Persistent (Co)Homology in Matrix Multiplication Time

Dmitriy Morozov
Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Primoz Skraba
Queen Mary University of London, UK

Abstract

Most algorithms for computing persistent homology do so by tracking cycles that represent homology classes. There are many choices of such cycles, and specific choices have found different uses in applications. Although it is known that persistence diagrams can be computed in matrix multiplication time for the more general case of zigzag persistent homology [21], it is not clear how to extract cycle representatives, especially if specific representatives are desired. In this paper, we provide the same matrix multiplication bound for computing representatives for the two choices common in applications in the case of ordinary persistent (co)homology. We first provide a fast version of the reduction algorithm, which is simpler than the algorithm in [21], but returns a different set of representatives than the standard algorithm [15]. We then give a fast version of a variant called the row algorithm [10], which returns the same representatives as the standard algorithm.

2012 ACM Subject Classification Computing methodologies → Algebraic algorithms; Mathematics of computing → Algebraic topology; Computing methodologies → Linear algebra algorithms; Theory of computation → Computational geometry

Keywords and phrases persistent homology, matrix multiplication, cycle representatives

Digital Object Identifier 10.4230/LIPIcs.SoCG.2025.68

Related Version *Full Version*: `https://arxiv.org/abs/2412.02591` [23]

Funding *Dmitriy Morozov*: supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
Primoz Skraba: supported by the EPSRC AI Hub on Mathematical Foundations of Intelligence: An “Erlangen Programme” for AI No. EP/Y028872/1.

1 Introduction

Persistent homology [15] is one of the core techniques in topological data analysis. Because of its theoretical importance as well as its wide use in applications, a lot of work has been dedicated to its efficient computation both in theory [21, 10, 7, 4, 20] and in practice [2, 3, 1, 18, 17, 26]. On the theoretical side, the connection between persistent homology and Gaussian elimination was established early on [27], and as a result it has been long accepted that persistence can be computed in matrix multiplication time. Milosavljevic et al. [21] gave an explicit algorithm for the more general case of zigzag persistence [5, 6]. The main difficulty in applying standard linear algebra techniques is the strict constraint on the row and column ordering of the boundary matrix.

As persistent homology has evolved as a field, the significance of its formulation as an $\mathbf{R} = \mathbf{D}\mathbf{V}$ decomposition [8] of the input boundary matrix $\mathbf{D}$ has become clear. The cycles and chains recovered from this decomposition found many uses in attributing topological features to the input data [11, 9, 24]. However, it is not immediately clear from the algorithm in [21] how to recover the matrices $\mathbf{R}$ and $\mathbf{V}$ necessary in applications. Rather than deconstructing the algorithm of Milosavljevic et al. [21], we simplify it for the case of ordinary persistence.

We describe how to compute two different forms of the $\mathbf{R} = \mathbf{DV}$ decomposition that come up in applications – lazy and exhaustive reductions. This allows us to recover specific cycle representatives in matrix multiplication time. As a benefit, due to its simpler setting, the algorithms in this paper are greatly simplified compared to [21]. Finally, using Dey and Hou's fast zigzag construction [13], which reduces computation of zigzag persistence to ordinary persistence; the algorithms in this paper give another way to compute zigzag persistence in matrix multiplication time.

2 Preliminaries

We are concerned with the persistent homology of a filtration, an increasing sequence of simplicial complexes indexed by the natural numbers $\mathbb{N}$:

$$\emptyset = K_0 \subset K_1 \subset \ldots \subset K_N = K$$

While persistent homology can be defined more generally, from an algorithmic perspective this is sufficiently general as nearly all considered cases may be reduced to this setting. We further assume that $|K_i - K_{i-1}| = 1$, or equivalently that the filtration is a total order, i.e., each step in the filtration adds a single cell. If the initial filtration does not satisfy this requirement, and defines only a partial order, we may always extend it to an arbitrary compatible total order, e.g., breaking ties by lexicographical ordering and dimension.

Notation. Throughout, we use the following notation:

- Matrices are denoted by bold capital letters, e.g., $\mathbf{M}$, with sub-matrices indexed by square brackets, e.g., the j-th column is $\mathbf{M}[\cdot, j]$, the i-th row is $\mathbf{M}[i, \cdot]$.
- Indexing starts at 1 and indices may be indexed by sets, e.g., $A = \{1, \ldots, k\}$, $\mathbf{M}[\cdot, A]$ refers to the first k columns. Note that the set may not refer to contiguous columns/rows.
- We consider permutations implicitly, so $(\mathbf{P} \cdot \mathbf{M})[A, \cdot]$ refers to the rows indexed **after** the permutation $\mathbf{P}$ is applied to matrix $\mathbf{M}$. For all the algorithms, performing an explicit permutation, then undoing it, does not affect the asymptotic running time.

2.1 Persistence Algorithm

For completeness, we recount the standard persistence algorithm, see [15], [27], and [16] for a more complete description. Let $\mathbf{D}$ denote the boundary operator, which we represent as an $n \times n$-matrix over some fixed (finite) field $\mathbb{F}$. The rows (top-down) and columns (left-right) are ordered by appearance in the filtration. Hence, the i-th row and column represents the simplex $\sigma = K_i - K_{i-1}$ and $\mathbf{D}[\cdot, i]$ represents the boundary of σ respectively. The goal is to compute the **RV**-decomposition [8]. That is, find matrices $\mathbf{R}$ and $\mathbf{V}$ such that

$$\mathbf{R} = \mathbf{D} \cdot \mathbf{V},$$

where $\mathbf{R}$ is a matrix in reduced column echelon form and $\mathbf{V}$ is a full-rank upper-triangular matrix; we let $\mathbf{U} = \mathbf{V}^{-1}$. We introduce the helper function

$$\mathbf{low}(j) = \arg\max_i \mathbf{R}[i, j] \neq 0,$$

which is defined whenever $\mathbf{R}[\cdot, j] \neq \mathbf{0}$. When defined, it returns the largest row index i where the column is non-zero; we refer to these lowest non-zero elements as **pivots**. Throughout this paper, **low** always applies to $\mathbf{R}$. The persistence diagram is then given by:

$$\text{dgm} = \{(i, j) \mid i = \mathbf{low}(j)\} \cup \{(i, \infty) \mid \mathbf{R}[\cdot, i] = \mathbf{0} \text{ and } i \neq \mathbf{low}(j)\ \forall j\}. \tag{1}$$

Algorithm 1 Standard Persistence Algorithm (Lazy reduction).

