# Efficient Computation of Persistent Homology for Cubical Data

Hubert Wagner, Chao Chen, Erald Vuçini

**Abstract** In this paper we present an efficient framework for computation of persistent homology of cubical data in arbitrary dimensions. An existing algorithm using simplicial complexes is adapted to the setting of cubical complexes. The proposed approach enables efficient application of persistent homology in domains where the data is naturally given in a cubical form. By avoiding triangulation of the data, we significantly reduce the size of the complex. We also present a data-structure designed to compactly store and quickly manipulate cubical complexes. By means of numerical experiments, we show high speed and memory efficiency of our approach. We compare our framework to other available implementations, showing its superiority. Finally, we report performance on selected 3D and 4D data-sets.

## 1 Introduction

Persistent homology [10, 11] has drawn much attention in visualization and data analysis, mainly due to the fact that it extracts topological information that is resilient to noise. This is especially important in application areas, where data typically comes from measurements which are inherently inexact. Although direct ap-

Hubert Wagner
Vienna University of Technology, Austria and
Jagiellonian University, Poland
e-mail: hubert.wagner@ii.uj.edu.pl

Chao Chen
Institute of Science and Technology Austria and
Vienna University of Technology, Austria
e-mail: chao.chen@ist.ac.at

Erald Vuçini
VRVis Center for Virtual Reality and Visualization Research-Ltd, Austria and
Vienna University of Technology, Austria
e-mail: erald.vucini@vrvis.at

plication of persistent homology is still at an early stage, closely related concepts like size functions [4], contour trees [2, 5], Reeb graphs [24] and Morse-Smale complexes [14] have been successfully used.

The under-usage of persistence in applications is largely due to its high computational cost. The standard algorithm [10] takes cubic running time, which can be prohibitive even for small size data (e.g., $64 \times 64 \times 64$). In addition to the high time complexity, there are two further issues: **(1)** the memory consumption of the currently available implementations, even for small data sizes, is very large and hence prohibitive for commodity computers, and **(2)** the focus of several applications is in data of higher dimensions, e.g., 4D, 5D or higher. Few implementations for general dimension are available and the existing ones do not scale well with the increase of dimensions, hence introducing larger computational times and memory inefficiency.

In this paper, we present an efficient framework that computes persistent homology exactly[1]. To our knowledge, this is the very first implementation that could handle large size and high dimensional data in reasonable time and memory. We focus on uniformly/regularly sampled data which is common in visualization and data analysis, i.e. image data consisting of pixels (2D images), voxels (3D scans, simulations), or their higher-dimensional analogs, e.g., 4D time-varying data. In this work, we use the name 'cubical' for such data.

We depart from the standard method which involves triangulating the space, and computing persistent homology of the resulting simplicial complex [10, 11]. We use cubical complexes [15], which do not require subdivision of the input. The advantage is twofold. First, the size of the complexes is significantly reduced, especially for high dimensional data (see Section 5 for a quantitative analysis). Second, cubical complexes allow the usage of more compact data-structures.

The standard persistence algorithm requires the computation of a sorted boundary matrix. This step can be a significant bottleneck, especially in terms of memory consumption. In this work we provide an efficient and compact algorithm for this step, using techniques from (non-persistence) cubical homology [15] (see Section 4).

Finally, in Section 7, we present experimental results. Comparison with existing packages shows significant efficiency improvement. We also explore how our method scales with respect to data size and dimension. In conclusion, our framework can handle data of large size and high dimension, and therefore, makes the persistence computation of cubical data more feasible.

## 2 Related Work

The first algorithm for computing persistence [11] has cubic running time with regard to the complex size (which is larger than the input size). Morozov [20] formulated a worst case scenario for which the persistence algorithm reaches this asymp-

[1] We emphasize that our work focuses on computing persistence exactly. There are approximation methods which trade accuracy for efficiency. See Section 2.

totic bound. When focusing on 0-dimensional homology, union-find data structures can be used to compute persistence in time $O(n\alpha(n))$ [10], where $\alpha$ is the inverse of the Ackermann functions and $n$ is the input size. Milosavljevic et al. [18] compute persistent homology in matrix multiplication time $O(n^{\omega})$ where the currently best estimation of $\omega$ is 2.376. Chen and Kerber [6] proposed a randomized algorithm whose complexity depends on the number of persistence pairs whose persistence is over a certain threshold. Despite showing better theoretical complexity, it is unclear whether these methods are better than the standard persistence algorithm in practice.

In terms of implementation, Morozov [19] provides a C++ code for the persistence algorithm. Chen and Kerber [7] devised a technique which, in practice, significantly improves the matrix-reduction part of this algorithm. We build upon their work, to improve the overall performance of the persistence algorithm.

