related by a Mayer-Vietoris diamond, any two intervals of $\mathsf{Pers}_*(\mathcal{L})$ and $\mathsf{Pers}_*(\mathcal{L}')$ mapped by the Diamond Principle have the same set of creator and destroyer, though the creator and destroyer may swap. This observation combined with Proposition 9 implies that there is a bijection from $\mathsf{Pers}_*(\mathcal{U})$ to $\mathsf{Pers}_*(\hat{\mathcal{F}})$ s.t. every two corresponding intervals have the same set of creator and destroyer.*
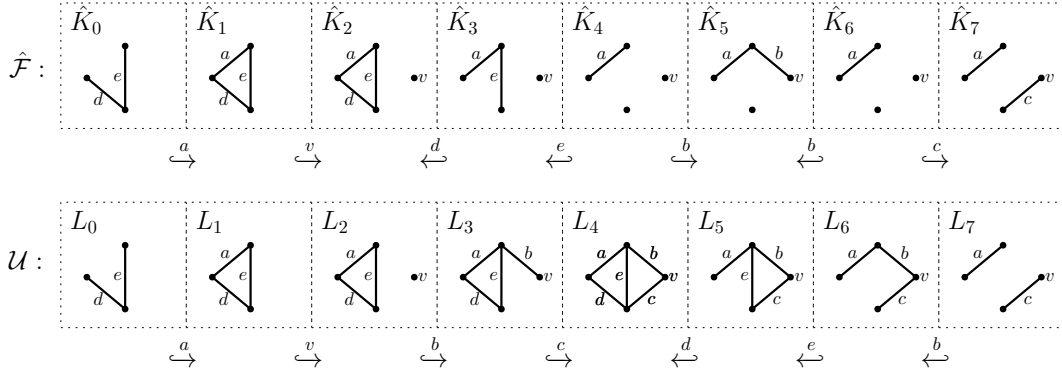
▶ **Remark 14.** The only time when the creator and destroyer swap in a Mayer-Vietoris diamond is when the interval $[j,j]$ for the upper filtration in Equation (1) turns into the same interval (of one dimension lower) for the lower filtration.

Consider the example in Figure 3 for an illustration of Proposition 13. In the example, $[1,2] \in \mathsf{Pers}_1(\hat{\mathcal{F}})$ corresponds to $[1,4] \in \mathsf{Pers}_1(\mathcal{U})$, where their creator is $\downarrow a$ and their destroyer is $\uparrow d$. Moreover, $[4,6] \in \mathsf{Pers}_0(\hat{\mathcal{F}})$ corresponds to $[4,5] \in \mathsf{Pers}_1(\mathcal{U})$. The creator of $[4,6] \in \mathsf{Pers}_0(\hat{\mathcal{F}})$ is $\uparrow e$ and the destroyer is $\downarrow c$. Meanwhile, $[4,5] \in \mathsf{Pers}_1(\mathcal{U})$ has the same set of creator and destroyer but the roles swap.

For any $\downarrow\sigma$ or $\uparrow\sigma$ in $\hat{\mathcal{F}}$, let $\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\sigma)$ or $\mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\sigma)$ denote the index (position) of the addition or deletion. For example, for an addition $\hat{K}_i \xrightarrow{\hat{\sigma}_i} \hat{K}_{i+1}$ in $\hat{\mathcal{F}}$, $\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\hat{\sigma}_i) = i$. Proposition 13 indicates the following explicit mapping from $\mathsf{Pers}_*(\mathcal{U})$ to $\mathsf{Pers}_*(\hat{\mathcal{F}})$:

▶ **Proposition 15.** *There is a bijection from $\mathsf{Pers}_*(\mathcal{U})$ to $\mathsf{Pers}_*(\hat{\mathcal{F}})$ which maps each $[b,d] \in \mathsf{Pers}_p(\mathcal{U})$ by the following rule:*

Type	Condition		Type	Interval in $\mathsf{Pers}_*(\hat{\mathcal{F}})$	Dim
closed-open	-	$\mapsto$	closed-open	$\left[\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_{b-1}) + 1, \mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_d)\right]$	p
open-closed	-	$\mapsto$	open-closed	$\left[\mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_{b-1}) + 1, \mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_d)\right]$	p
closed-closed	$\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_{b-1}) < \mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_d)$	$\mapsto$	closed-closed	$\left[\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_{b-1}) + 1, \mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_d)\right]$	p
	$\mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_{b-1}) > \mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_d)$	$\mapsto$	open-open	$\left[\mathtt{id}_{\hat{\mathcal{F}}}(\uparrow\tau_d) + 1, \mathtt{id}_{\hat{\mathcal{F}}}(\downarrow\tau_{b-1})\right]$	$p-1$

Figure 3 An example of filtration $\hat{\mathcal{F}}$ and its corresponding up-down filtration $\mathcal{U}$. For brevity, $\hat{\mathcal{F}}$ does not start and end with empty complexes (which can be treated as a truncated case).

