PRIP-TR-122

Efficient Computation of Persistent Homology for Cubical Data

Hubert Wagner, Chao Chen, Erald Vuçini

Pattern

Recognition &

Image

Processing

Group

Institute of

Computer Graphics and

Algorithms

Pattern Recognition and Image Processing Group
Institute of Computer Graphics and Algorithms
Vienna University of Technology
Favoritenstr. 9/1863, 4 Stock
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
hubert.wagner@ii.uj.edu.pl
E-mail: chao.chen@ist.ac.at
vucini@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-122 December 10, 2010

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 persistent homology results for 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 application of persistent homology is still at an early stage, closely related concepts like Size functions [5], contour trees [3, 7], Reeb graphs [24] and Morse-Smale complexes [13] 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, 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 [14], 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). 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

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

terms of memory consumption. In this work we provide an efficient and compact algorithm for this step, using techniques from (non-persistence) cubical homology [14] (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 (in the complex size, which is larger than input size). Morozov [20] formulated a worst case scenario for which the persistence algorithm requires cubic time. When focusing on 0-dimensional homology, union-find data structures can be used to compute persistence in time $O(n\alpha(n))$ [10], where α is the inverse of the Ackermann functions and n is input size. For practical applications, this approach requires linear time . Milosavljevic et al. [18] compute persistent homology in matrix multiplication time $O(n^\omega)$ where the currently best estimation of ω is 2.376. Although such algorithm has a better theoretical complexity, it is unclear whether it is better than the standard persistent algorithm in a real implementation.

In terms of implementation, Morozov [19] provides a C++ implementation for standard persistence algorithm. Kerber and Chen [16] 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 very 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 [15, 25]. They do not, however, tackle the problem of performance. In [25], experiments with datasets containing several thousands of cubes are reported. In comparison, real world applications require processing of data in the range of millions or billions of cubes.

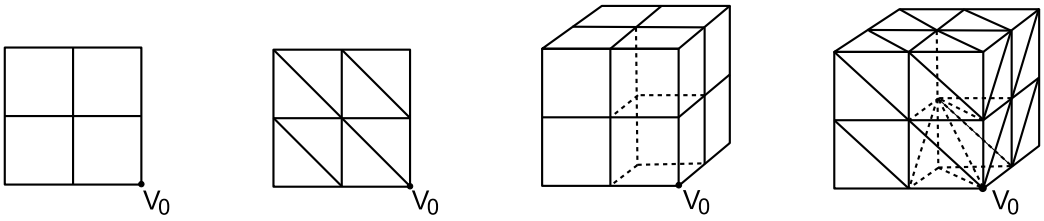
Recently, Mrozek and Wanner [21] showed that cubical persistent homology can be used for real-world data. A detailed performance summary is

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 size of the input data, the complexity is at least quadratic. In Section 7, we provide comparison results of 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. [4] uses octrees to approximate the input. A simplicial complex of small size is then used to complete persistence computation. When approximative results are satisfactory, this approach can be used to efficiently handle large cubical datasets.

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.



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

Figure 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).

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* 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 image. In Figure 1 we show 2-dimensional and 3-dimensional cubical complexes, describing a 2D image of size 3×3 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* [12, 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*. In specific, the boundary of a set of cells is the modulo 2 sum of the boundaries of each of its element. 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-simplices.

Persistent homology. We review persistent homology [10, 11], focusing on $\mathbb{Z}_2$ homology [23].

Given a topological space $\mathbb{X}$ and a *filter 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 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 two dimensional

plane, each corresponding to a persistence homology class. The coordinates of such point is the birth and death time of the class.

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

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

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 complex). It requires preprocessing of the data (also see Figure 3). 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, we compute a *filtration* of the complex and generate *sorted boundary matrix*. 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 algorithm* extends the function to all cells of the complex. In specific, each cell takes 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.

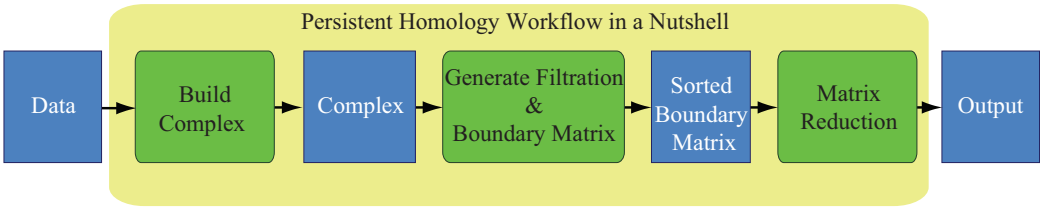


Figure 2: A workflow of the persistent homology computation.

