Cubical Approximation and Computation of Homology

W.D. Kalies, K. Mischaikow, and G. Watson

Center for Dynamical Systems and Nonlinear Studies
Georgia Institute of Technology
Atlanta, GA 30332

July 9, 1998

Abstract

The purpose of this article is to introduce a method for computing the homology groups of cellular complexes composed of cubes. We will pay attention to issues of storage and efficiency in performing computations on large complexes which will be required in applications to the computation of the Conley index. The algorithm used in the homology computations is based on a local reduction procedure, and we give a subquadratic estimate of its computational complexity. This estimate is rigorous in two dimensions, and we conjecture its validity in higher dimensions.

1 Introduction

The computability of homology groups is well-known and is found in most standard textbooks, e.g. [18], and the classical algorithm is based on performing row and column operations on the boundary matrices as a whole and reducing them to the Smith Normal Form (SNF), which is known to exist for any integer matrix, [20]. The homology groups can then be immediately determined from this canonical form. However, explicit examples can be given for which this algorithm has a worst-case computational complexity which grows exponentially in both space (i.e., storage) and time with the size of the matrix, [11].

Many algorithms have been devised to improve this complexity bound, and we will mention them briefly, but the reader should consult the references for details. Kannan and Bachem [17] provide the first polynomial time algorithm for the computation of the SNF, and Chou and Collins improve it in [1]. Further improvement is given by Iliopoulos [14], and Storjohann [21] proposes a near optimal algorithm for computing the SNF. The algorithm's complexity for computing the SNF of square matrices equals that of the best known algorithm for computing the determinant. However, the number of operations required by all of these algorithms is superquadratic in the size of the input matrix, which corresponds to the size of the complex in homology computations. These complexity estimates count operations in a variety of ways, from the number of arthimetic operations to the number of bit operations, and we discuss this issue further in Section 5.

If it is known that there is no torsion, or one is only interested in the Betti numbers, then there are other methods which avoid the computation of the SNF for the boundary matrices. Delfinado and Edelsbrunner [3, 4] provide an algorithm which computes the Betti numbers of a simplicial complex in $\mathbb{R}^3$ in time $O(n\alpha(n))$, where n is the number of simplices and $\alpha(n)$ is the extremely slowly growing inverse of the Ackermann function. Their method requires that the complex be given in a sequential form known as a filtration. Friedman [12] exhibits an algorithm for computing the Betti numbers of a simplicial complex with using the combinatorial Laplacian and Hodge theory. Again, the reader should consult the references for details.

We note that in the standard development, a simplicial complex is usually assumed. This is certainly the case in [4], [12] and [18] and is usually the case for applications involving geometric modeling or computational geometry, c.f. [8], [9], and [19]. We have taken a different approach by using cubical complexes instead. Since the reduction algorithm we develop depends only on chain complexes, c.f. [16], we can guarantee that the computations yield the correct homology groups and gain some advantages of using cubes instead of simplices. We address some of these advantages here, but we refer the reader to [15] for a more extensive discussion on homology computations based on cubical complexes. Also, except for Delfinado and Edelsbrunner [3, 4], all of the works mentioned above require that the entire simplicial complex (or the entire boundary matrices) be stored in memory and manipulated. Such a requirement is not necessary for our algorithm, which we make clear in Section 2.3. Kaczyński, Mrozek, and Ślusarek [16] analyze the computation of homology via local reduction in the context of general chain complexes. Their Theorem 2 implies that the simplification (reduction) operations in our procedure preserve homology.

Finally, we introduce some standard notation which we will use. Let (C, ∂) be a finitely generated free chain complex with coefficients in $\mathbb{Z}$. If $\{C_k\}_{k\in\mathbb{Z}}$ is the gradation of C and $\{\partial_k : C_k \to C_{k-1}\}_{k\in\mathbb{Z}}$ is the gradation of ∂, then for simplicity we will write ∂c for $\partial_k c$ if $c \in C_k$ for a given k. Of course, ∂c is the alternating sum of $p-1$ dimensional cells contained in the (geometric) boundary of a p dimensional cell c. In a similar manner we can define the coboundary operator δ on the cochain complex. If we fix a basis U_k of C_k for each k, then we will define the bilinear form $\langle\cdot,\cdot\rangle : C_k \times C_k \to \mathbb{Z}$ on generators $u, u' \in U_k$ by

$$\langle u, u'\rangle = \begin{cases} 1 & \text{if } u = u' \\ 0 & \text{otherwise} \end{cases} .$$

For cells c and $b \in \partial c$, the *incidence number* of b in ∂c is $\langle \partial c, b \rangle$.