The application of cubical homology is straightforward in the areas of image processing and visualization, where cubical data is the typical input. Non-persistent cubical homology has found practical applications in a number of cases [21, 22]. A few attempts of cubical persistence computations have been made recently [16, 25]. They do not, however, tackle the problem of performance. In [25], experiments with datasets containing several thousands of voxels are reported. In comparison, real world applications require processing of data in the range of millions or billions of voxels.

Recently, Mrozek and Wanner [21] showed that cubical persistent homology can be used for medium-sized datasets. A detailed performance summary is given for 2D and 3D images. One downside of this approach is the dependency on the number of unique values of the image. When such number is close to the input size, the complexity is prohibitively high. In Section 7, we compare our method with this algorithm.

We must differentiate between two main types of persistence computations: exact and approximative (where the persistence is calculated approximately). While we focus on the first type, approximation is less computationally intensive, and thus is important for large data. Bendich et al. [3] use octrees to approximate the input. A simplicial complex of small size is then used to complete persistence computation.

## 3 Theoretical Background

***Simplicial and cubical complexes.*** In computational topology, simplicial complexes are frequently used to describe topological spaces. A simplicial complex consists of simplices like vertices, edges and triangles. In general, a *$d$-simplex* is the convex hull of $d+1$ points. The convex hull of any subset of these $d+1$ points is a *face* of this $d$-simplex. A collection of simplices, $K$, is a *simplicial complex* if: 1) for any simplex in $K$, all its faces also belong to $K$, and 2) for any two simplices in $K$, their intersection is either empty, or a common face of them.

Next, we define cubical complexes. An *elementary interval* is defined as a unit interval $[k, k+1]$, or a degenerate interval $[k, k]$. For a $d$-dimensional space, a *cube*

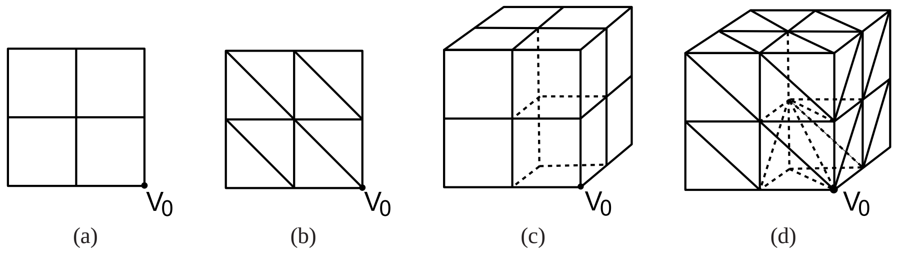


**Fig. 1** Cubical complex triangulations: a) a 2D cubical complex, and b) its triangulation, c) a 3D cubical complex, and d) its triangulation (only simplices which contain $V_0$ are drawn).

is a product of $d$ elementary intervals $I$: $\prod_{i=1}^{d} I_i$. The number of non-degenerate intervals in such product is the *dimension* of this cube. 0-cubes, 1-cubes, 2-cubes and 3-cubes are vertices, edges, squares and 3D cubes (voxels) respectively. Given two cubes: $a, b \subseteq R^d$, $a$ is a *face* of $b$ if and only if $a \subseteq b$. A *cubical complex* of dimension $d$ is a collection of cubes of dimension at most $d$. Similarly to the definition of a simplicial complex, it must be closed under taking faces and intersections.

In this paper, we will use cubical complexes to describe the data. In Figure 1 we show 2-dimensional and 3-dimensional cubical complexes, describing a 2D image of size $3 \times 3$ and a 3D image of size $3 \times 3 \times 3$. The corresponding simplicial complex representations are also shown. We use one specific triangulation, namely, the *Freudenthal triangulation* [13, 17]. Such triangulation is easy to extend to general dimension.

***Boundary matrix.*** For any $d$-dimensional *cell* (that is: simplex or cube), its *boundary* is the set of its $(d-1)$-dimensional faces. This extends linearly to the boundary of a set of $d$-cells, namely, a *$d$-chain*. Specifically, the boundary of a set of cells is the modulo 2 sum of the boundaries of each of its elements. In general, if we specify a unique index for each simplex, a $d$-chain corresponds to a vector in $\mathbb{Z}_2^{n_d}$, where $n_d$ is the number of $d$-dimensional cells in the complex. Furthermore, the $d$-dimensional boundary operator can be written as a $n_{d-1} \times n_d$ binary matrix whose columns are the boundaries of $d$-cells, while rows represent $(d-1)$-cells.

***Persistent homology.*** We review persistent homology [10, 11], focusing on $\mathbb{Z}_2$ homology. Due to space limitation, we do not introduce homology in this paper. Please see [12] for an intuitive explanation, and [23, 10] for related textbooks.