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 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. In specific, we explore the regular structure of cubical complex and adapt a compact data structure which has shown its power in non-persistent cubical homology.

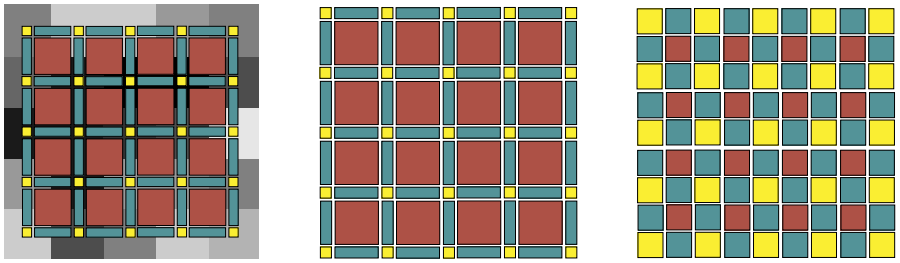
Cubical complex representation. We first describe CubeMap, a compact representation of cubical complexes. To the best of our knowledge, similar structure was first introduced in CAPD library [1] for non-persistent cubical homology.

For an example 2D image with 5×5 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 9×9 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.

The above-mentioned properties generalize for arbitrary dimensions. This due to the inductive construction of cubical complexes, related to cubes being products of intervals.

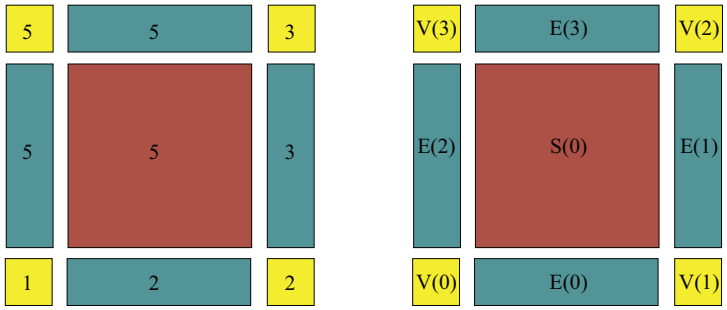
Let us consider input data of dimension d and size w^d, where w is the number of vertices in each dimension. We store information attached to cells in a d-dimensional array with $(2w-1)^d$-elements. This array is composed of overlapping copies of arrays of size 3^d. We call such an array the $CubeMap$.

The major advantage of the proposed data-structure is the improved memory efficiency. Boundary relations are implicitly encoded in the coor-



(a) (b) (c)

Figure 3: 3(a) Cubical complex built over a gray-scale 2D image with 5×5 pixels. 3(b) The cubical complex itself. 3(c) the corresponding CubeMap, all informations for filtration-building are encoded in a 9×9 array.



(a) (b)

Figure 4: **Left:** Values of f assigned to vertices and extended to all cubes. **Right:** Cells are assigned indices in the filtration. These indices are separate for each dimension.