$\mathbf{R} = \mathbf{D}$
$\mathbf{V} = \mathbf{I}_n, \mathbf{U} = \mathbf{I}_n$
for $j = 1$ to n **do**
  while $\mathbf{R}[\cdot, j] \neq \mathbf{0}$ and $\exists j' < j$ such that $\mathbf{low}(j) = \mathbf{low}(j')$ **do**
    $i \leftarrow \mathbf{low}(j)$
    $\alpha \leftarrow \mathbf{R}[i, j]/\mathbf{R}[i, j']$
    $\mathbf{R}[\cdot, j] \leftarrow \mathbf{R}[\cdot, j] - \alpha \cdot \mathbf{R}[\cdot, j']$
    $\mathbf{V}[\cdot, j] \leftarrow \mathbf{V}[\cdot, j] - \alpha \cdot \mathbf{V}[\cdot, j']$
    $\mathbf{U}[j', \cdot] \leftarrow \mathbf{U}[j', \cdot] + \alpha \cdot \mathbf{U}[j, \cdot]$ (equivalently, $\mathbf{U}[j', j] \leftarrow \alpha$)

The standard algorithm is a variant of Gaussian elimination. Algorithm 1, which we call the **lazy reduction**, reduces each column by considering pivots in the columns to the left. In each step of the loop, it removes the lowest non-zero entry until a new pivot is found or the entire column is zeroed out. A straightforward analysis gives a running time bound of $O(n^3)$. In [22], a filtration where this algorithm takes cubic time was given, showing that the bound is tight. The updates of matrix $\mathbf{V}$ follow those of matrix $\mathbf{R}$. The updates in matrix $\mathbf{U}$ undo the updates in $\mathbf{V}$ to maintain $\mathbf{U} \cdot \mathbf{V} = \mathbf{I}$. Because the columns of $\mathbf{R}$ are processed from left to right, during the update of matrix $\mathbf{U}$, the row $\mathbf{U}[j, \cdot]$ has a single (diagonal) element. Therefore, the update in $\mathbf{U}$ is equivalent to just setting the entry $\mathbf{U}[j', j]$ to α.

Algorithm 2 "Look-ahead" Variant of Standard Persistence (Exhaustive reduction).

$\mathbf{R} = \mathbf{D}$
$\mathbf{V} = \mathbf{I}_n, \mathbf{U} = \mathbf{I}_n$
for $j = 1$ to n **do**
  if $\mathbf{low}(j)$ is defined **then**
    $i \leftarrow \mathbf{low}(j)$
    for $j' > j$ and $\mathbf{R}[\mathbf{low}(j), j'] \neq 0$ **do**
      $\alpha \leftarrow \mathbf{R}[i, j']/\mathbf{R}[i, j]$
      $\mathbf{R}[\cdot, j'] \leftarrow \mathbf{R}[\cdot, j'] - \alpha \cdot \mathbf{R}[\cdot, j]$
      $\mathbf{V}[\cdot, j'] \leftarrow \mathbf{V}[\cdot, j'] - \alpha \cdot \mathbf{V}[\cdot, j]$
      $\mathbf{U}[j, \cdot] \leftarrow \mathbf{U}[j, \cdot] + \alpha \cdot \mathbf{U}[j', \cdot]$ (**not** the same as $\mathbf{U}[j, j'] \leftarrow \alpha$!)

We introduce another variant of persistence computation used in applications. This one "looks ahead" and eliminates as many elements from the matrix as it can; so we call it the **exhaustive reduction**. In Algorithm 2, by the time a column is considered, it is already reduced. It is then applied to all columns to the right, zeroing out the entire row. In this way, when a column is processed, all previous pivots have already been applied. Because column $\mathbf{R}[\cdot, j']$ may have already been used to reduce another column $\mathbf{R}[\cdot, j'']$, when an update is applied to it, the row $\mathbf{U}[j', \cdot]$ need not consist of a single diagonal element. Hence, the row update in matrix $\mathbf{U}$ is not equivalent to just setting $\mathbf{U}[j, j']$ to α, like in the lazy version of the algorithm.

Lazy vs. Exhaustive. The algorithms based on the two reductions are in a sense the extreme cases: lazy reduction reduces columns only when necessary; and the exhaustive reduction reduces them as much as possible. To illustrate the difference, in the exhaustive reduction, the entire row to the right of a pivot will be zeroed out, whereas in the case of the lazy reduction, only those columns will be zeroed out which have a conflicting pivot in the same row. There are, of course, any number of possible other reductions between these – but we do not know of any applications that rely on them.

▶ Remark 1. As noted in [10, Section 3.4], persistent cohomology can be computed by performing the same algorithm on the anti-transpose of $\mathbf{D}$. As such all the algorithms in this paper can compute persistent cohomology.

2.2 Cycle Representatives

Given an $\mathbf{R} = \mathbf{DV}$ decomposition, we distinguish between two types of simplices in the filtration: *positive* simplices create new homology classes; *negative* simplices destroy them. From the matrix decomposition, one can recover a set of cycle representatives of persistent homology. There are two sources of this information, although with different content.

1. $\mathbf{V}$ matrix: The columns of positive simplices in $\mathbf{V}$ correspond to zero columns in the reduced matrix $\mathbf{R}$. These columns in $\mathbf{V}$ are cycles, by definition.
2. $\mathbf{R}$ matrix: The columns of negative simplices in $\mathbf{R}$ are non-zero. They store linear combinations of boundaries, which are cycles (at the time of their birth) that eventually die in the filtration.

The cycles one recovers from matrix $\mathbf{R}$ differ between the lazy and exhaustive reductions. Exhaustive reduction was called total reduction by Cohen-Steiner et al. [9]. They show that these cycles form a lexicographically optimal basis, which can be used to triangulate point cloud data. Nigmetov and Morozov [24] use the lazy reduction – specifically, the columns and rows from matrices $\mathbf{V}$ and $\mathbf{U}$ – to recover "critical sets," subsets of simplicial complexes affected by optimization with the loss formulated in terms of a persistence diagram.

Minimum Spanning Acycle Basis. The negative simplices in the filtration represent a minimal spanning acycle (MSA) [19, 25]. The columns in $\mathbf{V}$ obtained from either lazy or exhaustive reduction – or any reduction that never adds columns of positive simplices to other columns – have the form $\sum_{\tau \in MSA} \lambda_\tau \tau + \lambda_\sigma \sigma$. The support of the chain is one (positive) simplex σ plus a linear combination of simplices in the minimum spanning acycle.

▶ **Lemma 2.** *The representatives from* $\mathbf{V}$ *are the same for lazy and exhaustive reduction.*

Proof. Which columns of $\mathbf{R}$ are non-zero does not depend on the algorithm used; the set of negative simplices only depends on the order of the filtration. We observe that the boundaries of the negative simplices form a basis (since their pivots are in distinct rows). By definition, the boundary of a positive simplex may be expressed as a linear combination of boundaries of negative simplices. Since the boundaries of the negative simplices form a basis, this linear combination is unique. ◀

Death Basis. An alternative is to get cycle representatives directly from the $\mathbf{R}$ matrix. Its non-zero columns represent cycles because multiplying $\mathbf{D} \cdot \mathbf{R} = 0$. This follows directly since $\mathbf{R} = \mathbf{D} \cdot \mathbf{V}$, so $\mathbf{D} \cdot \mathbf{R} = \mathbf{D} \cdot \mathbf{D} \cdot \mathbf{V} = 0$, since $\mathbf{D} \cdot \mathbf{D} = 0$. These columns give cycle representatives for all finite classes, i.e., those homology classes that eventually die.

▶ Remark 3. Following [10], this may be used for cohomology – applying the algorithms to $\mathbf{D}^{\perp}$, the anti-transpose of $\mathbf{D}$ to obtain the decomposition $\mathbf{R}^{\perp} = \mathbf{D}^{\perp} \cdot \mathbf{V}^{\perp}$. The cocycle representatives are then given by columns in $\mathbf{V}^{\perp}$. To see that these are cocycles, observe that by construction, they map to zero via the coboundary operator $\mathbf{D}^{\perp}$ before the corresponding death time, i.e., below the pivot in the corresponding column of $\mathbf{R}^\perp$, as we have reversed the indexing with the anti-transpose. Likewise, the cocycle representatives of essential cocycles are given by the columns of $\mathbf{V}^\perp$ whose: corresponding columns in $\mathbf{R}^\perp$ are zero *and* they must not be coboundaries, i.e., the corresponding rows in $\mathbf{R}^\perp$ must not contain pivots.