Given a topological space $\mathbb{X}$ and a *filtering function* $f : \mathbb{X} \to \mathbb{R}$, *persistent homology* studies homological changes of the sublevel sets, $\mathbb{X}^t = f^{-1}(-\infty, t]$. The algorithm captures the birth and death times of homology classes of the sublevel set as it grows from $\mathbb{X}^{-\infty}$ to $\mathbb{X}^{+\infty}$, e.g., components as 0-dimensional homology classes, tunnels as 1-dimensional classes, voids as 2-dimensional classes, and so on. By birth, we mean that a homology class comes into being; by death, we mean it either becomes trivial or becomes identical to some other class born earlier. The persistence, or lifetime of a class, is the difference between the death and birth times. Homology classes with larger persistence reveal information about the global structure of the space $\mathbb{X}$, as described by the function $f$.

Persistence could be visualized in different ways. One well-accepted idea is the persistence diagram [8], which is a set of points in a two-dimensional plane, each corresponding to a persistent homology class. The coordinates of such a point are the birth and death time of the related class.

An important justification of the usage of persistence is the stability theorem. Cohen-Steiner et al. [8] proved that for any two filtering functions $f$ and $g$, the difference of their persistence is always upperbounded by the $L^\infty$ norm of their difference:

$$\|f-g\|_\infty := \max_{x\in\mathbb{X}} |f(x)-g(x)|.$$

This guarantees that persistence can be used as a signature. Whenever two persistence outputs are different, we know that the functions are definitely different.

In our framework, for 2D images we assume 4-connectivity. In general, for $d$-dimensional cubical data, we use $2d$-connectivity.

***Persistence computation.*** Edelsbrunner et al. [11] devised an algorithm to compute persistent homology, which works in cubic time (in the size of a complex). It requires preprocessing of the data (also see Figure 2). In case of images, function $f$ is defined on all pixels/voxels. First, these values are interpreted as values of vertices of a complex. Next, the *filtration* of the complex is computed and the *sorted boundary matrix* is generated. This matrix is the input to the *reduction algorithm*.

*Filtration* can be described as adding cells with increasing values to a complex, one by one. To achieve this, a *filtration-building algorithm* extends the function to all cells of the complex, by assigning each cell the maximum value of its vertices. Then, all cells are sorted in ascending order according to the function value, so that each cell is added to the filtration after all of its faces. Such a sequence of cells is called a *lower-star filtration*. Having calculated the ordering of cells, a sorted boundary matrix can be generated.

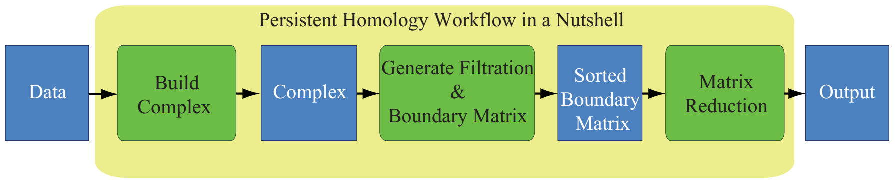

Persistent Homology Workflow in a Nutshell
Data
Build Complex
Complex
Generate Filtration & Boundary Matrix
Sorted Boundary Matrix
Matrix Reduction
Output

**Fig. 2** A workflow of the persistent homology computation.

In the reduction step, the algorithm performs column reductions on the sorted boundary matrix from left to right. Each new column is reduced by addition with the already reduced columns, until its lowest nonzero entry is as high as possible. The reduced matrix encodes all the persistent homology information.

## 4 Efficient Filtration-Building Algorithm

The filtration-building is one of the main bottlenecks of the persistence algorithm. A straightforward approach would choose to store the boundary relationship between cells and their faces. In this section, we describe the first major contribution of the paper, a new algorithm for the filtration-building step. Our algorithm uses the regular structure of cubical complex and adapts a compact data structure which has shown its power in non-persistent cubical homology.

***Cubical complex representation.*** We first describe CubeMap, a compact representation of cubical complexes. To the best of our knowledge, a similar structure was first introduced in CAPD library [1] for non-persistent cubical homology.

For an example 2D image with $5 \times 5$ pixels see Figure 3. Due to the regular structure, relationship between cells can be read immediately from their coordinates. We can store the necessary information (i.e. order in the filtration, function value) for each cell in a $9 \times 9$ array (Figure 3(c)). We can immediately get the dimension of any cell (whether it is a vertex, edge, or square), as well as its faces and *cofaces*, namely, cells of whom it is a face. We do this by checking coordinates modulo 2. To explain this fact, we recall that we defined cubes as products of intervals. Even coordinates correspond to degenerate intervals of a cube.

