Homology Computations via Acyclic Subspace

Piotr Brendel[1], Paweł Dłotko[1⋆], Marian Mrozek[1], Natalia Żelazna[2]

[1] Institute of Computer Science, Jagiellonian University
piotr.brendel@ii.uj.edu.pl, pawel.dlotko@ii.uj.edu.pl,
marian.mrozek@ii.uj.edu.pl
[2] Motorola Solutions
natalia.zelazna@motorolasolutions.com

Abstract. Homology computations recently gain vivid attention in science. New methods, enabling fast and memory efficient computations are needed in order to process large simplicial complexes. In this paper we present the acyclic subspace reduction algorithm adapted to simplical complexes. It provides fast and memory efficient preprocessing of the given data. A variant of the method for distributed computations is also presented.

Keywords: Homology algorithms, reduction algorithms, acyclic subspace method

1 Introduction

In this paper we describe two variants of the acyclic subspace homology algorithm introduced in [9] for cubical sets. The method presented there is adapted to simplicial complexes. Moreover, we show how to extend this algorithms for the purposes of distributed computations.

2 Preliminaries

We recommend [10] as a standard introduction to classical homology theory. In this paper a finite family of finite sets $\mathcal{S}$ is called an *abstract simplicial complex* if for every $P \in \mathcal{S}$ and for every $Q \subset P$ we have $Q \in \mathcal{S}$. [3] An element $P \in \mathcal{S}$ is called a *simplex*. For $P \in S$ and $Q \subset P$ Q is called a *face* of P. A simplex $P \in \mathcal{S}$ is said to be *maximal* if there is no simplex $Q \in \mathcal{S}$ such that $P \subsetneq Q$. Throughout this paper $S_{max}(\mathcal{S})$ denotes the set of maximal simplices of $\mathcal{S}$. The *algebraic closure* of a simplex P, denoted by $cl(P)$ is a family of simplices consisting of P and all its faces. The closure of a family of simplices $\mathcal{K}$ is $cl(\mathcal{K}) = \bigcup_{P \in \mathcal{K}} cl(P)$. For a given simplex Q belonging to simplicial complex $\mathcal{S}$ its *neighbourhood* consists of

⋆ Corresponding author

[3] One may think here of geometrical simplex being represented as a set of labels of its vertices.

all maximal simplices from $\mathcal{S}$ having nonempty intersection with Q. We denote this set by:

$$n(Q) = \{P \in \mathcal{S} \mid Q \cap P \neq \emptyset \text{ and } P \text{ is a maximal in } \mathcal{S}\}.$$

Dimension of a simplex P is $dim(P) = card(P) - 1$. For a simplicial complex $\mathcal{S}$ by $S_0(\mathcal{S})$ we denote the set of all the vertices of $\mathcal{S}$ (i.e. its 0-dimensional simplices) and we make a technical assumption that every vertex from S_0 has a unique label. In the sequel we use an extra data structure built upon the simplicial complex, namely a hash table [1], dentoted by H, whose keys are labels of vertices and for each key the value is the list of all maximal simplices containg the vertex labelled with this key. For a survey of cubical homology and cubical complexes we refer to [5].

In this paper we are extensively using two classical topological concepts - exact sequence of a pair and Mayer-Vietoris sequence [10]. From the exact sequence of a pair is clear, that a subcomplex with trivial homology can be removed from the initial complex without changing its reduced homology. From Mayer-Vietoris sequence it follows, that a simplex can be add to the constructed acyclic subcomplex iff its intersection with the acyclic subcomplex has trivial reduced homology.

3 Incidence Graph

We say that a graph $G = (V, E)$ is an *incidence graph* of a simplicial complex $\mathcal{S}$ if V is the set of maximal simplices of $\mathcal{S}$ and $(S_1, S_2) \in E$ if $S_1 \cap S_2 \neq \emptyset$. An *augmented incidence graph* is a triple (V, E, C) where (V, E) is the incidence graph and C is the list of connected components of incidence graph, in which each connected component is represented by a single maximal simplex from this component. We will use augmented incidence graphs to retrieve all the information about neighourhoods in a simplicial complex, necessary in the process of constructing an acyclic subset.