dinates of cells. The coordinates itself are also implicit. Furthermore, we can randomly access each 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). It uses the data-structure in this section. Specifically, we use CubeMap 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 filtration index to C_j
6:          f(C_j) ← f(V_i).
7: for each cube C_i of K do
8:    row ← filtration index of C_i
9:    for each cube B_j in boundary of C_i do
10:      col ← filtration index of B_j
11:      boundary matrix(row, col) ← 1
```

5 Sizes of Complexes

When switching from simplicial complex to cubical complex, an obvious efficiency improvement, in both time and memory, is that the complex size is significantly reduced. 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 complex has a significant impact.

In this section, we analyze how the ratio of the sizes of the simplicial and cubical complexes increases with regard to the data dimension. For simplicity we disregard boundary effects, assuming that the number of cells lying on the boundary (or *frontier*) is insignificant, if complexes are large. Such analysis provides a theoretical ground for our approach. In Theorem 5.1 we show that

such ratio increases exponentially with the dimension of the data. This is a strong motivation for the usage of cubical approaches, such as ours.

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

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. It is nontrivial to give an exact formula of ρ_d, since the minimal-cardinality cube-triangulation is an open problem [26]. By τ_d we denote a number of d-simplices in a triangulation of a d-cube. Here we give a lower-bound of ρ_d for $d \leq 7$. The following theorem holds:

Theorem 5.1 *When $d \leq 7$, ρ_d increases at least exponentially to d.*

Proof. We give a lower bound for this ratio by triangulating all cubes of a cubical complex separately in each dimension. 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 $\rho_d \geq \frac{\sum_{i=0}^{d}\binom{d}{i}\tau_i + \sum_{i=0}^{d-1}\binom{d}{i+1}(\tau_{i+1}-1)}{2^d}$. In Table 1 we present the values for different dimensions. We distinguish 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.

Table 1: Lower-bound of the size ratio ρ_d.

Dimension (d)	1	2	3	4	5	6	7
τ_d (optimal)	1	2	5	16	67	308	1493
τ'_d (Freudenthal)	1	2	6	24	120	720	5040
ρ_d for optimal	1.0	1.5	2.75	5.625	12.937	33.968	90.265
ρ'_d Freudenthal	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.

Filtration algorithm. We use a 2-pass modification of the standard filtration algorithm. Reversing the iteration order simplifies the first part of the algorithm, which required random accesses to memory. This is typically slow, due to caching issues. Second part allows for sequential iteration, which makes it fast.

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: $O(d3^d n + d^2 2^d n)$.

Using the properties of CubeMap, we can reduce the 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.

We compare our implementation (referred as CubPers) to three existing implementations: **1)** the method introduced by Kerber and Chen [16], referred to as SimpPers, which uses simplicial complexes. Both SimpPers and CubPers use the same reduction algorithm, but our approach uses cubical complexes and CubeMap to accelerate the filtration process, **2)** an implementation of the reduction algorithm, Dionysus, by Morozov [19]. Since this implementation takes a filtration as input, we only measure the time and memory consumption of building the boundary matrix and reducing it. **3)** CAPD is an implementation by Mrozek [21], which is 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 from the Volvis repository[2]. In order to explore 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. Memory usage is reduced by an order of magnitude. This is extremely important, as it enables the usage of much larger data-sets. We observe that SimpPers is also very efficient in time. However, the usage of cubical complexes allows our approach to be even faster.

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 (the choice was arbitrary). The distribution is uniform. Number of vertices (1000000) is constant for all dimensions. We can see that performance deteriorates exponentially. This is understandable, since size of cubical complex increases exponentially in time (2^d). Size of its boundary matrix increases even faster ($d2^d$).

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.

	50^3	100^3	150^3	200^3	256^3
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. Several down-sampled version of the original dataset were used. For specific cases the times are not reported due to memory or time limitations. For SimpPers and CubPers, we report both filtration-building time and for reduction time, the whole computation is the sum of the two times.

	50^3	100^3	150^3	200^3	256^3
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	0.067	0.12	0.23	0.87	4.80	17.70