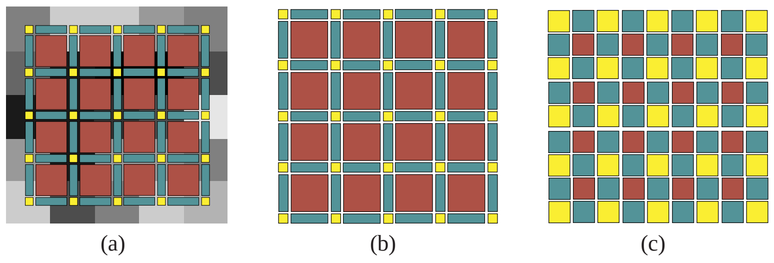


**Fig. 3** a) Cubical complex built over a gray-scale 2D image with $5 \times 5$ pixels. Each vertex (yellow) corresponds to a pixel. Edges (blue) and cubes (red) are constructed accordingly. b) The cubical complex itself. c) The corresponding CubeMap, all informations for filtration-building are encoded in a $9 \times 9$ array. Each element corresponds to a cell.

The aforementioned properties generalize for arbitrary dimensions. This is due to the inductive construction of cubical complexes, and is related to cubes being products of intervals.

Let us consider input data of dimension $d$ and size $w^d$, where $w$ is the number of vertices in each dimension. We store information attached to cells in a $d$-dimensional array with $(2w-1)^d$ elements. This array is composed of overlapping copies of arrays of size $3^d$. We call this structure the $CubeMap$.

The major advantage of the proposed data-structure is the improved memory efficiency. Boundary relations are implicitly encoded in the coordinates of cells. The coordinates itself are also implicit. Furthermore, we can randomly access each

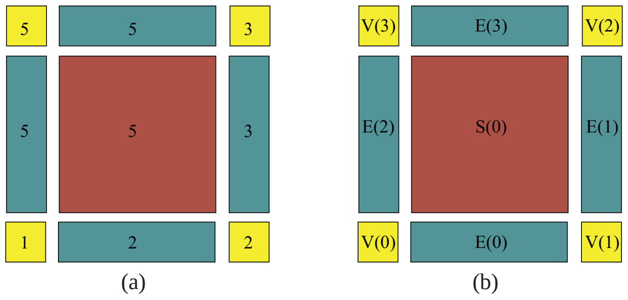


**Fig. 4** a) Values of $f$ assigned to vertices and extended to all cubes. b) Cells are assigned indices in the filtration. These indices are separate for each dimension. Vertices are marked as V, edges as E, squares as S.

cell and quickly locate its boundaries. See Section 6 for further details and Section 7 for an experimental justification.

***Filtration-building.*** Let us now present an efficient algorithm to compute a filtration of a cubical complex induced by a given function $f$ (see Algorithm 1). We use the CubeMap datastructure to store additional information for each cell (function value, filtration order). The outcome of this algorithm is a sorted boundary matrix, being the input of the reduction step. Since in case of cubical data boundary matrices have only $O(d)$ non-zero elements per column, sparse representations are typically used.

The intuition behind the algorithm is that when we iterate through all vertices in *descending* order, we know that the vertices' cofaces, which were not added to the filtration, belong to their lower-stars, and can be added to the filtration. We cannot build the boundary matrix in the same step, since the indices of the adjacent cells might be not yet computed. Do note that on line 5, filtration indices are assigned from higher to lower. Figure 4 illustrates the algorithm. Exploiting the properties of cubical complexes makes this algorithm efficient (refer to section 6 for details).

**Algorithm 1** Computing filtration and sorted boundary matrix

```
Input: function $f$, given on vertices of a cubical complex $K$
Output: sorted boundary matrix, extension of function $f$ to all cells of $K$
 1: sort vertices of $K$ by values of $f$ (descending)
 2: for each vertex $V_i$ in sorted order do
 3:     for each cube $C_j$ with $V_i$ as one of its vertices do
 4:         if $C_j$ was not assigned filtration index then
 5:             assign next (smaller) filtration index to $C_j$
 6:             $f(C_j) \leftarrow f(V_i)$.
 7: for each cube $C_i$ of $K$ do
 8:     column $\leftarrow$ filtration index of $C_i$
 9:     for each cube $B_j$ in boundary of $C_i$ do
10:         row $\leftarrow$ filtration index of $B_j$
11:         boundary matrix(row, column) $\leftarrow 1$
```

## 5 Sizes of Complexes

When switching from simplicial complexes to cubical complexes, the size of the complex is significantly reduced. This is a clear improvement in both memory and runtime efficiency. We should emphasize that the complexity of the standard reduction algorithm is given in the size of the complex, not the number of vertices. Therefore, reducing the size of a complex has a significant impact.