▶ **Remark 16.** Notice that $\mathsf{Pers}_*(\mathcal{U})$ contains no open-open intervals. However, a closed-closed interval $[b, d] \in \mathsf{Pers}_p(\mathcal{U})$ turns into an open-open interval in $\mathsf{Pers}_{p-1}(\hat{\mathcal{F}})$ when $\mathrm{id}_{\hat{\mathcal{F}}}(\downarrow \tau_{b-1}) > \mathrm{id}_{\hat{\mathcal{F}}}(\uparrow \tau_d)$. Such a change happens when a closed-closed interval turns into a single point interval $[j, j]$ during the sequence of outward switches, after which the closed-closed interval $[j, j]$ becomes an open-open interval $[j, j]$ with a dimension shift (see Theorem 7).

▶ **Remark 17.** Although it may take $O(m^2)$ diamond switches to go from $\hat{\mathcal{F}}$ to $\mathcal{U}$ or from $\mathcal{U}$ to $\hat{\mathcal{F}}$ as indicated in Proposition 9, we observe that these switches do not need to be actually executed in the algorithm. To convert the intervals in $\mathsf{Pers}_*(\mathcal{U})$ to those in $\mathsf{Pers}_*(\hat{\mathcal{F}})$, we only need to follow the mapping in Proposition 15, which takes constant time per interval.

We can take the example in Figure 3 for the mapping in Proposition 15. The interval $[4, 5] \in \mathsf{Pers}_1(\mathcal{U})$ is a closed-closed one whose creator is $\downarrow c$ and destroyer is $\uparrow e$. We have that $\mathrm{id}_{\hat{\mathcal{F}}}(\downarrow c) = 6 > \mathrm{id}_{\hat{\mathcal{F}}}(\uparrow e) = 3$. So the corresponding interval in $\mathsf{Pers}_0(\hat{\mathcal{F}})$ is

$$[\mathrm{id}_{\hat{\mathcal{F}}}(\uparrow e) + 1, \mathrm{id}_{\hat{\mathcal{F}}}(\downarrow c)] = [4, 6].$$

3.3 Conversion to non-zigzag filtration

We first convert the up-down filtration $\mathcal{U}$ to an extended persistence [12] filtration $\mathcal{E}$, which is then easily converted to an (absolute) non-zigzag filtration using the "coning" technique [12].

Inspired by the Mayer-Vietoris pyramid in [8], we relate $\mathsf{Pers}_*(\mathcal{U})$ to the barcode of the filtration $\mathcal{E}$ defined as:

$$\mathcal{E} : \varnothing = L_0 \hookrightarrow \cdots \hookrightarrow L_n = (\hat{K}, L_{2n}) \hookrightarrow (\hat{K}, L_{2n-1}) \hookrightarrow \cdots \hookrightarrow (\hat{K}, L_n) = (\hat{K}, \hat{K})$$

where $L_n = \hat{K} = (\hat{K}, L_{2n} = \varnothing)$. When denoting the persistence intervals of $\mathcal{E}$, we let the increasing index for the first half of $\mathcal{E}$ continue to the second half, i.e., $(\hat{K}, L_{2n-1})$ has index $n + 1$ and $(\hat{K}, L_n)$ has index $2n$. Then, it can be verified that an interval $[b, d] \in \mathsf{Pers}_*(\mathcal{E})$ for $b < n < d$ starts with the complex L_b and ends with $(\hat{K}, L_{3n-d})$.

▶ **Remark 18.** A filtration in extended persistence [12] is originally defined for a PL function f, where the first half is the lower-star filtration of f and the second half (the relative part) is derived from the upper-star filtration of f. The filtration $\mathcal{E}$ defined above is a generalization of the one in [12].