In Section 1.1 we describe the cubical complexes on which our algorithm is based, and in Section 2 we present outlines of the procedures used. Section 3 contains running times for computations performed by a similar, preliminary scheme which we have coded, but it differs slightly from the algorithm in Section 2 for which the coding has not been completed. In Section 4 we state a subquadratic complexity bound in the two-dimensional case. We conjecture that the computational complexity is subquadratic even in the higher-dimensional case, but we do not yet have a complete proof. The two-dimensional case has been proved, but for brevity the proof will be omitted and presented for the general case in a future work. Finally, in Section 5 we also briefly describe an outline for computing the Conley index of neighborhoods generated by a program developed by Dellnitz et.al. [6] called GAIO.

1.1 Cubical Approximations

Any compact subset $S \subset \mathbb{R}^d$ can be approximated using a type of regular cubical cell complex which can be stored in a binary tree. This approximation can be made as accurately as desired using a collection of d-cubes by executing the procedure SUBDIVIDE which we describe heuristically below. First define a *rectangular box* B in $\mathbb{R}^d$ as $B := \{(x_0, x_1, \ldots, x_{d-1}) \in \mathbb{R}^d : a_i \leq x_i \leq b_i \text{ for each } i\}$. We will refer to the rectangular boxes approximating S as either *boxes*, *cubes*, or *d-cells*. The collection will be referred to as a *cubical complex* and will be encoded in a binary tree whose *levels* will be numbered from 0 (at the root) to M. Level M is said to be at the *bottom* of the tree and is referred to as the *depth* of the tree.

• SUBDIVIDE:
Input: compact subset $S \subset \mathbb{R}^d$ and the final depth M of the tree.
Output: binary tree T with M levels encoding a collection of d-cells which approximates S.

Select a rectangular box B_0 such that $S \subset B_0$.
Create a node representing B_0. This will be the root node in the output tree.
for $i = 0$ to $i = M$
    $m := i \bmod d$.
    Denote all boxes present in the tree at level i by $B_{i,j}$ (j counts these boxes).
    for each box present in the tree at level i
        Divide $B_{i,j}$ in half in the x_m coordinate direction to obtain B^1 and B^2.
        if $S \cap B^1 \neq \emptyset$
            then create the left child of the node representing $B_{i,j}$.
        if $S \cap B^2 \neq \emptyset$
            then create the right child of the node representing $B_{i,j}$.

Remark: In order to simplify the subsequent analysis and computation, we will always choose M such that $M \bmod d = 0$. It is clear that M determines the size of the boxes and therefore also determines the accuracy of the final approximation for S. Each node in the tree corresponds to a box in a particular location in B_0, and hence we have a convenient method for storing and finding geometric information about the complex.

In line 7 we have $B^1 \cup B^2 = B_{i,j}$, and B^1 will be the half of $B_{i,j}$ which contains those points whose x_m coordinates are smaller than the x_m coordinates of the points in B^2. In lines 8-11 the new node represents B^1 or B^2 respectively. These final lines also imply that it is possible for a node in the tree to have 0, 1, or 2 child nodes when SUBDIVIDE exits. A tree is said to be *dense* if the only nodes which have exactly one child are at level $M-1$. A node which has two children will be referred to as a *branch point* and a node which has no children will be called a *leaf*. Every leaf of any tree created by the algorithm will be at level M. That is, SUBDIVIDE will not produce a tree like that in Figure 1.1.

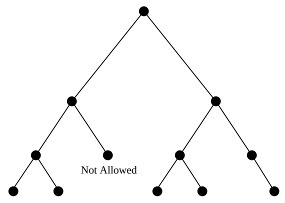


Figure 1.1: SUBDIVIDE will not produce such a tree

We now give several examples to illustrate the algorithm and its output.