In this section, we analyze how the ratio of the sizes of simplicial and cubical complexes increases with regard to the data dimension. We show that this ratio increases exponentially with the dimension, which motivates the usage of cubical approaches, such as ours. For simplicity we disregard boundary effects, assuming that the number of cells lying on the boundary is insignificant[2].

In Figure 1, we show examples of cubical complexes and their triangulations. The ratio between the number of cofaces of the vertex $V_0$ in a simplicial and in a cubical complex is $(6:4)$ and $(26:8)$ for 2D and 3D complexes, respectively. This is also the ratio of the size of simplicial and cubical complexes, since these selected cells serve as their *generators*.

For a $d$-dimensional data, we denote the concerned ratio as $\rho_d = S_d/C_d$, where $C_d$ and $S_d$ are the sizes of a cubical complex and its triangulation, respectively. It is nontrivial to give an exact formula of $\rho_d$, since the minimal-cardinality cube-triangulation is an open problem [26]. Here we give a lower-bound of $\rho_d$ for $d \leq 7$ by triangulating all cubes of a cubical complex separately in each dimension. When triangulating a $d$-cube, we count only the resulting $d$-simplices, and their $(d-1)$-dimensional intersections. Finally, taking into account the fact that certain simplices will be common faces of multiple higher-dimensional simplices, we get

$$\rho_d \geq \frac{\sum_{i=0}^{d} \binom{d}{i} \tau_i + \sum_{i=0}^{d-1} \binom{d}{i+1} (\tau_{i+1} - 1)}{2^d}$$

where $\tau_d$ is the number of $d$-simplices in a triangulation of a $d$-cube.

In Table 1 we present the values of such lower-bounds for different dimensions $(d = 1, \cdots, 7)$. We consider two cases: optimal triangulation [26] and Freudenthal, using $d!$ simplices. It is clear that in both cases the lower-bound increases exponentially with regard to the data dimension.

This observation leads to the following conjecture. Such conjecture, if correct, shows how algorithms based on cubical and simplicial complexes scale with respect to the dimension.

*Conjecture 1.* $\rho_d$ increases exponentially in $d$.

[2] This assumption is realistic when complexes are large.

**Table 1** Lower-bounds of the size ratios $\rho_d$.

| | Dimension ($d$) | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Optimal | $\tau_d$ | 1 | 2 | 5 | 16 | 67 | 308 | 1493 |
| | lower-bound of $\rho_d$ | 1.0 | 1.5 | 2.75 | 5.625 | 12.937 | 33.968 | 90.265 |
| Freudenthal | $\tau_d$ | 1 | 2 | 6 | 24 | 120 | 720 | 5040 |
| | lower-bound of $\rho_d$ | 1.0 | 1.5 | 3.0 | 7.125 | 19.375 | 60.156 | 213.062 |

## 6 Implementation Details

In this section we briefly comment on the techniques we used to enhance the performance of our implementation. We focus on the choice of proper data-structures, and exploiting various features of cubical complexes. We implemented this algorithm in C++.

***Filtration-building algorithm.*** We use a 2-pass modification of the standard filtration-building algorithm. Reversing the iteration order over the vertices does not affect the asymptotic complexity, but simplifies the first pass of the algorithm, which resulted in better performance.

We calculate the time complexity of this algorithm. To do this precisely, we assume that the dimension $d$ is not a constant. This is a fair assumption since we consider general dimensions. We use a $d$-dimensional array to store our data, so random access is not $O(1)$, but $O(d)$, as it takes $d-1$ multiplications and additions to calculate the address in memory.

Let $n$ be the size of input (the number of vertices in our complex). In total there are $O(2^d n)$ cubes in the complex. We ignore what happens at boundaries of the complex. Each $d$-cube has exactly $2d$ boundary cubes, and each vertex has $3^d-1$ cofaces. Accessing each of them costs $O(d)$. This yields the following complexity of calculating the filtration and the boundary matrices: $O(d3^d n + d^2 2^d n)$.

Using the properties of CubeMap, we can reduce this complexity. Since the structure of the whole complex is regular, we can precalculate memory-offsets from cubes of different dimensions and orientations to its cofaces and boundaries. Accessing all boundary cubes and cofaces takes constant amortized time. The preprocessing time does not depend on input size and takes only $O(d^2 3^d)$ time and memory. With the CubeMap data structure, our algorithm can be implemented in $\Theta(3^d n + d2^d n)$ time and $\Theta(d2^d n)$ memory.

***Storing boundary matrices.*** Now we present a suggestion regarding performance, namely, the usage of a proper data-structure for storing the columns of (sparse) boundary matrices. In [10] a linked-list data-structure is suggested. This seems to be a sub-optimal solution, as it has an overhead of at least one pointer per stored element. For 64-bit machines this is 8B - twice as much as the data we need to store in a typical situation (one 32-bit integer).