In this section we show an algorithm constructing such a graph for a given simplicial complex. The input data for this agorithm is the list of maximal simplices $S_{max}(\mathcal{S})$ and VertexHash H, described in Section 2. For each vertex v we consider the list $H[v]$ storing the maximal simplices that contain v. $\mathcal{Q}$ denotes the queue used to store simplices which have not yet been added to the incidence graph and whose neighbours are already there. Functions `Enqueue` and `Dequeue` are standard operations on queues and their description can be found in [1].

Theorem 3.1 Algorithm 3.1 stops and constructs the incidence graph $G = (V, E, C)$ for simplicial complex $\mathcal{S}$ in $O(card(V) \cdot dim(\mathcal{S}) \cdot deg(H))$ where $dim(S) = max_{P \in \mathcal{S}}\{dim(P)\}$ and $deg(H) = max_v\{length(H[v])\}$. Moreover, for each connected component $G' \subset G$ its set of nodes $V(G')$ equals to the set of maximal simplices in the corresponding connected component $\mathcal{S}' \subset \mathcal{S}$.

Algorithm 3.1 IncidenceGraph(MaximalSimplexList $S_{max}(\mathcal{S})$, VertexHash H)

```
1: V := ∅; E := ∅; C := ∅; Q := EmptyQueue;
2: for all Simplex P ∈ S_max(S) do
3:     if P ∉ V then
4:         C := C ∪ {P};
5:         Enqueue(Q, P);
6:         while Q ≠ ∅ do
7:             Simplex current := Dequeue(Q);
8:             V := V ∪ {current};
9:             for all Vertex v ∈ current do
10:                for all Simplex neighbour ∈ H[v], neighbour ≠ current do
11:                    if neighbour ∉ V then
12:                        e := (current, neighbour); E := E ∪ {e};
13:                        if neighbour ∉ Q then
14:                            Enqueue(Q, neighbour);
15: return Graph(V, E, C);
```

Proof Obviously V contais all maximal simplices from $S_{max}(\mathcal{S})$. Pair $(S_1, S_2) \in E$ only if S_1 and S_2 share a vertex. Therefore $S_1 \cap S_2 \neq \emptyset$, satisfying incidence graph definition. Simplex P is added to C in line 4 only if $P \notin V$ which means $P \cap S = \emptyset$ foreach $S \in V$ and P represents new connected component G'. Two simplices S_1 and S_2 belong to the same G' iff there exist path in G' connecting S_1 and S_2. In that case there exist path connecting S_1 and S_2 in $\mathcal{S}$, so they also belong to the same connected component in $\mathcal{S}' \subset \mathcal{S}$. Simplex P is added to $\mathcal{Q}$ only once, so **while** loop in line 6 always stops. Internal **for all** loop in line 9 is performed for every d-dimensional simplex at most $d * h$ times where $h = deg(H)$ as described above. □

4 Acyclic subset

The classical way of computing homology consists in finding the Smith Normal Form of the boundary maps [10]. The complexity of the classical algorithm is supercubical. Among the methods intended to speed up the computations are the reduction algorithms which aim at finding a smaller complex with the same homology as the original complex $\mathcal{S}$ [6, 8]. We recall that a simplicial complex $\mathcal{A}$ is acyclic if $\mathcal{A}$ has the same homology as a single point. The acyclic subspace algorithm [9] is a reduction algorithm based on the general observation that if $\mathcal{A}$ is an acyclic subcomplex of a connected complex $\mathcal{S}$, then:

$$H_n(\mathcal{S}) \cong \begin{cases} H_n(\mathcal{S}, \mathcal{A}) & \text{for } n \geq 1 \\ \mathbb{Z} \oplus H_n(\mathcal{S}, \mathcal{A}) & \text{for } n = 0 \end{cases}$$