3 Matrix Preliminaries

We recount a few classical results on matrix multiplication. We assume the matrices are over some field $\mathbb{F}$.

▶ **Lemma 4.** *Let $\mathbf{B}$ be an $n \times k$ matrix and $\mathbf{C}$ be a $k \times k$ matrix. The product $\mathbf{B} \cdot \mathbf{C}$ can be computed in $O(nk^{\omega-1})$-time.*

Proof. Divide $\mathbf{B}$ into $\lceil \frac{n}{k} \rceil$ sub-matrices $\mathbf{B}_i$ of size $k \times k$:

$$\mathbf{B} \cdot \mathbf{C} = \begin{bmatrix} \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_{\lceil n/k \rceil} \end{bmatrix} \cdot \mathbf{C} = \begin{bmatrix} \mathbf{B}_1 \cdot \mathbf{C} \\ \mathbf{B}_2 \cdot \mathbf{C} \\ \vdots \\ \mathbf{B}_{\lceil n/k \rceil} \cdot \mathbf{C} \end{bmatrix}$$

Each $\mathbf{B}_i \cdot \mathbf{C}$ is a product of two $k \times k$ matrices, which by definition takes $O(k^\omega)$ time. There are $\lceil \frac{n}{k} \rceil$ products to compute yielding a running time of $O\left(\frac{n+1}{k} k^\omega\right) = O(nk^{\omega-1})$. ◀

Column Operations via Matrix Multiplication. Here we relate matrix multiplication with the reduction steps used in the persistence algorithms, i.e., representing column operations via matrix multiplication. Assume that we have the following block matrix,

$$\left[\begin{array}{c|c} \mathbf{B} & \mathbf{C} \end{array} \right] = \left[\begin{array}{ccc|ccc} b(1) & \dots & b(n) & c(1) & \dots & c(m) \end{array} \right]$$

where $b(i)$ and $c(j)$ are (column) vectors. Reducing $c(j)$ which we denote $c'(j)$, using the vectors $b(i)$, is expressed as a linear combination

$$c'(j) = c(j) + \sum_i \lambda_i(j) b(i),$$

where $\lambda_i(j)$ are the coefficients. Reducing only the j-th column of $\mathbf{C}$ may be written as

$$\begin{bmatrix} \mathbf{B} & c'(j) \end{bmatrix} = \begin{bmatrix} \mathbf{B} & c(j) \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I} & \lambda(j) \\ 0 & \mathbf{1}_j \end{bmatrix},$$

where $\mathbf{1}_j$ is a column vector where the j-th entry is 1 and zero everywhere else. If we consider $\mathbf{\Lambda}[i, j] := \lambda_i(j)$, we can rewrite the reduction of $\mathbf{C}$ by

$$\begin{bmatrix} \mathbf{B} & \mathbf{C}' \end{bmatrix} = \begin{bmatrix} \mathbf{B} & \mathbf{C} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I} & \mathbf{\Lambda} \\ 0 & \mathbf{I} \end{bmatrix},$$

or equivalently, the reduced matrix is $\mathbf{C}' = \mathbf{C} - \mathbf{B} \cdot \mathbf{\Lambda}$.

▶ **Lemma 5.** *Multiplying a permutation matrix with an $n \times m$ matrix, takes $O(nm)$ time.*

Proof. As a permutation matrix rewrites each element once, the total number of elements gives the bound. ◀

4 Column Algorithms

Here we describe the algorithm in [21] for the special case of classical persistent homology. To aid in exposition, we first recall the **Schur complement**. Given a block matrix

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{C}_1 \\ \mathbf{B}_2 & \mathbf{C}_2 \end{bmatrix},$$

assuming $\mathbf{B}_2$ is non-singular, we can compute an updated matrix where the rows corresponding to $\mathbf{B}_2$ are zeroed out by the following transformation:

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{C}_1 \\ \mathbf{B}_2 & \mathbf{C}_2 \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\mathbf{B}_2^{-1}\mathbf{C}_2 \\ 0 & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{C}_1 - \mathbf{B}_1\mathbf{B}_2^{-1}\mathbf{C}_2 \\ \mathbf{B}_2 & 0 \end{bmatrix} \tag{2}$$

where $\mathbf{C}_1 - \mathbf{B}_1\mathbf{B}_2^{-1}\mathbf{C}_2$ is referred to as the Schur complement.

▶ **Observation 6.** *The matrix $\mathbf{B}_2^{-1}\mathbf{C}_2$ encodes the column operations required to zero out $\mathbf{C_2}$, that is, $\mathbf{B}_2^{-1}\mathbf{C}_2 = \mathbf{\Lambda}$.*

A key fact we will use in the algorithm is that matrix inversion has the same algorithmic complexity as matrix multiplication.

▶ **Theorem 7.** *Inverting a (square) non-singular (upper or lower) triangular matrix can be done in matrix multiplication time.*

Proof. See [23, Appendix] for a simplified proof for the special case of triangular matrices. ◀

4.1 Fast Exhaustive Algorithm

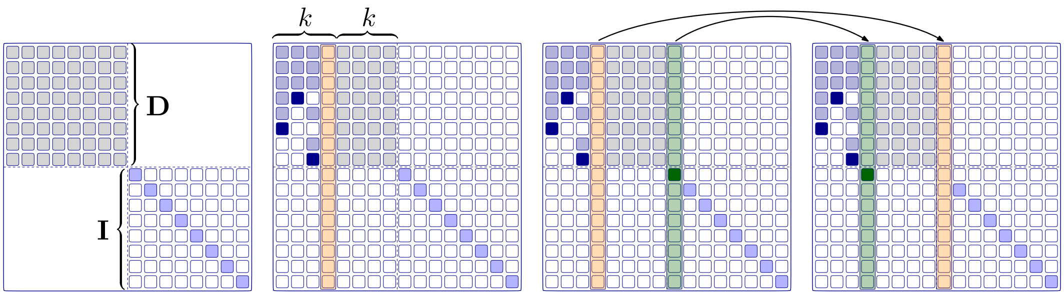
Algorithm 3 Column Algorithm($A = [i, j]$).

if $i = j$ **then** ▷ *leaf*
  if $\mathbf{R}[\cdot, i] = \mathbf{0}$ **then** ▷ *If no pivot exists*
    $\ell \leftarrow \arg\min_{\ell > n} \mathbf{R}[\cdot, \ell] \neq 0$ ▷ *find the next pivot in $\mathbf{I}_{m+n}$*
    Permute columns i and ℓ
    $Z \leftarrow Z \cup i$
else
  $B \leftarrow [i, (i+j)/2)$ ▷ *left child*
  $C \leftarrow [(i+j)/2, j]$ ▷ *right child*
  Recurse on B
  Apply updates from columns B to C:
    $L \leftarrow \langle \mathbf{low}(\ell) : \ell \in B \rangle$ ▷ *Implicit permutation; L preserves order*
    $\mathbf{\Lambda} \leftarrow \mathbf{R}[L, B]^{-1} \cdot \mathbf{R}[L, C]$ ▷ $\mathbf{A}_2^{-1}\mathbf{B}_2$
    $\mathbf{R}[\bar{L}, C] \leftarrow \mathbf{R}[\bar{L}, C] - \mathbf{R}[\bar{L}, B] \cdot \mathbf{\Lambda}$ ▷ *Apply B to C; Schur update $(k \times k) \cdot (k \times n)$*
    $\mathbf{R}[L, C] = 0$
    $\mathbf{V}[\cdot, C] \leftarrow \mathbf{V}[\cdot, C] - \mathbf{V}[\cdot, B] \cdot \mathbf{\Lambda}$ ▷ *Matching update in $\mathbf{V}$*
  Recurse on C