▶ **Proposition 19.** *There is a bijection from* $\mathsf{Pers}_*(\mathcal{E})$ *to* $\mathsf{Pers}_*(\mathcal{U})$ *which maps each* $[b, d] \in$ $\mathsf{Pers}_*(\mathcal{E})$ *of dimension* p *by the following rule*:

Type	Condition		Type	Interv. in $\mathsf{Pers}_*(\mathcal{U})$	Dim
Ord	$d < n$	$\mapsto$	closed-open	$[b, d]$	p
Rel	$b > n$	$\mapsto$	open-closed	$[3n - d, 3n - b]$	$p-1$
Ext	$b \leq n \leq d$	$\mapsto$	closed-closed	$[b, 3n - d - 1]$	p

▶ **Remark 20.** The types "Ord", "Rel", and "Ext" for intervals in $\mathsf{Pers}_*(\mathcal{E})$ are as defined in [12], which stand for intervals from the *ordinary* sub-barcode, the *relative* sub-barcode, and the *extended* sub-barcode.

▶ **Remark 21.** The above proposition can also be stated by associating the creators and destroyers as in Proposition 13 and 15. The association of additions in the first half of $\mathcal{U}$ and $\mathcal{E}$ is straightforward and the deletion of a σ in $\mathcal{U}$ is associated with the addition of σ (to the second complex in the pair) in $\mathcal{E}$. Then, corresponding intervals in $\mathsf{Pers}_*(\mathcal{E})$ and $\mathsf{Pers}_*(\mathcal{U})$ in the above proposition also have the same set of creators and destroyers. Combined with Proposition 13, we further have that intervals in $\mathsf{Pers}_*(\mathcal{F})$ and $\mathsf{Pers}_*(\mathcal{E})$ can be associated by a bijection where corresponding intervals have the same pairs of simplices though they may switch roles of being creators and destroyers.

Proof. We can build a Mayer-Vietoris pyramid relating the second half of $\mathcal{E}$ and the second half of $\mathcal{U}$ similar to the one in [8]. A pyramid for $n = 4$ is shown in Figure 4, where the second half of $\mathcal{E}$ is along the left side of the triangle and the second half of $\mathcal{U}$ is along the bottom. In Figure 4, we represent the second half of $\mathcal{E}$ and $\mathcal{U}$ in a slightly different way considering that $L_4 = \hat{K}$ and $L_8 = \varnothing$. Also, each vertical arrow indicates the addition of a simplex in the second complex of the pair and each horizontal arrow indicates the deletion of a simplex in the first complex.

To see the correctness of the mapping, we first note that each square in the pyramid is a (more general version of) Mayer-Vietoris diamond as defined in [8]. Then, the mapping stated in the proposition can be verified using the Diamond Principle (Theorem 7). However, there is a quicker way to verify the mapping by observing the following: corresponding intervals in $\mathsf{Pers}_*(\mathcal{E})$ and $\mathsf{Pers}_*(\mathcal{U})$ have the same set of creator and destroyer if we ignore whether it is the addition or deletion of a simplex. For example, an interval in $\mathsf{Pers}_*(\mathcal{E})$ may be created by the addition of a simplex σ in the first half of $\mathcal{E}$ and destroyed by the addition of another simplex σ' in the second half of $\mathcal{E}$. Then, its corresponding interval in $\mathsf{Pers}_*(\mathcal{U})$ is also created by the addition of σ in the first half but destroyed by the *deletion* of σ' in the second half. Note that the dimension change for the case $b > n$ is caused by the swap of creator and destroyer. ◀

By Proposition 15 and 19, we only need to compute $\mathsf{Pers}_*(\mathcal{E})$ in order to compute $\mathsf{Pers}_*(\mathcal{F})$. The barcode of $\mathcal{E}$ can be computed using the "coning" technique [12], which converts $\mathcal{E}$ into an (absolute) non-zigzag filtration $\hat{\mathcal{E}}$. Specifically, let ω be a vertex different from all vertices in $\hat{K}$. For a p-cell σ of $\hat{K}$, we let $\omega \cdot \sigma$ denote the *cone* of σ, which is a $(p + 1)$-cell having cells $\{\sigma\} \cup \{\omega \cdot \tau \mid \tau \in \partial\sigma\}$ in its boundary. The *cone* $\omega \cdot L_i$ of a complex L_i consists of three parts: the vertex ω, L_i, and cones of all cells of L_i. The filtration $\hat{\mathcal{E}}$ is then defined as [12]:

$$\hat{\mathcal{E}} : L_0 \cup \{\omega\} \hookrightarrow \cdots \hookrightarrow L_n \cup \{\omega\} = \hat{K} \cup \omega \cdot L_{2n} \hookrightarrow \hat{K} \cup \omega \cdot L_{2n-1} \hookrightarrow \cdots \hookrightarrow \hat{K} \cup \omega \cdot L_n$$