A reduction method presented in this paper relies on a fast algorithm constructing a possibly large acyclic subset $\mathcal{A}$ of $\mathcal{S}$ and then computing the relative homology of the pair $(\mathcal{S}, \mathcal{A})$. Relative homology requires knowledge only of elementary

simplices in the neighborhood of $\mathcal{S} \setminus \mathcal{A}$ thanks to the excision property [10]. Since the constructed acyclic subset is a closed subset in the sense of [8], we have the following theorem ([8] Theorem 3.5).

Theorem 4.1 If $\mathcal{A}$ is closed in $\mathcal{S}$ then:

$$H(\mathcal{S} \setminus \mathcal{A}) \cong H(\mathcal{S}, \mathcal{A}).$$

Therefore, after construction of acyclic subset $\mathcal{A} \subset \mathcal{S}$ it suffices to compute homology of the S-complex $\mathcal{S} \setminus \mathcal{A}$ (for definitions and properties of S-complexes we refer to [8]). Having simplicial complex $\mathcal{S}$ represented by list of its maximal simplices we use them to construct acyclic subcomplex $\mathcal{A}$. Finally we exclude those simplices from of $\mathcal{S}$ that are contained in $\mathcal{A}$. For simplices that left we need to store information about their intersection with $\mathcal{A}$. We can now compute Betti numbers for such complex. It is also possible to use this method for cohomology computations as shown in [2].

5 Constructing acyclic subset

A linear time algorithm constructing acyclic subspace for cubical sets is proposed in [9]. In the following section we present two approaches to contructing acyclic subset for simplicial complexes. All these algorithms apart from the incidence graph, need a function deciding whether a simplex may be added to the constructed acyclic set. Such a function, named `AcyclicityTest` is described in Section 7.

First algorithm, referenced in the following sections as `AccST`, is an extension of algorithm presented in [9]. The main difference lies in the way the neighbourhood is determined. In the case of cubical sets as described in [9] obtaining this information is trivial but having simplices we need a way of finding neighbours, which in this case will be an incidence graph (described in Section 3). For given simplex P we denote list of its neighbours by $n(P)$. Another difference is that instead of one, we construct several acyclic subsets in $\mathcal{S}$ and then join them with a spanning tree. To do this we need two auxilary functions: `FindSimplexNotInAccSub` and `CreateSpanningTree`. Both of them are standard graph algorithms of which detailed description can be found in [1]. First one finds simplex that has no intersection with acyclic subset using breadth-first search algorithm. It returns `NULL` if it cannot find such. Second one takes as an input a list of elements which are simplices representing disjoint parts of constructed acyclic subset (the same way as connected components are represented in the incidence graph). Next it finds the shortest paths connecting subsets. Each path is a list of one-dimensional simplices. Having graph structure, in which nodes are disjoint acyclic subsets and edges are paths connecting them, we use Kruskal algorithm [1] to create spanning tree that joins parts of acyclic subset. Description of this procedure can be found in the proof of Theorem 5.1.

Theorem 5.1 Algorithm 5.1 stops and creates acyclic subset $\mathcal{A}$ for given simplicial complex $\mathcal{S}$ represented by incidence graph $G = (V, E, C)$.

Algorithm 5.1 AccST(IncidenceGraph (V, E, C))

```
 1: A := ∅; Q := EmptyQueue;
 2: for all Simplex P ∈ C do
 3:     L :=EmptyList;
 4:     while P ≠ ∅ do
 5:         A := A ∪ {P};
 6:         Enqueue(Q, P);
 7:         while Q ≠ ∅ do
 8:             Simplex Q := Dequeue(Q);
 9:             for all Simplex S ∈ n(Q) \ A do
10:                 if AcyclicityTest(A, S) = true then
11:                     A := A ∪ {S};
12:                     Enqueue(Q, S);
13:         P :=FindSimplexNotInAccSub(V, E, P, A);
14:         if P ≠ NULL then
15:             L := L ∪ {P};
16:     A := A ∪ CreateSpanningTree(L);
17: return A;
```