We present a fast version of exhaustive reduction in Algorithm 3. The general structure follows Algorithm 2. However, rather than apply the pivots when we find them, we apply them in batches via the Schur complement through a standard recursion. We first augment the boundary matrix with an identity matrix, which is of size at least n, for reasons we shall describe below. To simplify notation, we assume n is a power of 2. If it is not, we can set the size of the identity matrix to $m + n$ with $m < n$ such that $m + 2n = 2^x$. Therefore, our initial matrices are:

$$\mathbf{R} = \begin{bmatrix} \mathbf{D} & 0 \\ 0 & \mathbf{I}_{m+n} \end{bmatrix}, \qquad \mathbf{V} = \mathbf{I}_{m+2n}.$$

Throughout the algorithm, we implicitly perform two types of permutations by taking appropriate subsets of rows and columns. The first are column permutations. To ensure that the appropriate submatrices are invertible, we require that after processing k columns of $\mathbf{R}$, there are k pivots, or, equivalently, that the matrix corresponding to the processed columns is full (column) rank. Because cycles reduce to zero columns, the if-statement in Algorithm 3 permutes a zero column from $\mathbf{R}$ with a non-zero column from the identity matrix $\mathbf{I}_{m+n}$, and records the indices of such columns in list Z, to eventually undo the permutation.



(a) (b) (c) (d)

Figure 1 An example of the column permutation in the exhaustive algorithm. (a) Initially, we append an identity matrix to the boundary matrix. (b) Assume we have processed the first k columns. The blue entries represent non-zero entries, and the dark blue boxes represent the pivots. To apply these pivots to next k columns (shown in gray), the first k columns must contain k pivots. (c-d) If there is a a zero column (shown in orange), we transpose it with the first non-zero entry in the appended identity matrix, ensuring the processed k columns all have pivots.

We must also consider row permutations. To apply Theorem 7, we must find the inverse $\mathbf{R}^{-1}[L, B]$ in Algorithm 3, which is used in the Schur complement. We construct a sequence $L = \langle \mathbf{low}(\ell) : \ell \in B \rangle$ of the pivot rows in columns B. Denoting by $\bar{L}$ the sequence of rows outside of L, we denote a permutation that places rows L below $\bar{L}$ as $\mathbf{P}_L$:

$$(\mathbf{P}_L \cdot \mathbf{R}) = \begin{bmatrix} \mathbf{R}[\bar{L}, \cdot] \\ \mathbf{R}[L, \cdot] \end{bmatrix}. \tag{3}$$

See Figure 2(b-c). The permutation is explicitly given by,

$$P_L : \mathbf{low}(B(i)) \mapsto m + n - |B| + i \qquad \text{for } \; i = 1, \dots, |B|$$

where $B(i)$ is the i-th entry of the indices in B.

▶ **Lemma 8.** *After permuting the rows by $\mathbf{P}_L$, the bottom-left $k \times k$ matrix is lower triangular, where $k = |L|$.*

We delay the proof until after the description of the algorithm. We now follow a recursion: we split matrix $\mathbf{R}$ into the first half of the columns denoted by B and the second half denoted by C. We proceed to call the algorithm on the first half of the matrix. Once we reach a

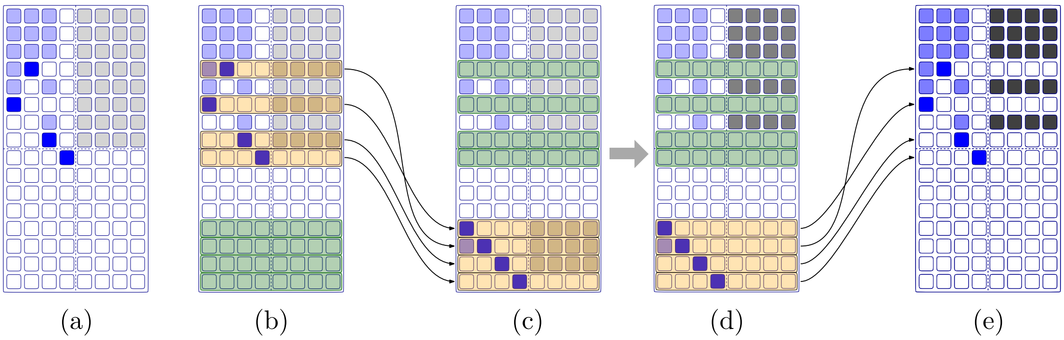


Figure 2 An example of the row permutations. (a) A submatrix consisting of 8 columns where the 4 columns on the left have been reduced, and we need to apply the pivots to the following 4 columns (in gray). (b-c) Each row which has a pivot gets permuted to the bottom of the matrix in the same order as the columns – resulting in a lower triangular matrix. (d) Applying the Schur complement zeros out the gray entries on the bottom right and we update the entries in the upper right (dark gray). (e) With the pivot rows zeroed out on the right, we reverse the permutation.

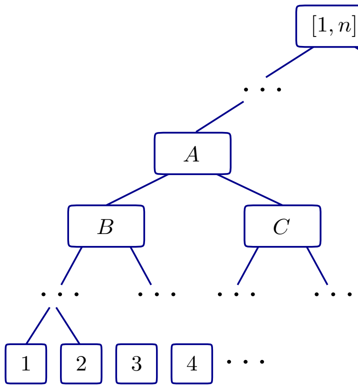


Figure 3 The standard binary recursion tree in how we split the matrix by column. Observe that $|B| = |C| = |L|$ in Algorithm 3 and that $\mathbf{R}[L, B] = (\mathbf{P}_L \cdot \mathbf{R})[\{n - |B| + 1, \ldots, n\}, B]$, the bottom $|B|$ rows in the permuted matrix.

leaf in the recursion tree (shown in Figure 3), the matrix consists of a single column. The key invariant is that whenever we reach a leaf, the corresponding column is reduced with respect to all the columns which come prior to it in the filtration. For the first column, this is tautologically true and we delay the proof for the general case until later.