Example 1.1 Let S be the unit square in $\mathbb{R}^2$ and set $B_0 = S$ and $M = 2$. The cubical complex produced by SUBDIVIDE is represented in a tree as shown in Figure 1.2. In fact, for this B_0 and for any value of M, SUBDIVIDE will create a full binary tree with M levels.

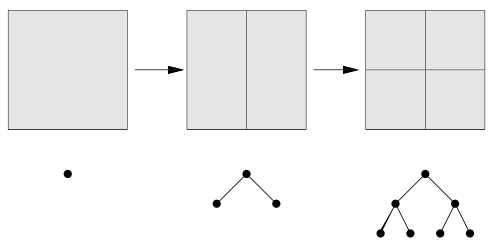


Figure 1.2: Square with $M = 2$ for Example 1.1

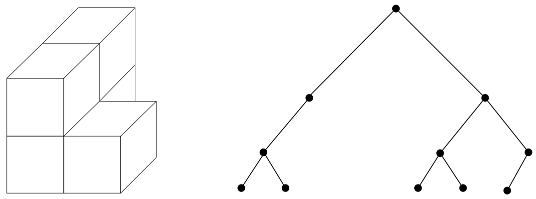


Figure 1.3: Subset of Unit Cube in $\mathbb{R}^3$ for Example 1.2

Example 1.2 Let B_0 be the unit cube in $\mathbb{R}^3$ and let S be the subset pictured in the left part of Figure 1.3. For $M = 3$, the tree produced by SUBDIVIDE which encodes this complex is also shown in the figure.

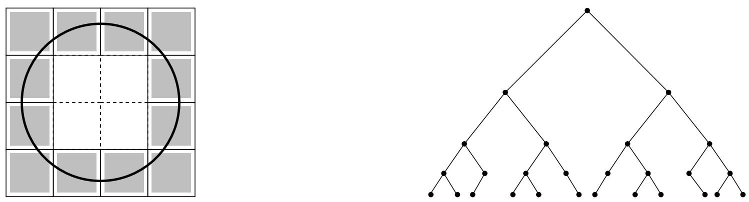
Example 1.3 Consider the subset of $\mathbb{R}^2$ given by

$$S^1 := \{(x_1, x_2) : x_1^2 + x_2^2 = 1\}.$$

We set

$$B_0 := \{(x_1, x_2) : -\frac{5}{4} \leq x_1 \leq \frac{5}{4}, -\frac{5}{4} \leq x_2 \leq \frac{5}{4}\}$$

which contains S^1 and perform the algorithm SUBDIVIDE with $M = 4$. Figure 1.4 indicates the final set of cells (shown shaded in the figure) which are retained and encoded in the binary tree shown.



(a) (b)

Figure 1.4: Unit circle cubical approximation with tree for Example 1.3

Remark: The method described in this section has been extensively investigated by Dellnitz, et. al. in [5], [6], [7] and [13]. It has been implemented in a computational program called GAIO (Global Analysis of Invariant Objects), which is a tool for numerically exploring dynamical systems by finding cubical approximations to unstable manifolds and finding invariant measures on these complexes. See Section 5.2 for more information about how we use GAIO to produce cubical complexes which serve as isolating neighborhoods.

2 Homology Algorithm

In this section we present an algorithm for computing the relative homology groups of cubical complexes of the type described in the previous section. For clarity of exposition we present the homology algorithm only, and the method for computing relative homology is a straightforward modification.

2.1 Data Structures and Storage

The algorithm described in this section operates on two fundamental data types: cells and blocks. A *cell* is described by its dimension, boundary list, coboundary list, and simplification status which will be denoted by its *simplify-flag*. The *boundary list* of a p-cell, c, is the list of the $(p-1)$-cells in ∂c. Similarly the *coboundary list* of a p-cell, c, is the list of $(p+1)$-cells $b \in \delta c$ together with their incidence numbers $\langle \partial b, c \rangle$. These incidence numbers are used to sort the coboundary lists as follows. All cells with invertible incidence number (i.e. ± 1 for computations over $\mathbb{Z}$) are placed at the head of the list, and these cells are then stored in a priority queue whose key is the length of the boundary list of the cell.