We have that $\mathsf{Pers}_*(\mathcal{E})$ equals $\mathsf{Pers}_*(\hat{\mathcal{E}})$ discarding the only infinite interval [12]. Note that if a cell σ is added (to the second complex) from $(\hat{K}, L_i)$ to $(\hat{K}, L_{i-1})$ in $\mathcal{E}$, then the cone $\omega \cdot \sigma$ is added from $\hat{K} \cup \omega \cdot L_i$ to $\hat{K} \cup \omega \cdot L_{i-1}$ in $\hat{\mathcal{E}}$.
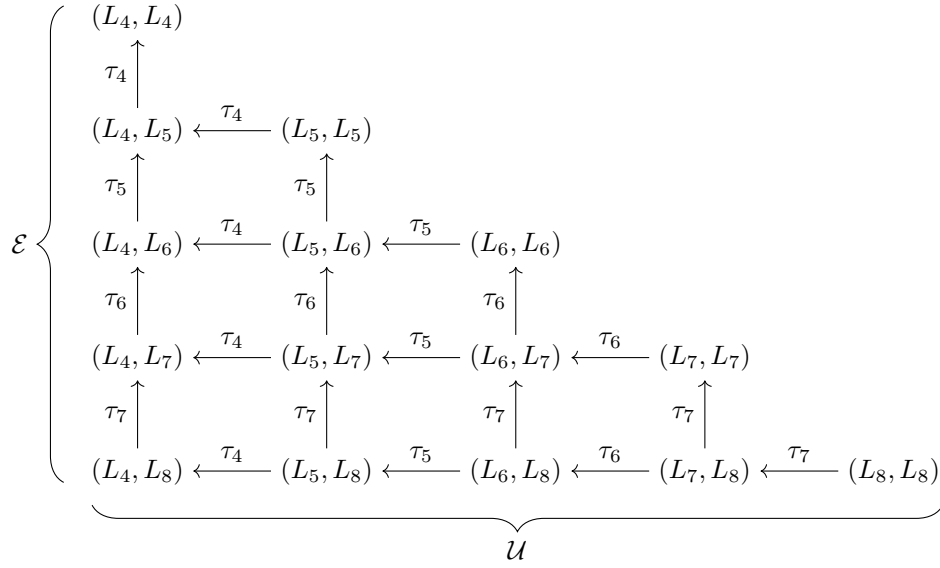
Figure 4 A Mayer-Vietoris pyramid relating the second half of $\mathcal{E}$ and $\mathcal{U}$ for $n = 4$.

3.4 Summary of filtration conversion

We summarize the filtration conversion process described in this section in Algorithm 3.1, in which we assume that each simplex in $\mathcal{F}$ is given by its set of vertices. The converted standard filtration $\hat{\mathcal{E}}$ is represented by its boundary matrix D, whose columns (and equivalently the chains they represent) are treated as sets of identifiers of the boundary cells. Algorithm 3.1 also maintains the following data structures:

- `cid` denotes the map from a simplex σ to the identifier of the *most recent* copy of cell corresponding to σ.
- `del_list` denotes the list of cell identifiers deleted in the input filtration.
- `cone_id` denotes the map from the identifier of a cell to that of its coned cell.

Subroutine CELLBOUNDARY in Line 8 converts boundary simplices of σ_i to a column of cell identifiers based on the map `cid`. Subroutine CONEDCELLBOUNDARY in Line 16 returns boundary column for the cone of the cell identified by `del_id`.

We provide an example of the up-down cell-wise filtration $\mathcal{U}$ built from a given simplex-wise filtration $\mathcal{F}$ in Figure 5. In the example, edge c and triangle t are repeatedly added twice in $\mathcal{F}$, and therefore each correspond to two copies of cells in $\mathcal{U}$. We provide another example of a complete conversion from a given zigzag filtration to a non-zigzag filtration in Figure 6.