Using an automatically-growing array, such as std::vector available in STL is much more efficient (speed-up by a factor of at least 2). Also the memory overhead is much smaller - 16B per column (not per element as before). All the required operations have the same (amortized) complexity [9], assuming that adding an element at the back can be done in constant amortized time. Also, iterating the array from left to right is fast, due to memory-locality, which is not the case for linked-list implementations.

## 7 Results

The testing platform of our experiments is a six-core AMD Opteron(tm) processor 2.4GHz with 512KB L2 cache per core, and 66GB of RAM, running Linux. Our algorithm runs on a single core. We use 3D and 4D (3D+time) cubical data for testing and comparing our algorithm. We compare our method with existing implementations. We measure memory usage, filtration-building and reduction times.

***Comparing with existing implementations.*** We compare our implementation (referred to as CubPers) to three existing implementations:

1. **SimpPers:** (by Chen and Kerber [7]) Uses simplicial complexes. Both SimpPers and CubPers use the same reduction algorithm, but our approach uses cubical complexes and CubeMap to accelerate the filtration-building process.
2. **Dionysus:** (by Morozov [19]) This code is suited for more general complexes and computes also other information like vineyards. We adapt this implementation to operate on cubical data, by triangulating the input, which is the standard approach. Since this implementation takes a filtration as input, the time for building the filtration is not taken into account.
3. **CAPD:** (by Mrozek [21], a part of CAPD library [1]) We stress that this approach was designed for data with a small number of unique function values, which is not the case for the data we use. Additionally it produces and stores persistent homology generators which incur a significant overhead.

In Tables 2 and 3 we compare the memory and times of our approach to the aforementioned implementations. For testing we have used the Aneurysm dataset [3]. In order to explore the behavior of the algorithms when the data size increases linearly, we uniformly scale the data into $50^3$, $100^3$, $150^3$, $200^3$, using nearest neighbor interpolation. Clearly, our implementation, CubPers, outperforms other programs in terms of memory and time efficiency.

Due to the usage of CubeMap, the memory usage is reduced by an order of magnitude. This is extremely important, as it enables the usage of much larger datasets on commodity computers. While SimpPers significantly improves over other methods in terms of reduction time [7], our method further improves the filtration-building time. It is also shown that using cubical complexes instead of simplicial complexes improves the reduction time.

***Scalability.*** Table 4 shows how our implementation scales with respect to dimension. We used random data - each vertex is assigned an integer value from 0 to 1023

[3] From the Volvis repository (http://volvis.org/).

**Table 2** Memory consumption for the computation of persistence of the Aneurysm dataset for different implementations. Several down-sampled version of the original dataset were used. For specific cases the results are not reported due to memory or time limitations.

| | $50 \times 50 \times 50$ | $100 \times 100 \times 100$ | $150 \times 150 \times 150$ | $200 \times 200 \times 200$ | $256 \times 256 \times 256$ |
|---|---|---|---|---|---|
| CAPD | 500MB | 2700MB | 16000MB | - | - |
| Dionysus | 200MB | 6127MB | 21927MB | 49259MB | - |
| SimpPers | 352MB | 3129MB | 11849MB | 25232MB | - |
| CubPers | 42MB | 282MB | 860MB | 2029MB | 4250MB |

**Table 3** Times (in minutes) for the computation of persistence of the Aneurysm dataset for different techniques. For SimpPers and CubPers, we report both filtration-building time and reduction time, the whole computation is the sum of the two times.

| | $50 \times 50 \times 50$ | $100 \times 100 \times 100$ | $150 \times 150 \times 150$ | $200 \times 200 \times 200$ | $256 \times 256 \times 256$ |
|---|---|---|---|---|---|
| CAPD | 0.26 | 12.3 | 134.55 | - | - |
| Dionysus | 0.32 | 3.03 | 13.74 | 47.23 | - |
| SimpPers | (0.05+0.02) | (0.43+0.16) | (1.63+0.9) | (3.53+3.33) | - |
| CubPers | (0.01+0.001) | (0.10+0.01) | (0.33+0.13) | (0.87+0.43) | (1.25+0.78) |

**Table 4** Times (in minutes) for the computation of persistence for one million vertices in different dimensions (1-6). Both times for filtration and persistence (filtration+reduction) are given.

| Dimension | 1D | 2D | 3D | 4D | 5D | 6D |
|---|---|---|---|---|---|---|
| Filtration | 0.017 | 0.05 | 0.15 | 0.55 | 1.65 | 3.70 |
| Persistence (reduction) | 0.067 | 0.12 | 0.23 | 0.87 | 4.80 | 17.70 |

(the choice was arbitrary). The distribution is uniform and the number of vertices (1,000,000) is constant for all dimensions. We can see that performance deteriorates exponentially. This is understandable, since the size of a cubical complex increases exponentially in dimension ($2^d$). The size of its boundary matrix increases even faster ($d2^d$).