A *block* represents a node in the binary tree and is composed of lists of cells of each dimension $0, \ldots, d$. These lists are also sorted so that cells possessing coboundaries with invertible incidence numbers are at the head of the list and, additionally, are stored in a priority queue whose key is the length of the coboundary list of the cell. A block also contains information about the cells on its boundary, and we will use standard data structures for which list manipulation is optimized, i.e. data structures which search, insert, delete, and merge lists (or priority queues) in time proportional to the logarithm of the length of the list (c.f. [2]). Therefore, the global storage for the algorithm consists of a binary tree whose elements are blocks containing various lists of cells.

2.2 Procedures

All while–loops used below will step through a list, and the increment to the next element in the list will be omitted. Unless otherwise specified, we assume that we are performing all calculations in $\mathbb{Z}$.

• Create-Cube:
Input: position at the bottom level of the binary tree
Output: block containing a cubical d-cell and all its lower dimensional faces

1 Create vertices from position in binary tree
2 Create faces of dimension $1, \ldots, d-1$

• Concatenate-Blocks:
Input: four subblocks at level $l-1$

Output: one unsimplified block at level l

Compare block boundaries and identify common cells
Merge boundary and coboundary lists of common cells

Remark: The computational complexity of this procedure is determined by the number of operations to search and merge block boundary data and boundary and coboundary lists. We have omitted a detailed explanation of these data structures and assume standard algorithms are used; see Section 4.

• SIMPLIFY-BLOCK:
Input: block whose level in binary tree is a multiple of d
Output: block containing only cells which do not have incidence numbers ± 1 or which are on the block boundary; all cells are connected to the block boundary

Mark simplify-flag to be true for all vertices not on the block boundary
Mark simplify-flag to be true for all cells containing at least one marked vertex
for $p = 0$ to $p = d - 1$
  while block cell list of dimension p is non-empty **do**
    $b =$ next cell with simplify-flag $=$ true
    if $c \in \delta b$ with $\langle b, \partial c \rangle = \pm 1$ **then**
      Define $X := \delta b - \{c\}$
      Define $Y := \partial c - \{b\}$
      for each $x \in X$ replace b in ∂x by Y
      for each $y \in Y$ add X to δy with
        $\langle \partial x, y \rangle = \langle \partial x, y \rangle - \langle b, \partial x \rangle \langle b, \partial c \rangle \langle y, \partial c \rangle$ for all $x \in X$
      Delete b and c
while block cell list of dimension 0 is non-empty **do**
  if v is interior to block
    then delete v and δv from block cell lists

Remark: The cells removed at the end of SIMPLIFY-BLOCK represent a completely simplified connected component of the final complex. They are removed from the block, because they do not need to be considered in any further simplification steps.

• SIMPLIFY-BRANCH:
Input: node at current position in the tree
Output: block at the input node containing all cells from branches below the input node which remain after recursive simplification

if node at bottom level **then return**
SIMPLIFY-BRANCH to the left
SIMPLIFY-BRANCH to the right
if both branch nodes are non-empty
  then if current node level is a multiple of d

6   **then** CONCATENATE-BLOCKS
7   **else** Store subblocks from both branch nodes
8 **if** current node level is a multiple of d
9   **then** SIMPLIFY-BLOCK at this node

Remark: This procedure is most easily described and implemented recursively. However, in that case the number of operations depends on the number of levels in the tree as well as the number of cubes. This procedure can be implemented in a iterative way in which the extra cost above that of CONCATENATE-BLOCKS and SIMPLIFY-BLOCK does not depend on the number of levels; see Section 4.