Proof Simplex P is added to $\mathcal{Q}$ and to acyclic subset $\mathcal{A}$ at the same time in lines 11 and 12. Since P may be added to $\mathcal{A}$ only once and we have finite numbers of simplices inner `while` loop in line 7 always stops. `FindSimplexNotInAccSub` and `CreateSpanningTree` are respectively BFS and Kruskal algorithms [1] so they both stop. Every simplex that is found with `FindSimplexNotInAccSub` in line 13 is added to the acyclic subset. Because of that and finiteness of V `while` loop in line 4 stops. Because number of simplices in C is also finite we are sure that algorithm stops. Every single simplex is acyclic, so starting with the one that represents a connected component of incidence graph (as described in Section 3) we begin construction of acyclic subset $\mathcal{A}$. As long as we can find another simplex having acyclic intersection with $\mathcal{A}$ we can add it to $\mathcal{A}$ without losing its acyclicity. Once we cannot find such simplex we look in the same connected component for another one that has no intersection with $\mathcal{A}$ and we build acyclic subset around it as described above. We stop this procedure when there are no simplices that do not intersect $\mathcal{A}$. Now lets assume, we have few disjoint subsets of $\mathcal{A}$ and we want to connect them into acyclic subset. First we need to find paths (lists of one-dimensional simplices) joining them. Since all parts of acyclic subset $\mathcal{A}$ are contained in the same connected component we can always find a path between every two of them. Such paths though can intersect eachother or even other parts of $\mathcal{A}$ and thus create cycles. To prevent this we need special procedure while adding them to acyclic subset. During construction of a spanning tree we consider two types of acyclic subsets - subsets already organized into spanning tree and subsets that we want to connect. We choose an element of second type that has a direct connection to an element of first type. Then, we move on path *"towards"* spanning tree unless we find a simplex that has intersection with $\mathcal{A}$ (in the worst case it will be last simplex on path). Since paths were minimal,

this part of acyclic subset cannot be the one we started from. Intersection of path with each part of $\mathcal{A}$ is zero-dimensional simplex, which is acyclic. Since both subsets and path itself are acyclic and intersection of each two of them are either acyclic or empty we can connect them without losing acyclicity. □

Algorithm 5.2, referenced in the following sections as `AccIG`, constructs simultaneously the incidence graph and an acyclic subset. The vertices of the resulting graph G are these simplices from $S_{max}(\mathcal{S})$ which are not in the acyclic subset. This algorithm requires some auxiliary functions. Among these functions there are general graph functions `AddToGraph` and `RemoveFromGraph` which respectively adds and removes a simplex from a given graph. Another one is `EnqNeighb`, which adds to queue all neighbours of given simplex that have not been added neither to the queue nor to the acyclic subset yet.

Algorithm 5.2 AccIG(MaximalSimplexList $S_{max}(\mathcal{S})$, VertexHash H)

```
1: V := ∅; E := ∅; A := ∅; Q := EmptyQueue;
2: for all Simplex P ∈ S_max(S) do
3:     if P ∉ V and P ∉ A then
4:         A := A ∪ {P};
5:         EnqNeighb(P, H, Q);
6:         while Q ≠ ∅ do
7:             Simplex current := Dequeue(Q);
8:             if AcyclicityTest(A, current) = true then
9:                 A := A ∪ {current};
10:                EnqNeighb(current, H, Q);
11:                if current ∈ V then
12:                    RemoveFromGraph(current, V, E);
13:            else if current ∉ V then
14:                AddToGraph(current, V, E, H);
15:                EnqNeighb(current, H, Q);
16: return Graph(V, E), A;
```

Theorem 5.2 Algorithm 5.2 stops and returns an acyclic set $\mathcal{A}$ and a graph $G = (V, E)$, which is the incidence graph whose nodes are the maximal simplices in $S_{max}(\mathcal{S}) \setminus \mathcal{A}$.