In Table 5 we report the timings and memory consumptions for several 3D datasets[4] and a 4D time-varying data[5] consisting of 32 timesteps. We stress the following three observations:

- Due to the significant improvement of memory efficiency, our implementation could compute these data on commodity computers.
- Both memory and filtration-building times grow linearly in the size of data (number of voxels). This also reveals that for very large scale data ($\geq 1000^3$), the memory consumption would be too large. In such a case, we would need an approximation algorithm (as in [3]) or an out-of-core algorithm.

[4] From the Volvis repository (http://volvis.org/) and ICGA repository (www.cg.tuwien.ac.at)

[5] From the Osirix repository (http://pubimage.hcuge.ch:8080/)

- Reduction time varies for different data. Among all data-files we tested, two medium ($256^3$) cases (Christmas Tree and Christmas Present) took significantly more reduction time. This data-dependent behavior of the reduction algorithm is an open problem in persistent homology literature.

**Table 5** Times in minutes for different 3D datasets and a 4D time-varying data (32 timesteps). Times below 0.001 min were reported as 0.00.

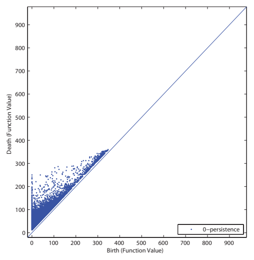
| Data set | Size | Memory (MB) | Times (min) |
|---|---|---|---|
| Silicium | $98 \times 34 \times 34$ | 30 | (0.02+0.07) |
| Fuel | $64 \times 64 \times 64$ | 82 | (0.02+0.00) |
| Marschner-Lobb | $64 \times 64 \times 64$ | 82 | (0.03+0.00) |
| Neghip | $64 \times 64 \times 64$ | 82 | (0.03+0.00) |
| Hydrogene | $128 \times 128 \times 128$ | 538 | (0.22+0.40) |
| Engine | $256 \times 256 \times 128$ | 2127 | (1.07+0.30) |
| Tooth | $256 \times 256 \times 161$ | 2674 | (1.43+1.48) |
| Christmas Present | $246 \times 246 \times 221$ | 3112 | (2.43+264.35) |
| Christmas Tree | $256 \times 249 \times 256$ | 3809 | (3.08+11.1) |
| Aneurysm | $256 \times 256 \times 256$ | 4250 | (1.75+0.77) |
| Bonsai | $256 \times 256 \times 256$ | 4250 | (1.98+0.93) |
| Foot | $256 \times 256 \times 256$ | 4250 | (2.15+0.70) |
| Supine | $512 \times 512 \times 426$ | 26133 | (23.06+11.88) |
| Prone | $512 \times 512 \times 463$ | 28406 | (25.96+10.38) |
| Vertebra | $512 \times 512 \times 512$ | 31415 | (26.8+7.58) |
| Heart (4D) | $256 \times 256 \times 14 \times 32$ | 13243 | (20.20+1.38) |

***Understanding time-varying data with persistence.*** With our efficient tool, we are enabled to study 4D time-varying data using persistence. This is one of our future research focuses. We conclude this section by a pilot study of a dataset representing a beating heart. We treat all four dimensions of this data (3 spatial and time) equally[6], and then compute the persistence diagrams (Figures 5(a)-5(d)). In Figure 5(e) we display graphs of the Betti numbers of the sublevel sets. Blue, red, green and pink correspond to 0-3 dimensional Betti numbers, respectively.

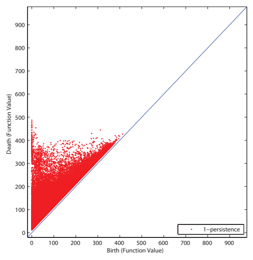
## 8 Summary and Future Work

In this paper, we showed that our approach can be used to compute persistent homology for large cubical data-sets in arbitrary dimensions. Our experiments show that our method is more efficient with regard to time and memory than the existing implementations. The reduction of memory usage is especially important, as it enables the use of persistent homology for much larger datasets.

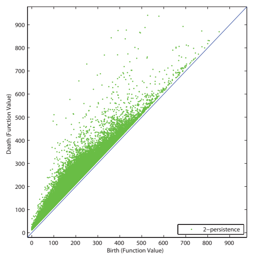
[6] In general, this may not be the right approach, as it does not assume the non-reversibility of time.