• MAIN:
Input: list of all bottom level positions in the tree describing the complex
Output: simplified complex containing no cells with incidence number $= \pm 1$

1 CREATE-CUBE at left-most position in bottom level of the tree
2 **while** new positions at bottom level **do**
3   Compute common node between current position and previous one
4   SIMPLIFY-BRANCH to the left at common node
5   CREATE-CUBE at current node
6 SIMPLIFY-BLOCK at root node with all cells marked as simplifiable

2.3 Geometric interpretation

We now elaborate on the procedures described in Section 2.2 above, but the focus will be on SIMPLIFY-BLOCK and CONCATENATE-BLOCKS since these procedures require the most operations. We will also illustrate MAIN and SIMPLIFY-BRANCH, and all of the examples will be in two dimensions for clarity of exposition.

We need to define some of the terms used in SIMPLIFY-BLOCK. Since a block represents a node in the binary tree, it also describes a particular region in $\mathbb{R}^2$ and the geometric boundary of this region is a well-defined and easily determined set. Initially, every cell is realizable geometrically as a cube of some dimension and all the cells which are completely contained in this geometric boundary make up the *block boundary*. Any cell b which is not completely contained in this set will be called *interior* since it is in the geometric interior of the region represented by the node.

Any changes to the boundary or coboundary of such an interior cell will affect only other cells in the block. Thus, it is a candidate for removal by SIMPLIFY-BLOCK and will have its simplify-flag set to true. Such cells will also be referred to as *simplifiable*. Cells on the block boundary cannot be simplified because not all of their coboundary information is stored in one block. The condition in line 6 of SIMPLIFY-BLOCK checks for elements c of the coboundary of a simplifiable cell, b, for which the incidence number of b in ∂c is invertible in $\mathbb{Z}$. If there exists such a c, then Theorem 2 in [16] states that the homology of the new complex after removal of c and b is the same as the homology of the original complex, provided the incidence numbers are updated as given in line 11. Thus, each pass through lines 4-12 of SIMPLIFY-BLOCK preserves the homology.

We now give several examples for SIMPLIFY-BLOCK, beginning with two at the first simplification level. Suppose the input for the procedure is as shown in position 1 in Figure 2.1. The procedure first removes the central vertex from the data structures since it is the only simplifiable cell of dimension 0. The edge labeled c in position 1 is the coboundary element of the vertex which is also removed from the data structures in the sequence illustrated. Note that any of the other three edges in the vertex's coboundary could have been chosen. Position 2 shows the new complex with all affected cells updated. At this point we no longer have a cubical complex, but rather a CW-complex (see [18]).

At this point, there are no other simplifiable vertices, but there are two possible simplifiable edges to be removed: the central vertical one and the diagonal of the triangle in the upper right of position 2. The removal of these is shown in positions 3 and 4 in the figure, although the edges could have been removed in the opposite order. Position 4 is the final complex left when SIMPLIFY-BLOCK exits. Note that none of the edges on the block boundary were changed, and there is only one interior edge, but it has no coboundary so it cannot be simplified further.

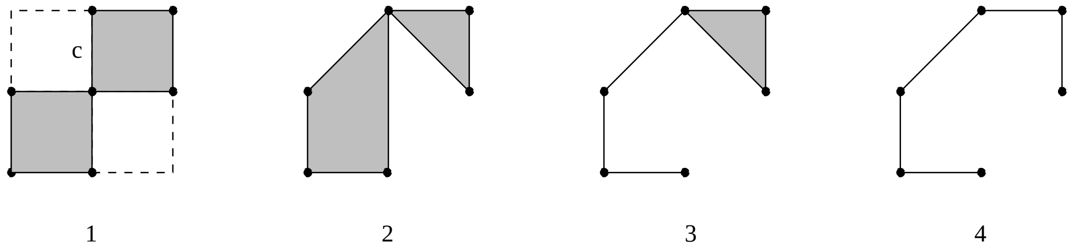


Figure 2.1: SIMPLIFY-BLOCK Example 1

As a second example, consider Figure 2.2 which illustrates the simplification of a collection of three squares as shown in position 1. As before, the central vertex is removed, together with one of its coboundary edges. Next, the edge labeled c is removed together with the triangular 2-cell in the upper left corner of the diagram in position 2 resulting in position 3. The other two edges are then removed and all coboundaries are updated to yield the complex in position 5. The order in which the edges are removed is arbitrary, although in Section 4.1 we address this issue in terms of the complexity analysis.