Proof Simplex can be added to $\mathcal{Q}$ only if it is not in acyclic subset and its neighbour is being added to graph or acyclic subset. Since each simplex can be added to graph or acyclic subset at most once, the algorithm stops. We start building acyclic subset by finding first simplex that has not beed added neither to graph nor to the acyclic subset yet. Having found such we are sure it is not a neighbour of any simplex already processed. It means it represents new connected component and we can start build new acyclic subset $\mathcal{A}$. We extend it only by adding those maximal simplices that have acyclic intersection with $\mathcal{A}$. For every simplex on $\mathcal{Q}$ we either add it to acyclic subset or to the incidence graph,

which means that nodes of created incidence graph are all maximal simplices of $S_{max}(\mathcal{S})$ that have not been added to $\mathcal{A}$. □

6 Distributed computations

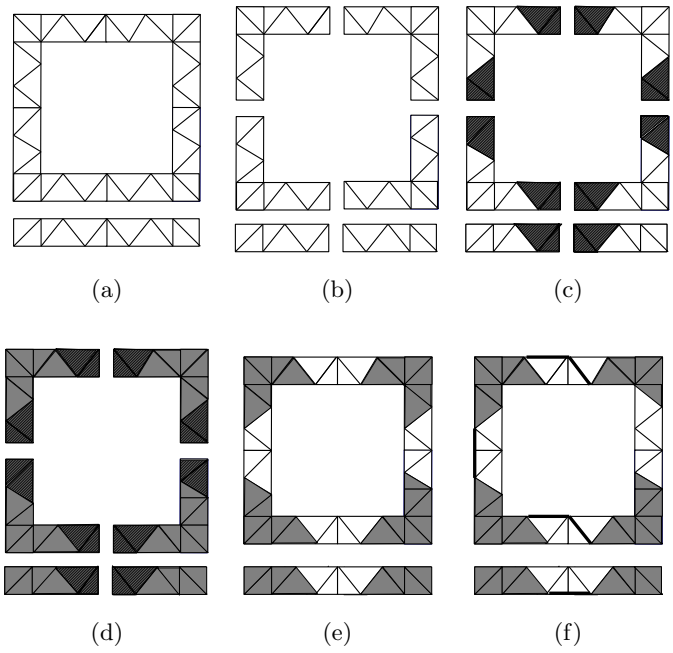

Fig. 1: (a) initial simplicial complex, (b) inital complex splitted into smaller complexes, (c) boundary simplices in complexes, (d) acyclic subsets in complexes, (e) combined results, (f) acyclic subsets joined with a spanning forest.

In this section we show how the algorithms that compute an acyclic subset for a given simplical complex can be used in distributed computations. The idea is to divide the initial complex into smaller ones, then construct an acyclic subset and incidence graph for each of them and finally combine the results into an acyclic subset and incidence graph for the initial complex. However, we need to ensure that after combining the results from the individual computations the obtained space is acyclic, i.e. we do not make cycles while connecting the acyclic subsets from the different packages. Moreover, we need a way to connect individual incidence graphs into the incidence graph of the initial complex. The whole procedure is very technical and resembles what we did in Algorithm 5.1 but in the more global scale. It needs to be emphasized that distributed computations

involve only construction of the incidence graph and acyclic subset for each package. After combining results from the individual reductions we create one complex for which we can perform homology computations (i.e. Betti numbers) just like in non-distributed case.