With the CONVERTFILT subroutine, Algorithm 3.2 provides a concise summary of FASTZIGZAG. Given that for a filtration $\mathcal{F}$ of length m, CONVERTFILT takes $O(m)$ time and CONVERTBARCODE takes $O(1)$ time per bar, we now have the following conclusion:

▶ **Theorem 22.** *Given a simplex-wise zigzag filtration $\mathcal{F}$ with length m, FASTZIGZAG computes $\mathsf{Pers}_*(\mathcal{F})$ in time $T(m) + O(m)$, where $T(m)$ is the time used for computing the barcode of a non-zigzag cell-wise filtration with length m.*

▶ **Remark 23.** Theoretically, $T(m) = O(m^\omega)$ [25], where $\omega < 2.37286$ is the matrix multiplication exponent [1]. So the theoretical complexity of FASTZIGZAG is $O(m^\omega)$.

■ **Algorithm 3.1** Pseudocode for converting input filtration.

1: **procedure** CONVERTFILT($\mathcal{F}$)
2:      initialize boundary matrix D, cell-id map `cid`, deleted cell list `del_list` as empty
3:      append an empty column to D representing vertex ω for coning
4:      `id` ← 1   ▷ variable keeping track of id for cells
5:      **for each** $K_i \xleftrightarrow{\sigma_i} K_{i+1}$ in $\mathcal{F}$ **do**
6:          **if** σ_i is being inserted **then**
7:              `cid`[σ_i] = `id`   ▷ get a new cell as a copy of simplex σ_i
8:              `col` ← CELLBOUNDARY(σ_i, `cid`)
9:              append `col` to D
10:             `id` ← `id` + 1
11:         **else**
12:             append `cid`[σ_i] to `del_list`
13:     initialize map `cone_id` as empty   ▷ `cone_id` tracks id for coned cells
14:     **for each** `del_id` in `del_list` (accessed reversely) **do**
15:         `cone_id`[`del_id`] ← `id`   ▷ get a new coned cell
16:         `col` ← CONEDCELLBOUNDARY(`del_id`, D, `cone_id`)
17:         append `col` to D
18:         `id` ← `id` + 1
19:   **return** D
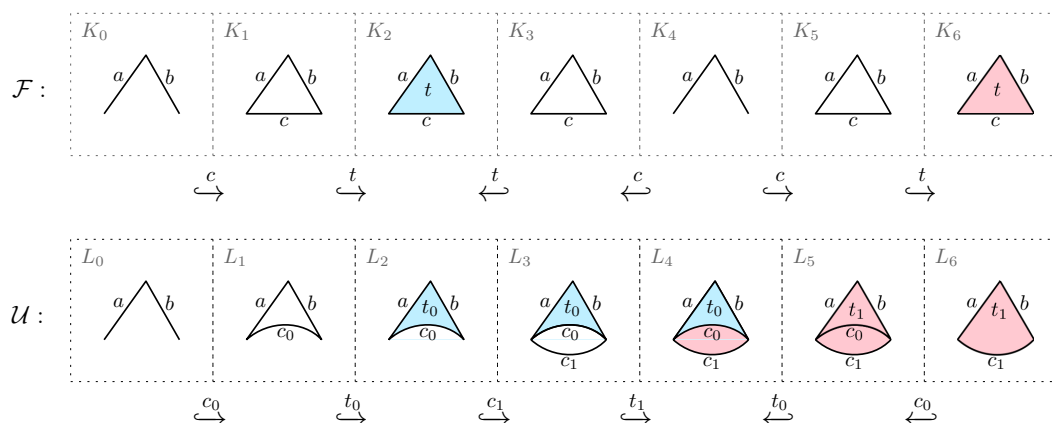
■ **Figure 5** An example of an up-down cell-wise filtration $\mathcal{U}$ built from a given simplex-wise filtration $\mathcal{F}$. For brevity, $\mathcal{F}$ does not start and end with empty complexes. The final conversion to $\hat{\mathcal{E}}$ is not shown for this example due to page-width constraint. A complete conversion for a smaller example is shown in Figure 6.

■ **Algorithm 3.2** Pseudocode for FASTZIGZAG.