In order to provide a third example for SIMPLIFY-BLOCK, we will digress briefly and examine the output from CONCATENATE-BLOCKS. Figure 2.3 shows four (simplified) subblocks in part (a) being concatenated to yield the unsimplified block shown in part (b). We will use the complex in part (b) as the input to SIMPLIFY-BLOCK.

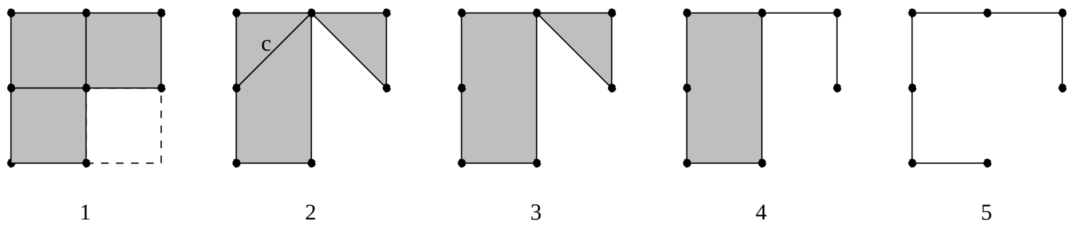


Figure 2.2: Simplify-Block Example 2

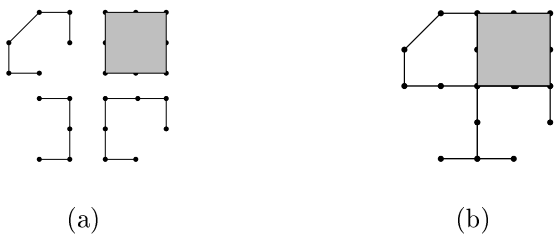


Figure 2.3: Concatenate-Blocks example

Eliminating some of the intermediate steps, we can obtain the sequence of simplifications shown in Figure 2.4. Position 1 is the initial complex and we see that there are now five vertices which are interior. After iteratively simplifying all of these, it is possible to obtain the complex shown in position 2 of Figure 2.4. Of course, since both the ordering of vertices in the simplification and the choice of coboundary elements to remove in each case can change, it is possible to obtain a homotopically equivalent (see below) complex which looks different from the one shown in the figure. After removing the interior edge and the 2-cell in its coboundary, we obtain the complex shown in position 3. There are two interior edges left, but they cannot be simplified further since they each have an empty coboundary.

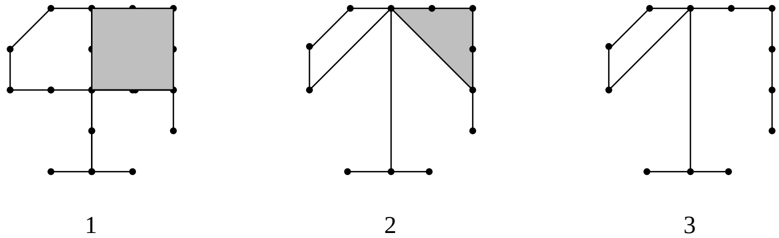


Figure 2.4: Simplify-Block Example 3

Main and Simplify-Branch are illustrated on the tree from Example 1.3 with $M = 4$ levels shown in Figure 1.4(b). Figure 2.5 shows the sequence of trees created by MAIN, where the numbers indicate the sequence in which the tree is created and then simplified. In particular, the dashed lines indicate which portions of the tree have not yet been created. In position 1, for example, MAIN creates the far left subtree and then SIMPLIFY-BRANCH simplifies it to the hollow circle shown in position 2. At this point SIMPLIFY-BLOCK removes all simplifiable cells and updates all the data structures. Positions 3-8 show the simplification of the remaining subtrees via SIMPLIFY-BRANCH. The hollow circles indicate that simplification takes place within that node using SIMPLIFY-BLOCK. Finally, the tree in position 8 reduces to the root node and the complex stored there is used as input to SIMPLIFY-BLOCK one last time. At this point every cell which is present is labeled as simplifiable so that on exit from MAIN the complex cannot be reduced further.