If we are not in the base case (a single column), we must apply the pivots in the columns indexed by B (left half of the submatrix) to the the columns indexed by C (right half of the submatrix). As described in Equation (3), we construct the permutation $\mathbf{P}_L$ such that

$$(\mathbf{P}_L \cdot \mathbf{R})[\cdot, B \cup C]) = \begin{bmatrix} \mathbf{R}[\bar{L}, B] & \mathbf{R}[\bar{L}, C] \\ \mathbf{R}[L, B] & \mathbf{R}[L, C] \end{bmatrix}$$

By construction of $\mathbf{P}_L$, $\mathbf{R}[L, B]$ is lower-triangular with non-zero diagonal entries and so is full-rank. We can then apply the Schur complement to zero out all the corresponding rows in the columns C, $\mathbf{R}[L, C] \leftarrow 0$ and update the rest of the rows:

$$\underbrace{\mathbf{R}[\bar{L}, C]}_{\mathbf{C}_1} \leftarrow \underbrace{\mathbf{R}[\bar{L}, C]}_{\mathbf{C}_1} - \underbrace{\mathbf{R}[\bar{L}, B]}_{\mathbf{B}_1} \cdot \underbrace{\mathbf{R}[L, B]^{-1}}_{\mathbf{B}_2^{-1}} \cdot \underbrace{\mathbf{R}[L, C]}_{\mathbf{C}_2} = \mathbf{R}[\bar{L}, C] - \mathbf{R}[\bar{L}, B] \cdot \mathbf{\Lambda}$$

To update matrix $\mathbf{V}$, we perform the same operations $\mathbf{\Lambda} = \mathbf{R}[L, B]^{-1} \cdot \mathbf{R}[L, C]$ on $\mathbf{V}$:

$$\mathbf{V}[\bar{L}, C] \leftarrow \mathbf{V}[\bar{L}, C] - \mathbf{V}[\bar{L}, C] \cdot \mathbf{\Lambda}$$

Once the left half of the matrix has been applied to the right half, we recurse on the second half, i.e., C. Here again, because $\mathbf{R}[L, B]$ is lower-triangular full-rank and $\mathbf{R}[L, C] = 0$, the first column of $\mathbf{R}[\cdot, C]$ is reduced. We now prove correctness.

▶ **Lemma 9.** *When the recursion base case is reached for column k, $\mathbf{R}[\mathbf{low}(j), k] = 0$ for all $j < k$.*

Proof. For $k = 1$, this is tautological since $\mathbf{low}(j)$ is empty. For $k > 1$, we observe that entry $\mathbf{R}[\mathbf{low}(i), k]$ was made zero when $i \in B$ and $k \in C$. As the recursion always splits the column range in half, this condition must be satisfied somewhere in the recursion before the base case of k is reached. ◀

Proof of Lemma 8. We observe that when $\mathbf{P}_L$ is applied, the columns corresponding to L have a pivot. By construction, $\mathbf{P}_L$ places the pivots onto the diagonal in the bottom $|L|$ rows. For any of these rows above the diagonal, the entries are 0 by Lemma 9, implying the bottom $k \times k$ matrix is lower triangular. ◀

▶ **Theorem 10.** *Algorithm 3 computes the persistence diagram correctly.*

Proof. By Equation (1), it suffices to show that for all i, $\mathbf{low}(i)$ is computed correctly. For $i = 1$, this is trivially true. For $i > 1$, observe that at any step of the algorithm, we maintain

$$\mathbf{R} = \mathbf{D} \cdot \mathbf{V},$$

as any operations on $\mathbf{R}$ are applied to $\mathbf{V}$. Assuming the pivots are correct for $j < i$, by the above and Lemma 9, $\mathbf{V}$ gives a transformation depending only on columns $j < i$, such that all elements in $\mathbf{R}[\mathbf{low}(j), i] = 0$. If $\mathbf{low}(i)$ does not exist, $D[\cdot, i]$ was in the span of the previous columns and hence a cycle. If $\mathbf{low}(i)$ exists, it is distinct from $\{\mathbf{low}(j)\}_{j<i}$ and so we have a new pairing. The correctness of this pairing follows from the Pairing Uniqueness Lemma [8]. ◀

Running Time Analysis. The base case takes $O(1)$ time. For a general step where $|B| = |C| = k$, applying the update requires a row permutation of a $2n \times 2k$ matrix which takes $O(nk)$ time. We must compute $\mathbf{R}[L, B]^{-1}$ which is inverting a $k \times k$ matrix. Multiplying it with $\mathbf{R}[L, C]$ takes $O(k^\omega)$ time as both are $k \times k$ matrices. Finally, we multiply this product with the $n \times k$ matrix $\mathbf{R}[\bar{L}, C]$, which by Lemma 4, takes $O(nk^{\omega-1})$ time. Solving the recursion, we find that the total running time is $O(n^\omega)$ for $\omega > 2$ or $O(n^2 \log n)$ if $\omega = 2$. For completeness, we include the derivation in the full version [23].

5 Row Algorithm

While Algorithm 3 produces the exhaustive reduction in matrix multiplication time, a natural question is whether the same can be done for computing the lazy reduction (and its representatives). In this section, we present an algorithm which achieves this. We first give the iterative version before describing how it can be computed in $O(n^\omega)$. The idea behind this reduction is to consider rows from the bottom up. Observe that a pivot in the bottom row can be directly identified in $\mathbf{D}$ as the earliest (leftmost) non-zero entry, similarly as the

Algorithm 4 Incremental Row Algorithm.

$\mathbf{R} = \mathbf{D}$
$\mathbf{V} = \mathbf{I}_n, \mathbf{U} = \mathbf{I}_n$
for $i = n$ to 1 **do**
  if $\mathbf{lft}(i)$ is defined **then**
    $j \leftarrow \mathbf{lft}(i)$
    for $j' > j$ and $\mathbf{low}(j') = i$ **do**
      $\alpha \leftarrow \mathbf{R}[i, j']/\mathbf{R}[i, j]$
      $\mathbf{R}[\cdot, j'] \leftarrow \mathbf{R}[\cdot, j'] - \alpha \cdot \mathbf{R}[\cdot, j]$
      $\mathbf{V}[\cdot, j'] \leftarrow \mathbf{V}[\cdot, j'] - \alpha \cdot \mathbf{V}[\cdot, j]$
      $\mathbf{U}[j, \cdot] \leftarrow \mathbf{U}[j, \cdot] + \alpha \cdot \mathbf{U}[j', \cdot]$ (equivalently, $\mathbf{U}[j, j'] \leftarrow \alpha$)

pivot in the first column can be directly identified. To formalize the notion of the earliest eligible pivot in a given row, we define

$$\mathbf{lft}(i) = \arg\min_{j} \{\mathbf{R}[i, j] \neq 0 : \mathbf{low}(j) = i\}$$

As in the case of **low**, we only apply this function to $\mathbf{R}$. The function **lft** returns the earliest column j which is non-zero and is the lowest such non-zero entry in its column. This condition is important as the earliest (left-most) non-zero entry may have pivots below it. This column is used to zero out the i-th row in columns to the right provided they do not already contain a pivot, i.e., if $\mathbf{low}(j) = i$, then the column is applied. This condition is why we obtain equivalence with the lazy reduction – we do not apply the pivot column to later columns which already contain a pivot.

▶ **Lemma 11.** *Algorithm 4 produces the same decomposition as the lazy reduction.*

Proof. This is equivalent to [10, Theorem 3.1]. ◀

5.1 Fast Row Algorithm

The fast version of Algorithm 4 is given in Algorithm 5. As above, it reduces the matrix row by row from the bottom up. This variant does not require matrix inversion and requires less padding. The trade-off is that it requires a more complex sequence of updates. Assuming $\mathbf{D}$ is an $n \times n$ matrix, we set $m < n$ such that $m + n = 2^x$. We initialize our matrices as:

$$\mathbf{R} = \left[\begin{array}{c|c} \mathbf{D} & \mathbf{I}_{m+n} \\ \mathbf{0}_m & \end{array}\right], \qquad \mathbf{\Lambda} = \mathbf{I}_{m+n}.$$

To simplify notation, from this point on, we assume n is a power of 2 (as $m + n < 2n$), so the resulting matrix is $\begin{bmatrix}\mathbf{D} & \mathbf{I}_n\end{bmatrix}$. This ensures that $\mathbf{lft}(i)$ is defined for all i in $\mathbf{R}$, i.e., the identity matrix ensures there are no zero rows.