(a) 0D Persistence

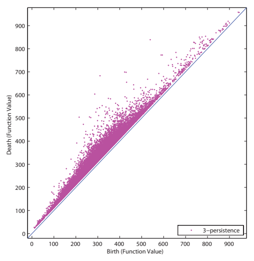


(b) 1D Persistence

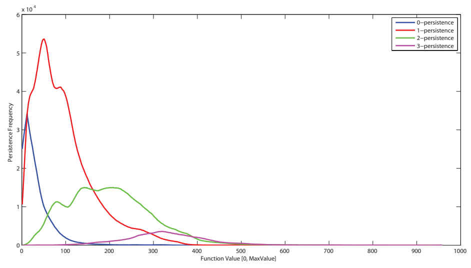


(c) 2D Persistence

(a) 0D Persistence (b) 1D Persistence (c) 2D Persistence



(d) 3D Persistence



(e) Betti Numbers of Sublevel Sets

**Fig. 5** Persistence diagrams of a beating heart.

There is a wide range of directions to be considered in the future research. We consider further development of the proposed method. In particular, a parallel implementation is a promising option. Further reduction of memory usage and moving towards out-of-core computations are important directions, but also very challenging.

As pointed out by one anonymous reviewer, another reason for the under-usage of persistence in visualization is the lack of a good interface to analyze and interact with persistence. With an efficient implementation in hands, we are motivated to further explore this research direction.

## Acknowledgments

This work was supported by the Austrian Science Fund (FWF) grant no. P20134-N13 and the Austrian COMET program. The authors would like to thank Prof. Herbert Edelsbrunner and Dr. Michael Kerber for the fruitful discussions.

## References

1. Computer Assisted Proofs in Dynamics: CAPD Homology Library, http://capd.ii.uj.edu.pl.
2. C. L. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *Proceedings of IEEE Visualization*, pages 167–174, 1997.
3. P. Bendich, H. Edelsbrunner, and M. Kerber. Computing robustness and persistence for images. In *Proceedings of IEEE Visualization*, volume 16, pages 1251–1260, 2010.
4. S. Biasotti, A. Cerri, P. Frosini, D. Giorgi, and C. Landi. Multidimensional size functions for shape comparison. *J. Math. Imaging Vis.*, 32(2):161–179, 2008.
5. H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, 43(1):42–58, 2010.
6. C. Chen and M. Kerber. An Output-Sensitive Algorithm for Persistent Homology. In *Proceedings of the 27th annual symposium on Computational geometry*, 2011.
7. C. Chen and M. Kerber. Persistent homology computation with a twist. In *27th European Workshop on Computational Geometry (EuroCG 2011)*, 2011.
8. D. Cohen-Steiner, H. Edelsbrunner, and J. Harer. Stability of persistence diagrams. *Discrete and Computational Geometry*, 37(1):103–120, 2007.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT press, 2009.
10. H. Edelsbrunner and J. Harer. *Computational Topology, An Introduction.* American Mathematical Society, 2010.
11. H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.
12. D. Freedman and C. Chen. *Computer Vision*, chapter Algebraic topology for computer vision. Nova Science, To appear.
13. H. Freudenthal. Simplizialzerlegungen von beschränkter Flachheit. *Annals of Mathematics*, 43(3):580–582, 1942.
14. A. Gyulassy, V. Natarajan, V. Pascucci, and B. Hamann. Efficient computation of morse-smale complexes for three-dimensional scalar functions. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1440–1447, 2007.
15. T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*, volume 157 of *Applied Mathematical Sciences*. Springer-Verlag, 2004.
16. G. Kedenburg. Persistent Cubical Homology. Master's thesis, University of Hamburg, 2010.
17. R. Kershner. The number of circles covering a set. *American Journal of Mathematics*, 61(3):665–671, 1939.
18. N. Milosavljevic, D. Morozov, and P. Skraba. Zigzag Persistent Homology in Matrix Multiplication Time. In *Proceedings of the 27th annual symposium on Computational geometry*, 2011.
19. D. Morozov. Dionysus : a C++ library for computing persistent homology. http://www.mrzv.org/software/dionysus/.
20. D. Morozov. Persistence algorithm takes cubic time in worst case. *BioGeometry News, Dept. Comput. Sci., Duke Univ., Durham, North Carolina*, 2005.
21. M. Mrozek and T. Wanner. Coreduction homology algorithm for inclusions and persistent homology. *Computers and Mathematics with Applications, accepted*, 2010.
22. M. Mrozek, M. Zelawski, A. Gryglewski, S. Han, and A. Krajniak. Extraction and analysis of linear features in multidimensional images by homological methods. preprint, 2010.
23. J. R. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, Redwook City, California, 1984.
24. V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26(58):1–8, 2007.
25. D. Strömbom. Persistent homology in the cubical setting: theory, implementations and applications. Master's thesis, Luleå University of Technology, 2007.
26. C. Zong. What is known about unit cubes. *Bull. Amer. Math. Soc. 42 (2005), 181-211*, 2005.