The first step is to split the initial list of maximal simplices of $\mathcal{S}$ (Figure 1a) into a lists $\mathcal{P}_i$ $i \in \{1, 2, ..., n\}$ such that $\bigcup_i \mathcal{P}_i = S_{max}(\mathcal{S})$ (Figure 1b). For our purposes we assume that $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ if $i \neq j$. For every $\mathcal{P}_i$ we defnie two sets: $BV_i := \bigcup_{i \neq j} \{S_0(\mathcal{P}_i) \cap S_0(\mathcal{P}_j)\}$ and $BS_i := \{Q \,|\, Q \in \mathcal{P}_i \wedge S_0(Q) \cap BV_i \neq \emptyset\}$. The elements of BS_i are referred to as the *boundary simplices* - simplices which have neighbourhood contained in other packages. (Figure 1c). In the process of constructing the acyclic subset $\mathcal{A}_i$ for each $\mathcal{P}_i$ we consider only those simplices that are not boundary simplices (Figure 1d). To do so, we need to change a little Algorithms 5.1 and 5.2 so they include such restriction. We will not present them here, but it is easy for the reader to do such modification. In our example acyclic subset is constructed from all simplices that are not boundary simplices, but in general case that is not true. Computations of lists of both incidence graphs G_i and acyclic subsets $\mathcal{A}_i$ may be performed sequentially or in distribution. In both cases we gain profits from lower memory usage, because list of simplices for which computations are performed are much smaller than the initial one. In the second case computations are performed much faster. Moreover, after constructing acyclic subset $\mathcal{A}_i$ we can discard all simplices contained in $\mathcal{A}_i$ from the incidence graph and construct new acyclic subset which is intersection of $\mathcal{A}_i$ with the set of maximal simplices that left in the incidence graph. In the latter case we save additional memory needed to store redundant simplices. Finally, after combining results (Figure 1e) into one incidence graph we obtain structure analogous to the one in Algorithm 5.1. We then create a spanning forest in which nodes are disjoint parts of the acyclic subset and edges are lists of one-dimensional simplices connecting them (Figure 1f).

Theorem 6.1 The family of simplices $\mathcal{A}$ constructed as above is acyclic subset of initial simplicial complex $\mathcal{S}$.

Proof By restricting acyclic subset algorithms to those simplices that are not boundary simplices we are sure that acyclic subsets in individual packages will not create cycles after combining them. The rest of the proof is analogous to the proof of Theorem 5.1 but isntead of spanning tree we deal with the spanning forest. □

7 Acyclicity tests

The `AcyclicityTest` function is a tool allowing to decide whether we can add a given cube or a simplex to the constructed acyclic subset. The function takes two arguments: the already constructed acyclic subset $\mathcal{A}$ and a simplex P. We distinguish two types of acyclicty tests:

– a *full test*, it returns `true` if and only if $\mathcal{A} \cap cl(P)$ is acyclic

– a *partial test*, it returns `true` if $\mathcal{A} \cap cl(P)$ is acyclic and `false` if it fails to prove that $\mathcal{A} \cap cl(P)$ is acyclic

In [9] some tests for testing acyclicity in cubical sets were proposed. Their main limitation is dimension of the complex. The full tests both in the cubical [9] and in the simplicial case are based on the idea of *tabulated configurations* for boundary elements. The number of configurations is 2^{3^d-1} for a d-dimensional cube and $2^{2^{d+1}}$ for a d-dimensional simplex. This makes the method prohibitive for $d > 3$ in the case of a cube [9] and for $d > 4$ in the case of a simplex [3]. The universal full test that works for every dimension is computation of the homology of $\mathcal{A} \cap cl(P)$. However, this method is computationally expensive in comparision to other tests.

We finish this section by introducing a partial test for the simplicial case. Given an acyclic subspace $\mathcal{A}$ and a d-dimensional simplex P we set $\mathcal{I} := \mathcal{A} \cap cl(P)$. The following test is based on the investigation of the maximal simplices of $\mathcal{I}$. If the number of maximal simplices of dimension $d-1$ is less than or equal to d and there are no maximal simplices of other dimensions then $\mathcal{I}$ is acyclic.