It will be convenient to again consider permutations, but here they will be column permutations. We build up the permutation incrementally. In the i-th step, we update the permutation with the column transposition

$$\pi(i) : \mathbf{lft}(n - i + 1) \leftrightarrow i.$$

Algorithm 5 Row Algorithm($A = [i, j]$).

if $i = j$ **then** ▷ *leaf*
  $k \leftarrow \mathbf{lft}(n - i + 1)$ ▷ *Identify the pivot in row* $(n - i + 1)$
  Update permutation matrix $\mathbf{P}$ with transposition $i \leftrightarrow k$.
  $(\mathbf{P}\cdot\mathbf{\Lambda}\cdot\mathbf{P})[i, j] \leftarrow -\frac{(\mathbf{R}\cdot\mathbf{P})[n-i+1,j]}{(\mathbf{R}\cdot\mathbf{P})[n-i+1,i]}$ for $j > i$.
  $(\mathbf{R}\cdot\mathbf{P})[n - i + 1, \{i + 1, \ldots, n\}] \leftarrow 0$ ▷ *Zero out* $(n - i + 1)$*-st row; see Remark 12:*
else
  $B \leftarrow [i, (i + j)/2)$ ▷ *left child*
  $C \leftarrow [(i + j)/2, j]$ ▷ *right child*
  $C_r \leftarrow n - C + 1$ ▷ *Rows are indexed top-down but the algorithm goes bottom up*
  Recurse on B
  Apply updates from rows B to C: ▷ *full row update;* $(k \times k) \cdot (k \times n)$
    $(\mathbf{R}\cdot\mathbf{P})[C_r, \cdot] \leftarrow (\mathbf{R}\cdot\mathbf{P})[C_r, \cdot] + (\mathbf{R}\cdot\mathbf{P})[C_r, B])(\mathbf{P}\cdot\mathbf{\Lambda}\cdot\mathbf{P})[B, \cdot]$
  Recurse on C
  $\bar{A}_r = [1, i]$ ▷ *everything above* A
  Full column update:
    $(\mathbf{R}\cdot\mathbf{P})[\bar{A}_r, C] \leftarrow (\mathbf{R}\cdot\mathbf{P})[\bar{A}_r, C] + (\mathbf{R}\cdot\mathbf{P})[\bar{A}_r, B] \cdot (\mathbf{P}\cdot\mathbf{\Lambda}\cdot\mathbf{P})[B, C]$ ▷ $(n \times k) \cdot (k \times k)$

which we use a shorthand for the two mappings $\mathbf{lft}(n - i + 1) \mapsto i$ and $i \mapsto \mathbf{lft}(n - i + 1)$. As $\mathbf{lft}$ is always defined, two columns are permuted in each step. The permutation after i steps is then given by

$$\mathbf{P}_i = \prod_{j \le i} \pi(j).$$

Note that we apply the permutations in order, i.e., for $i = 1, 2, \ldots$ While the columns corresponding to $\mathbf{lft}(n - i + 1)$ are fixed at i for all permutations, the original column at i may be permuted several times. In the algorithm, we omit the subscript; $\mathbf{P}$ refers to the accumulated permutation. As in the case of the column algorithm, we perform the permutations implicitly. We observe that in the permuted order, we arrange pivots on the left along the anti-diagonal, see Figure 4.

We keep track of the operations directly in the matrix $\mathbf{\Lambda}$ which is initialized as the identity matrix. Before delving into the details of the algorithm, assume that the all rows below the i-th row have been reduced. The updates in the i-th row are given by

$$(\mathbf{P}\cdot\mathbf{\Lambda}\cdot\mathbf{P})\,[i, j] = \begin{cases} -\frac{(\mathbf{R}\cdot\mathbf{P})[n-i+1,j]}{(\mathbf{R}\cdot\mathbf{P})[n-i+1,i]} & j > i \\ 1 & n - i + 1 = j \\ 0 & \text{else} \end{cases}$$

We digress here to explain why this works. As mentioned above, if we consider the column-permuted matrix $\mathbf{R}\cdot\mathbf{P}$, it has the pivots in the bottom-left along the anti-diagonal (Figure 4). The i-th column corresponds to the column given by $\mathbf{lft}(n - i + 1)$ (in the original filtration order). Hence, it is the earliest column with a non-zero entry in the i-th row from the bottom such that there is no pivot below. For any non-zero entry to the right of i, it must occur to the right of i in the original filtration order (before permuting), or it would have been returned by $\mathbf{lft}(n - i + 1)$ rather than the current i-th column in the permuted order.

▶ Remark 12. Hence in Algorithm 5, all non-zeros in the row to the right of i are pivots (lowest entries), because every column with a pivot below $n - i + 1$ has been moved to the left of i in the permuted order and so no reduction will be applied, i.e., coefficient will be 0.

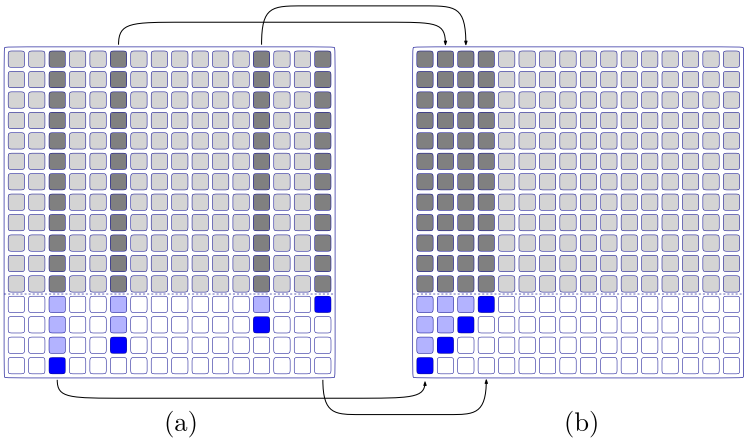


Figure 4 The permutation of the columns so that the pivots in the bottom k rows are on the right. An important observation is that despite this reshuffling of columns, because of the definition of $\mathbf{lft}(\cdot)$, we never "accidentally" reduce columns which occur earlier in the filtration order.

Finally, as we permute the columns we must also permute the rows of the $\mathbf{\Lambda}$ matrix so that the recorded column operations match. This represents the base case of the recursion. As $\mathbf{\Lambda}$ is initialized as the identity matrix, the case $n - i + 1 = j$ is taken care of implicitly.

Returning to the algorithm, we proceed by recursing as in the column algorithm, but on the rows rather than on the columns. We divide the rows of the sub-matrix into the top and bottom halves. We recurse first on the bottom half. Because the pivots are determined by rows below the current one, we can directly identify the pivot with **lft**.

The base case is a single row. By assumption, rows below the current row have been updated. At row $n - i + 1$, we first find the left-most non-zero entry and perform a column permutation setting the column $\mathbf{lft}(n - i + 1)$ to the i-th column, making $\mathbf{R}[n - i, i]$ a pivot as required. By the reasoning above, we may zero out the row, recording the operations in $\mathbf{\Lambda}$ and we can update $(\mathbf{R} \cdot \mathbf{P})[n - i, \{i + 1, \dots, n\}] \leftarrow 0$.