1: **procedure** FASTZIGZAG($\mathcal{F}$)
2:      D ← CONVERTFILT($\mathcal{F}$)
3:      B ← COMPUTEBARCODE(D)
4:      B' ← CONVERTBARCODE(B)
5:      **return** B'
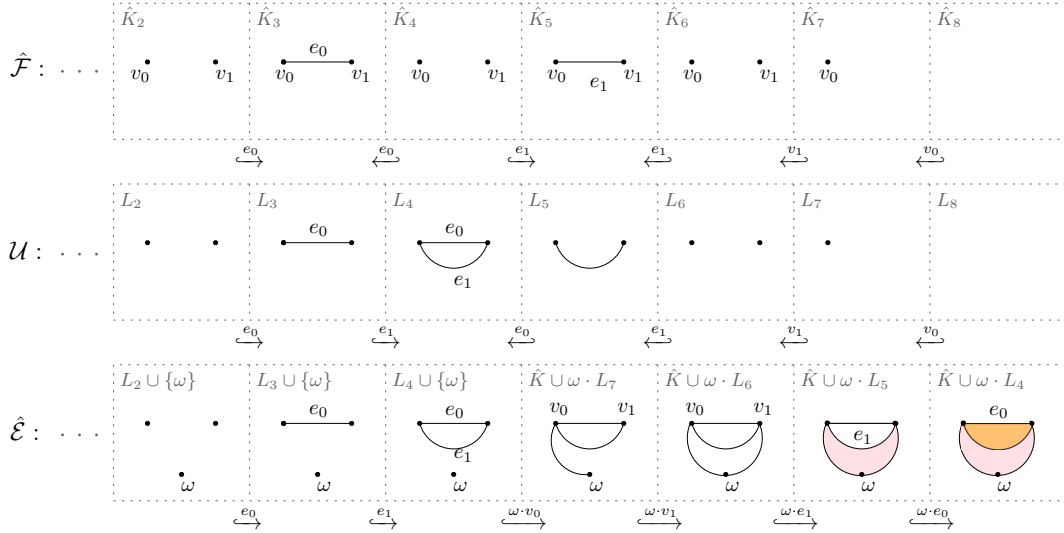
Figure 6 An example of converting a zigzag filtration $\hat{\mathcal{F}}$ to a non-zigzag filtration.

3.5   Experiments

We implement the FASTZIGZAG algorithm described in this section and compare the performance with DIONYSUS2 [26] (implementing the algorithm in [8]) and GUDHI[1] [28] (implementing the algorithm in [22, 24]). When implementing FASTZIGZAG, we utilize the PHAT [4] software for computing non-zigzag persistence. Our implementation is publicly available through: `https://github.com/taohou01/fzz`.

To test the performance, we generate eleven simplex-wise filtrations of similar lengths (5~6 millions; see Table 1). The reason for using filtrations of similar lengths is to test the impact of *repetitiveness* on the performance for different algorithms, where repetitiveness is the average times a simplex is repeatedly added in a filtration (e.g., repetitiveness being 1 means that the filtration is non-repetitive). We utilize three different approaches for generating the filtrations:

- The two non-repetitive filtrations (No. 1 and 2) are generated by first taking a simplicial complex with vertices in $\mathbb{R}^3$, and then taking the height function h along a certain axis. After this, we build an up-down filtration for the complex where the first half is the lower-star filtration of h and the second half is the (reversed) upper-star filtration of h. We then randomly perform outward switches on the up-down filtration to derive a non-repetitive filtration. Note that the simplicial complex is derived from a triangular mesh supplemented by a Vietoris-Rips complex on the vertices; one triangular mesh (Dragon) is downloaded from the Stanford Computer Graphics Laboratory.
- Filtration No. 3 – 8 are generated from a sequence of edge additions and deletions, for which we then take the *clique complex* (up to a certain dimension) for each edge set in the sequence. The edge sequence is derived by randomly adding and deleting edges for a set of points.
- The remaining filtrations (No. 9 – 11) are the *oscillating Rips zigzag* [27] generated from point clouds of size 2000 – 4000 sampled from some triangular meshes (Space Shuttle from an online repository[2]; Bunny and Dragon from the Stanford Computer Graphics Laboratory).

---

[1]   The code is shared by personal communication.

[2]   Ryan Holmes: `http://www.holmes3d.net/graphics/offfiles/`

Table 1 lists running time of the three algorithms on all filtrations, where the length, maximum dimension (D), repetitiveness (Rep), and maximum complex size (MaxK) are also provided for each filtration. From Table 1, we observe that FASTZIGZAG (T_{FZZ}) consistently achieves the best running time across all inputs, with significant speedups (see column "SU" in Table 1). The speedup is calculated as the min-time of DIONYSUS2 and GUDHI divided by the time of FASTZIGZAG. Notice that since GUDHI only takes a sequence of edge additions and deletions as input (and builds clique complexes on-the-fly), we do not run GUDHI on the first two inputs in Table 1, which are only given as simplex-wise filtrations. We also observe that the speedup of FASTZIGZAG tends to be less prominent as the repetitiveness increases. This is because higher repetitiveness leads to smaller max/average complex size in the input zigzag filtration, so that algorithms directly working on the input filtration could have less processing time [8, 22, 24]. On the other hand, the complex size in the converted non-zigzag filtration that FASTZIGZAG works on is always increasing.