Algorithm 7.1 AcyclicityTest(Set $\mathcal{A}$, Simplex P)

```
1: I := MaximalSimplices(A ∩ cl(P));
2: d :=Dim(P); i := 0;
3: for all Simplex Q ∈ I do
4:       if Dim(Q) = d − 1 then
5:             i++;
6:       else
7:             return false;
8: if i > 0 and i <= d  then
9:       return true;
10: else
11:      return false;
```

Theorem 7.1 For a given set $\mathcal{A}$ and simplex P algorithm 7.1 returns `true` if $\mathcal{A} \cap cl(P)$ is acyclic and `false` if it fails to prove that $\mathcal{A} \cap cl(P)$ is acyclic.

8 Numerical experiments

Algorithms described above were implemented using `C++`. The code is available as a part of RedHom [12] library. To provide communication between processes during distributed computation `MPI` [4] was used. Both local and distributed approaches were compared with the coreduction homology algorithm [8], denoted in the following table by `CoRed`. `AccIG` and `AccST` are the algorithms introduced in Section 5. Column `size` denotes the number of maximal simplices used as input. `Running time` is the total time needed for building the incidence graph,

performing reductions (which could be either removal or acyclic subset or coreductions [8]), creating the simplicial complex from the list of maximal simplices and computing Betti numbers for such complex [5]. Computing generators after reduction of acyclic subset is still an open problem.

Space name	Size	Running time (s)			Memory usage (MB)		
		CoRed	AccIG	AccST	CoRed	AccIG	AccST
Bjorner	3079k	778	400	438	6167	1447	4880
Dunce Hat	4758k	1264	620	720	9590	2267	7587
Projective Plane	2799k	632	347	379	5580	1305	4404

In the following table we present comparision of running times of local and distributed algorithms. Columns `AccIG` and `AccST` denote the same as above while `DAccIG` and `DAccST` denote the outcome of distributed computations using `AccIG` and `AccST` algorithms respectively for a local construction of an acyclic subspace. In the last two cases computations were performed on 6 nodes simultaneously and `running time` includes maximal time needed for performing computations on 1 node.

Space name	Size	AccIG	AccST	DAccIG	DAccST
Bjorner	3079k	400	438	162	214
Dunce Hat	4758k	620	720	244	377
Projective Plane	2799k	347	379	211	208

Acknolwdgements

P.D. and M.M. are partially supported by Polish MNSzW, Grant N N201 419639
P.D. is partially supported by grant Nr IP 2010 046370.

References

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press and McGraw-Hill, 1990.
2. P. Dłotko, R. Specogna, Efficient Cohomology Computation for Electromagnetic Modeling, *Computer Modelling in Engineering & Sciences*, 60(2010), no.3, 247–277.
3. P. Dłotko, *Acyclic configurations for boundary elements of 3 and 4 dimensional simplices*, http://www.ii.uj.edu.pl/∼dlotko/accconf.html
4. W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1990.
5. T. Kaczynski, K. Mischaikow, M. Mrozek, Computational homology, Appl. Math. Sci. 157, Springer - Verlag, New York, 2004.

6. T. Kaczynski, M. Mrozek, M. Ślusarek, Homology computation by reduction of chain complexes, *Computers and Math. Appl.* 35(1998), 59–70.
7. K. Mischaikow, M. Mrozek, P. Pilarczyk, Graph approach to the computation of the homology of continuous maps, *Foundations of Computational Mathematics* 5(2005), 199–229.
8. M. Mrozek, B. Batko, Coreduction Homology Algorithm, *Discrete and Computational Geometry*, 41(2009), 96-118.
9. M. Mrozek, P. Pilarczyk, N. Żelazna, Homology algorithm based on acyclic subspace, *Computers and Mathematics with Applications*, 55(2008), 2395-2412.
10. J.R. Munkres, Elements of algebraic topology, *Addison-Wesley*, Reading, 1984.
11. *Computer Assisted Proofs in Dynamics*, http://capd.wsb-nlu.edu.pl
12. *The RedHom homology algorithms library*, http://redhom.ii.uj.edu.pl