Though the recursion proceeds as in Algorithm 3, splitting the matrix into the indices in the "first half" and "second half," a complication is that the columns are processed left to right (same as the indexing), while the rows are processed bottom-up while being indexed top-down. Hence, we introduce:

$$C_r = n - C + 1,$$

by which we mean for any $i \in C$, $i \mapsto n - i + 1$. This indexes the rows from the bottom of the matrix as required by the algorithm. Returning to the recursion, once we processed the columns for B, we first apply the updates to the rows C_r,

$$(\mathbf{R} \cdot \mathbf{P})[C_r, \cdot] \leftarrow (\mathbf{R} \cdot \mathbf{P})[C_r, \cdot] + (\mathbf{R} \cdot \mathbf{P})[C_r, B] \cdot (\mathbf{P} \cdot \mathbf{\Lambda} \cdot \mathbf{P})[B, \cdot],$$

shown in orange in Figure 5. Note that though B_r is shown in Figure 5, it is not used explicitly in the algorithm as all the operations in those rows have already been performed. On the other hand, B in $(\mathbf{P} \cdot \mathbf{\Lambda} \cdot \mathbf{P})[B, \cdot]$ and in $(\mathbf{R} \cdot \mathbf{P})[C_r, B]$ do not need to be reversed, as they both represent column operations which are in the standard order. After this, the first row in C_r is now up-to-date with all operations from rows below it. This allows us to correctly identify **lft** for this row and perform the column transposition accordingly. Hence, when we recurse on C, the base case can be carried out.

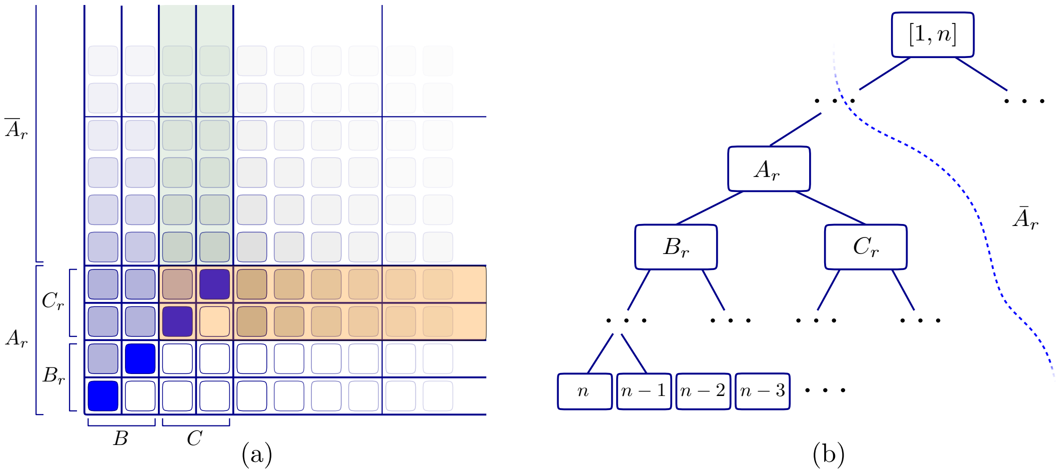


Figure 5 The updating steps in Algorithm 5. On the right we have the recursion tree for the rows, as opposed to the columns which is the same as in Figure 3 – as rows proceed bottom-up and columns proceed left to right. When we return from the recursion on B (or B_r), we have zeroed out all the entries to the right of columns B in rows B_r. We then first need to apply these to the rows in C_r (line 11 in Algorithm 5), shown in orange. We can then identify the correct pivot columns for rows C_r (after recursing), and complete the step by applying the operations from columns B to C above. This is shown in green – and corresponding to the right of $A_r = B_r \cup C_r$ in the recursion tree (the rows above A_r – denoted $\bar{A}_r$) to the right of the dashed line. This makes columns $B \cup C$ and rows A_r completely processed/reduced.

Returning from C, we complete the recursion by performing the update operations on the remaining parts of the columns in C, shown in green in Figure 5. Note, the columns are indexed by C not C_r. Let $\bar{A}_r = \{1, \ldots, \min(C_r) - 1\}$. The column update is given as

$$(\mathbf{R} \cdot \mathbf{P})[\bar{A}_r, C] \leftarrow (\mathbf{R} \cdot \mathbf{P})[\bar{A}_r, C] + (\mathbf{R} \cdot \mathbf{P})[\bar{A}_r, B] \cdot (\mathbf{P} \cdot \mathbf{\Lambda} \cdot \mathbf{P})[B, C]$$

Now all columns prior to and including C and all rows below and including C_r are up to date. We show the update sequence in Figure 5 (omitting the permutations). Note that in the column update, the rows in $\bar{A}_r$ are correctly indexed, so no reversal is necessary.

▶ **Theorem 13.** *Algorithm 5 computes the lazy reduction.*

Proof. The algorithm is equivalent to Algorithm 4, but it performs the operations by batches. Observe that in the base case for row $n - i + 1$, the column operations recorded in $\mathbf{\Lambda}$ are equal to α in the i-th step of Algorithm 4. If we performed the operations on the entire column, we would, in fact, obtain exactly Algorithm 4 (since there would be no need for any further updates in the recursion).

Hence, to show correctness, we need to ensure that we can correctly compute $\mathbf{lft}(i)$ in each base case. This is trivially true in the first step. In further steps, we need to ensure that all the column operations have been applied to $(\mathbf{R} \cdot \mathbf{P})[C, \cdot]$. We can directly verify that any column operations coming from row $j < i$, fall in exactly one B (with i in C) in the recursion and hence are applied before the base case for i. It only remains to show that the columns $(\mathbf{R} \cdot \mathbf{P})[\cdot, C]$ are up to date before the columns operations from rows in C_r are applied in the appropriate rows of $(\mathbf{R} \cdot \mathbf{P})[\bar{A}_r, \cdot]$, which is done in line 14. ◀

▶ **Theorem 14.** *Matrix* $\mathbf{U} = 2 \cdot \mathbf{I} - \mathbf{\Lambda}$ *satisfies* $\mathbf{D} = \mathbf{R} \cdot \mathbf{U}$*, or equivalently,* $\mathbf{V} = \mathbf{U}^{-1}$*.*

Proof. During a regular reduction, Algorithm 4, updates in matrix $\mathbf{U}$ use row operations to "undo" the column operations in matrices $\mathbf{R}$ and $\mathbf{V}$. When a column j in $\mathbf{R}$ is subtracted from column j', which itself has not been used to reduce any other column, its corresponding row $\mathbf{U}[j', \cdot]$ contains a single diagonal element – compare the situations in Algorithms 1 and 2. In this case, the update in matrix $\mathbf{U}$ is equivalent to setting entry $\mathbf{U}[j', j]$ to the coefficient that is the negation of the coefficient used for the column update. When a column j in $\mathbf{R}$ is subtracted from column j', which itself has not been used to reduce any other column, its corresponding row $\mathbf{U}[j', \cdot]$ contains a single diagonal element.

Algorithm 5 satisfies this property (which is why it produces a lazy reduction): when an update is recorded in matrix $\mathbf{\Lambda}$, the column that is being updated has not yet been used to reduce any other column. Therefore, off the diagonal, the corresponding matrix $\mathbf{U}$ is the negation of matrix $\mathbf{\Lambda}$ – compare the coefficients in Algorithm 5 with α in Algorithm 4. On the diagonal, both $\mathbf{\Lambda}$ and $\mathbf{U}$ are all ones. So adding $2 \cdot \mathbf{I}$ to $-\mathbf{\Lambda}$ recovers $\mathbf{U}$. ◀

▶ Remark 15. As $\mathbf{U}$ is an upper triangular matrix, by Theorem 7, it can be inverted in $O(n^\omega)$-time as required.