Table 1 Running time of DIONYSUS2, GUDHI, and FASTZIGZAG on different filtrations of similar lengths with various repetitiveness. All tests were run on a desktop with Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz, 16GB memory, and Linux OS.

No.	Length	D	Rep	MaxK	T_{DIO2}	T_{GUDHI}	T_{FZZ}	SU
1	5,260,700	5	1.0	883,350	2h02m46.0s	—	8.9s	873
2	5,254,620	4	1.0	1,570,326	19m36.6s	—	11.0s	107
3	5,539,494	5	1.3	1,671,047	3h05m00.0s	45m47.0s	3m20.8s	13.7
4	5,660,248	4	2.0	1,385,979	2h59m57.0s	29m46.7s	4m59.5s	6.0
5	5,327,422	4	3.5	760,098	43m54.8s	10m35.2s	3m32.1s	3.0
6	5,309,918	3	5.1	523,685	5h46m03.0s	1h32m37.0s	19m30.2s	4.7
7	5,357,346	3	7.3	368,830	3h37m54.0s	57m28.4s	30m25.2s	1.9
8	6,058,860	4	9.1	331,211	53m21.2s	7m19.0s	3m44.4s	2.0
9	5,135,720	3	21.9	11,859	23.8s	15.6s	8.6s	1.9
10	5,110,976	3	27.7	11,435	36.2s	39.9s	8.5s	4.3
11	5,811,310	4	44.2	7,782	38.5s	36.9s	23.9s	1.5

Table 2 lists the memory consumption of the three algorithms. We observe that FASTZIGZAG tends to consume more memory than the other two on the non-repetitive filtrations (No. 1 and 2) and the random clique filtrations (No. 3 – 8). However, FASTZIGZAG is consistently achieving the best memory footprint on the oscillating Rips filtrations (No. 9 – 11) despite the high repetitiveness.

4   Conclusions

In this paper, we propose a zigzag persistence algorithm called FASTZIGZAG by first treating repeatedly added simplices in an input zigzag filtration as distinct copies and then converting the input filtration to a non-zigzag filtration. The barcode of the converted non-zigzag filtration can then be easily mapped back to barcode of the input zigzag filtration. The efficiency of our algorithm is confirmed by experiments. This research also brings forth the following open questions:

■ **Table 2** Memory consumption (in gigabytes) of the three algorithms on all filtrations.

No.	Length	Rep	MaxK	M_{Dio2}	M_{Gudhi}	M_{FZZ}
1	5,260,700	1.0	883,350	3.23	—	0.59
2	5,254,620	1.0	1,570,326	3.93	—	0.61
3	5,539,494	1.3	1,671,047	15.52	13.49	9.76
4	5,660,248	2.0	1,385,979	7.64	8.43	11.04
5	5,327,422	3.5	760,098	3.27	3.40	6.22
6	5,309,918	5.1	523,685	4.94	5.27	10.23
7	5,357,346	7.3	368,830	4.03	3.91	8.19
8	6,058,860	9.1	331,211	2.12	1.48	3.68
9	5,135,720	21.9	11,859	0.92	0.47	0.50
10	5,110,976	27.7	11,435	0.88	0.48	0.47
11	5,811,310	44.2	7,782	0.95	0.60	0.51

✎ Parallel versions [9, 29] of the algorithms for computing standard and zigzag exist. While the computation of standard persistence in our FastZigzag algorithm can directly utilize the existing parallelization techniques, we ask if the conversions done in FastZigzag can be efficiently parallelized. Such an extension can provide further speedups by harnessing multi-cores.

✎ While persistence intervals are important topological descriptors, their *representatives* also reveal critical information (e.g., a recently proposed algorithm [16] for updating zigzag barcodes over local changes uses representatives explicitly). Can the FastZigzag algorithm be adapted so that representatives for the input zigzag filtration are recovered from representatives for the converted non-zigzag filtration?

── **References** ──

1 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

2 Ulrich Bauer. Ripser: Efficient computation of vietoris–rips persistence barcodes. *Journal of Applied and Computational Topology*, 5(3):391–423, 2021.