In Table 5 we report the timings and memory consumptions for several 3D datasets and a 4D time-varying data consisting of 32 timesteps. All the 3D datasets can be run on a commodity PC. It is clear, that our implementation is efficient, especially in terms of memory usage. This is especially important, since memory limits the size of applicable data.

Understanding time-varying data with persistence. With our efficient tool, we are able to compute persistent homology of a dataset representing an animation of a beating heart. We treat all four dimensions of this datafile (3 spacial and time) equally. See section 8 for a short discussion. We conclude this section by briefly discussing the computed persistence diagrams.

In Figures 5(a)-5(d) we show the persistence diagrams for the 4D Heart dataset. 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.

Observation reveals that during the filtration, most 0 and 1-dimensional homology classes die before the value 500. However, this is not the case for 2 and 3-dimensional classes. This suggests that the structure of the space changes drastically along the filtration. Focusing on value range larger that 500 might be a first step in the process of further data analysis.

We think, that in case of high-dimensional, and especially time-varying data, direct analysis of such diagrams might give certain hints about the structure of the data, but in general such analysis is very challenging. However, recent advances [6, 8] show that analyzing different spaces by computationally comparing their persistence-diagrams is a promising option. We believe that calculating persistent homology should be one step in a more complex process of data analysis.

8 Summary and Future Work

We showed that our approach can be used to compute persistent homology for large data-sets in arbitrary dimensions. Our experiments show that is more efficient with regard to time and memory than existing persistence implementations.

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

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)
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)
Heart (4D)	$256 \times 256 \times 14 \times 32$	13243	(20.20+1.38)

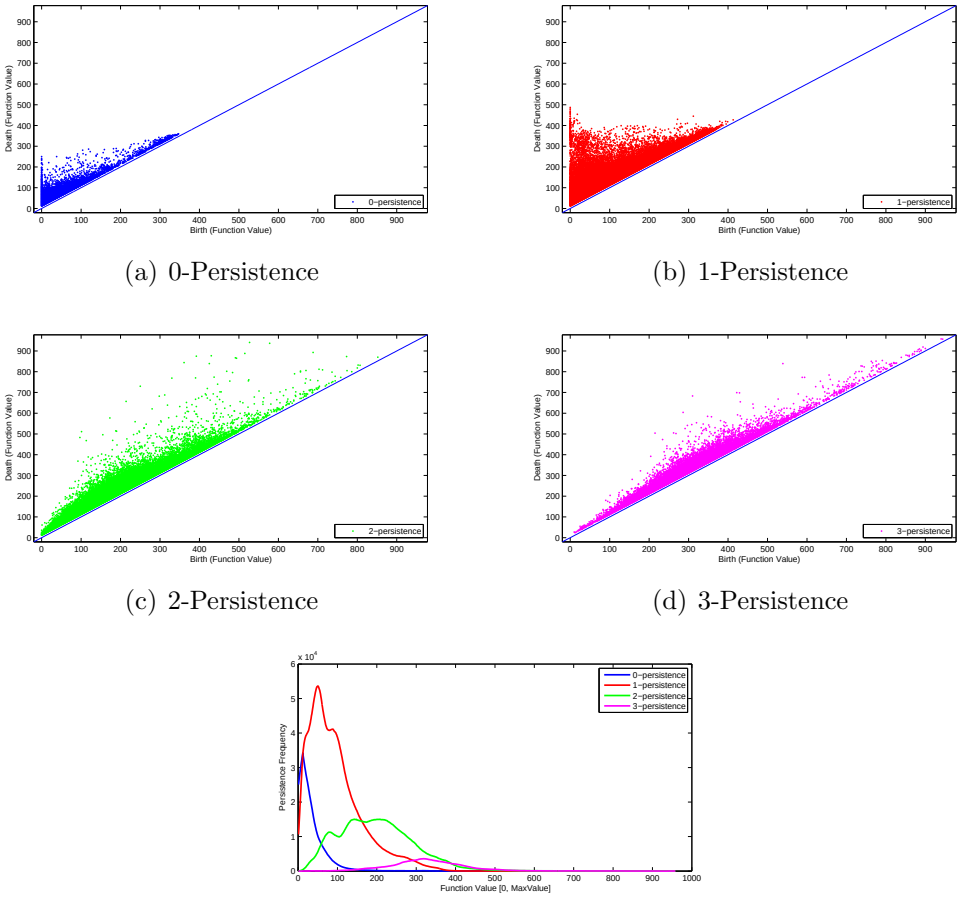
usage is another important research direction, but also challenging.

Currently, in the case of time-varying data, all dimensions are considered homogeneous. While this can give insight into the analyzed physical process, it does not assume non-reversibility of time. Our approach can be easily suited for a directed case as well, by altering the connectivity along selected dimensions.

A so-called multidimensional-persistence [6], taking multiple filtration functions, is an interesting new research direction. Computationally, it can be reduced to the one-dimensional case, so our framework can also be used in this setting.

Acknowledgments

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



(a) 0-Persistence

(b) 1-Persistence

(c) 2-Persistence

(d) 3-Persistence

(e) Persistence Statistics

Figure 5: Graphs showing persistent information.

References

[1] Computer assisted proofs in dynamics: Capd homology library, http://capd.ii.uj.edu.pl.

[2] Volvis: Voxel data repository, 2010.

[3] C. L. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *Proceedings of IEEE Visualization*, pages 167–174, 1997.

[4] P. Bendich, H. Edelsbrunner, and M. Kerber. Computing robustness and persistence for images. In *Proceedings of IEEE Visualization*, volume 16, pages 1251–1260, 2010.

[5] 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.

[6] F. Cagliari, B. Di Fabio, and M. Ferri. One-dimensional reduction of multidimensional persistent homology. *Proc. Amer. Math. Soc.*, 138(8):3003–3017, 2010.

[7] 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.

[8] D. Cohen-Steiner, H. Edelsbrunner, and J. Harer. Stability of persistence diagrams. *Discrete and Computational Geometry*, 37(1):103–120, 2007.

[9] T. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[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] H. Freudenthal. Simplizialzerlegungen von beschränkter Flachheit. *Annals of Mathematics*, 43(3):580–582, 1942.

[13] 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.

[14] T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*, volume 157 of *Applied Mathematical Sciences*. Springer-Verlag, 2004.

[15] G. Kedenburg. Persistent Cubical Homology. Master's thesis, University of Hamburg, 2010.

[16] M. Kerber and C. Chen. An improvement of persistence reduction. Technical Report, IST Austria, Nov 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. Research Report RR-7393, INRIA, 09 2010.

[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. Krajniak, A. Gryglewski, and S. Han. Homological Methods in Feature Extraction of Multidimensional Images. pages 1–5, 2009.

[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.