Running Time Analysis. The base case takes $O(n)$ time as recording the permutation and zeroing out operations each take linear time. Hence the base case takes $O(n)$ time. For the general case, we need to first apply the column permutations. Since we are permuting at most an $n \times k$ (or $k \times n$) matrix, applying the permutation (and undoing it) takes $O(nk)$ time by Lemma 5. Next, applying the row updates from B_r to C_r, assuming $|B| = |C| = k$, requires multiplying $(\mathbf{P} \cdot \mathbf{\Lambda} \cdot \mathbf{P})[B, \cdot]$, a $k \times n$ matrix, with $(\mathbf{R} \cdot \mathbf{P})[C_r, B]$, a $k \times k$ matrix – taking $O(nk^{\omega-1})$ time by Lemma 4. To complete the update, the subtractions take $O(nk)$ time. To compute the column updates, we must multiply $(\mathbf{R} \cdot \mathbf{P})[\bar{A}_r, B]$, an $n \times k$ matrix with $(\mathbf{P} \cdot \mathbf{\Lambda} \cdot \mathbf{P})[B, C]$, a $k \times k$ matrix, which again takes $O(nk^{\omega-1})$ time. Putting these together, we find that the full step can be completed in $O(nk^{\omega-1})$ time and as in the column algorithm case, solving the recursion gives an $O(n^\omega)$ running time for $\omega > 2$ and a $O(n^2 \log n)$ running time for $\omega = 2$.

6 Discussion

In this paper, we have shown that in the case of persistent (co)homology, the standard bases returned by the exhaustive and lazy reductions can be computed in matrix multiplication time. It is worth noting that while both the columns and row algorithms utilize batch updates, they are qualitatively different: (1) Only the column algorithm critically relies on fast matrix inversion; (2) The row algorithm update sequence is substantially more complex but requires only column permutations. While one can use the column algorithm to compute the lazy basis and the row algorithm for the exhaustive basis in the same asymptotic running time, the resulting algorithms are both complex and not particularly illuminating. There are also numerous technical complications to deal with, so we omit them from this paper. Another outstanding issue is whether it is possible to transform the exhaustive representatives to lazy representatives and vice versa without redoing the reduction. Again, we believe this possible, but omit the details due to space and because the resulting asymptotic running time is the same. Finally, we conclude with an open question: what is the optimal running time for computing representatives in zigzag persistence. Recently, this has been partially addressed in [12, 14] but optimality in full generality remains open.

References

1 Ulrich Bauer. Ripser: efficient computation of vietoris–rips persistence barcodes. *Journal of Applied and Computational Topology*, 5(3):391–423, 2021. doi:10.1007/S41468-021-00071-5.

2 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological Methods in Data Analysis and Visualization III: Theory, Algorithms, and Applications*, pages 103–117. Springer, 2014. doi:10.1007/978-3-319-04099-8_7.

3 Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat–persistent homology algorithms toolbox. *Journal of symbolic computation*, 78:76–90, 2017. doi:10.1016/J.JSC.2016.03.008.

4 Oleksiy Busaryev, Sergio Cabello, Chao Chen, Tamal K Dey, and Yusu Wang. Annotating simplices with a homology basis and its applications. In *Scandinavian workshop on algorithm theory*, pages 189–200. Springer, 2012. doi:10.1007/978-3-642-31155-0_17.

5 Gunnar Carlsson and Vin de Silva. Zigzag persistence. *Foundations of computational mathematics*, 10(4):367–405, August 2010. doi:10.1007/s10208-010-9066-0.

6 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*, SCG '09, pages 247–256, New York, NY, USA, 2009. ACM. doi:10.1145/1542362.1542408.

7 Chao Chen and Michael Kerber. An output-sensitive algorithm for persistent homology. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 207–216, 2011. doi:10.1145/1998196.1998228.

8 David Cohen-Steiner, Herbert Edelsbrunner, and Dmitriy Morozov. Vines and vineyards by updating persistence in linear time. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 119–126, 2006. doi:10.1145/1137856.1137877.

9 David Cohen-Steiner, André Lieutier, and Julien Vuillamy. Lexicographic optimal homologous chains and applications to point cloud triangulations. *Discrete & computational geometry*, 68(4):1155–1174, December 2022. doi:10.1007/s00454-022-00432-6.

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

11 Vin de Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Persistent cohomology and circular coordinates. *Discrete & computational geometry*, 45(4):737–759, June 2011. doi:10.1007/s00454-011-9344-x.

12 Tamal K Dey and Tao Hou. Updating barcodes and representatives for zigzag persistence. *arXiv preprint arXiv:2112.02352*, 2021.

13 Tamal K Dey and Tao Hou. Fast computation of zigzag persistence. In *30th Annual European Symposium on Algorithms (ESA 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

14 Tamal K Dey, Tao Hou, and Dmitriy Morozov. A fast algorithm for computing zigzag representatives. *arXiv preprint arXiv:2410.20565*, 2024. doi:10.48550/arXiv.2410.20565.

15 H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & computational geometry*, 28:511–533, 2002. doi:10.1007/S00454-002-2885-2.

16 Herbert Edelsbrunner and John L Harer. *Computational topology: an introduction*. American Mathematical Society, 2022.

17 G. Henselman and R. Ghrist. Matroid Filtrations and Computational Persistent Homology. *ArXiv e-prints*, June 2016. arXiv:1606.00199.

18 Alan Hylton, Greg Henselman-Petrusek, Janche Sang, and Robert Short. Performance enhancement of a computational persistent homology package. In *2017 IEEE 36th international performance computing and communications conference (IPCCC)*, pages 1–8. IEEE, 2017. doi:10.1109/PCCC.2017.8280468.

19 Gil Kalai. Enumeration of q-acyclic simplicial complexes. *Israel Journal of Mathematics*, 45:337–351, 1983.

20 Michael Kerber, Donald R Sheehy, and Primoz Skraba. Persistent homology and nested dissection. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1234–1245. SIAM, 2016. doi:10.1137/1.9781611974331.CH86.

21 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. doi:10.1145/1998196.1998229.

22 Dmitriy Morozov. Persistence algorithm takes cubic time in worst case. *BioGeometry News, Dept. Comput. Sci., Duke Univ*, 2, 2005.

23 Dmitriy Morozov and Primoz Skraba. Persistent (co)homology in matrix multiplication time, 2024. doi:10.48550/arXiv.2412.02591.

24 Arnur Nigmetov and Dmitriy Morozov. Topological optimization with big steps. *Discrete & computational geometry*, 72(1):310–344, July 2024. doi:10.1007/s00454-023-00613-x.

25 Primoz Skraba, Gugan Thoppe, and D Yogeshwaran. Randomly weighted d-complexes: Minimal spanning acycles and persistence diagrams. *The Electronic Journal of Combinatorics*, pages P2–11, 2020.

26 Hubert Wagner, Chao Chen, and Erald Vuçini. Efficient computation of persistent homology for cubical data. In *Topological methods in data analysis and visualization II: theory, algorithms, and applications*, pages 91–106. Springer, 2011.

27 Afra Zomorodian and Gunnar Carlsson. Computing persistent homology. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 347–356, 2004. doi:10.1145/997817.997870.