3 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological methods in data analysis and visualization III*, pages 103–117. Springer, 2014.

4 Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat – persistent homology algorithms toolbox. *Journal of Symbolic Computation*, 78:76–90, 2017.

5 Jean-Daniel Boissonnat, Tamal K. Dey, and Clément Maria. The compressed annotation matrix: An efficient data structure for computing persistent cohomology. *Algorithmica*, 73(3):607–619, 2015. doi:10.1007/s00453-015-9999-4.

6 Gunnar Carlsson and Vin de Silva. Zigzag persistence. *Foundations of Computational Mathematics*, 10(4):367–405, 2010.

7 Gunnar Carlsson, Vin de Silva, Sara Kališnik, and Dmitriy Morozov. Parametrized homology via zigzag persistence. *Algebraic & Geometric Topology*, 19(2):657–700, 2019.

8   Gunnar Carlsson, Vin de Silva, and Dmitriy Morozov. Zigzag persistent homology and real-valued functions. In *Proceedings of the Twenty-Fifth Annual Symposium on Computational Geometry*, pages 247–256, 2009.

9   Gunnar Carlsson, Anjan Dwaraknath, and Bradley J. Nelson. Persistent and zigzag homology: A matrix factorization viewpoint. *arXiv preprint*, 2019. `arXiv:1911.10693`.

10   Chao Chen and Michael Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, volume 11, pages 197–200, 2011.

11   Chao Chen and Michael Kerber. An output-sensitive algorithm for persistent homology. *Comput. Geom.: Theory and Applications*, 46(4):435–447, 2013. `doi:10.1016/j.comgeo.2012.02.010`.

12   David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Extending persistence using Poincaré and Lefschetz duality. *Foundations of Computational Mathematics*, 9(1):79–103, 2009.

13   Vin de Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Dualities in persistent (co)homology. *Inverse Problems*, 27(12):124003, 2011.

14   Harm Derksen and Jerzy Weyman. Quiver representations. *Notices of the AMS*, 52(2):200–206, 2005.

15   Tamal K. Dey and Tao Hou. Computing zigzag persistence on graphs in near-linear time. In *37th International Symposium on Computational Geometry, SoCG 2021*, volume 189 of *LIPIcs*, pages 30:1–30:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

16   Tamal K. Dey and Tao Hou. Updating zigzag persistence and maintaining representatives over changing filtrations. *arXiv preprint*, 2021. `arXiv:2112.02352`.

17   Tamal K. Dey, Woojin Kim, and Facundo Mémoli. Computing generalized rank invariant for 2-parameter persistence modules via zigzag persistence and its applications. In *38th International Symposium on Computational Geometry, SoCG 2022*, volume 224 of *LIPIcs*, pages 34:1–34:17, 2022.

18   Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 454–463. IEEE, 2000.

19   Peter Gabriel. Unzerlegbare Darstellungen I. *Manuscripta Mathematica*, 6(1):71–103, 1972.

20   Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

21   Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, 519(3):97–125, 2012.

22   Clément Maria and Steve Y. Oudot. Zigzag persistence via reflections and transpositions. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 181–199. SIAM, 2014.

23   Clément Maria and Steve Y. Oudot. Computing zigzag persistent cohomology. *arXiv preprint*, 2016. `arXiv:1608.06039`.

24   Clément Maria and Hannah Schreiber. Discrete morse theory for computing zigzag persistence. In *Workshop on Algorithms and Data Structures*, pages 538–552. Springer, 2019.

25   Nikola Milosavljević, Dmitriy Morozov, and Primoz Skraba. Zigzag persistent homology in matrix multiplication time. In *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry*, pages 216–225, 2011.

26   Dmitriy Morozov. `Dionysus2`. URL: `https://www.mrzv.org/software/dionysus2/`.

27   Steve Y. Oudot and Donald R. Sheehy. Zigzag zoology: Rips zigzags for homology inference. *Foundations of Computational Mathematics*, 15(5):1151–1186, 2015.

28   The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015. URL: `http://gudhi.gforge.inria.fr/doc/latest/`.

29   Simon Zhang, Mengbai Xiao, and Hao Wang. GPU-accelerated computation of Vietoris-Rips persistence barcodes. In *36th International Symposium on Computational Geometry (SoCG 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

30   Afra Zomorodian and Gunnar Carlsson. Computing persistent homology. *Discrete & Computational Geometry*, 33(2):249–274, 2005.