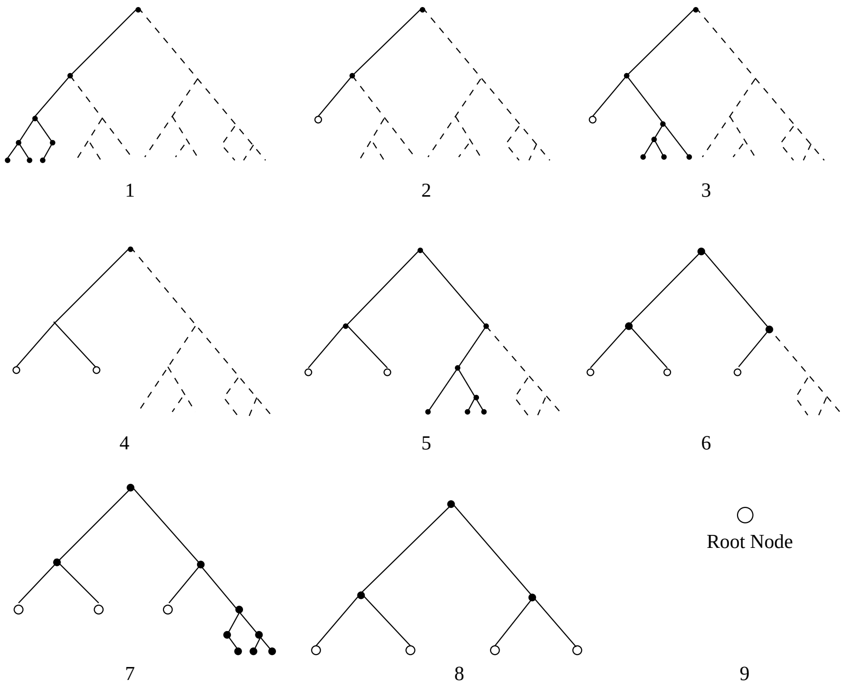


Figure 2.5: MAIN example

It is of fundamental importance to note that we never construct the entire tree, which implies that we never construct the entire complex. Since we remove only those cells which are in the geometric interior of a block, we are able to exploit the fact that homology is inherently a local computation.

As previously mentioned in the examples, throughout the SIMPLIFY-BLOCK procedure for the examples above we could get different complexes which are homotopically equivalent. In fact, each removal of a cell and one of its coboundary elements corresponds precisely to a simple homotopy. This implies directly that the homology is preserved by SIMPLIFY-BLOCK. Note also the only incidence numbers which appear are 0 and ± 1 (i.e., invertible), which means that the resulting complex on exiting from MAIN is minimal in the sense that it contains only generators of homology. In general, these facts are only true in two dimensions.

In higher dimensions there is no guarantee that only 0 and ± 1 appear as incidence numbers. Also, the removal of the cells cannot necessarily be viewed as a simple homotopy. Standard examples would be objects such as the k-fold dunce caps (c.f. [18]). It is important to note that Theorem 2 in [16] relies only on the algebraic consequences of the change in the incidence number given in line 11 of SIMPLIFY-BLOCK. Therefore, the homology is still preserved by the procedure although at the end we may not obtain a minimal complex as we do in two dimensions. We discuss the case of higher dimensions in Section 5.1.

3 Numerical Experiments

The algorithm described in the previous section has not been fully implemented, and code is currently under development. However, we have working code which implements a similar, but less efficient, strategy in $\mathbb{R}^3$. In this section we will briefly indicate some of these numerical results on some simple complexes. See Section 5.2 for a brief discussion an overall approach to analyzing flows using the Conley index by combining our homology code with another program (GAIO) which outputs cubical approximations to unstable sets.

Note that all data presented in this section was obtained by running the code on a Sun SPARCStation 20 with 160 megabytes of RAM. The output of the code is simply the number generators of each dimension which are left after the reduction algorithm ends and their boundary matrices if non-zero. For two examples, the tables below show the number of levels in the input tree, the number of cells in the complex (which is also the number of leaves in the tree), the run time in seconds, and finally a rough estimate for the order of growth based on the run times listed. The latter number is calculated as follows. If we assume that the growth is $O(n^p)$ for a complex with n cubes, then we take the run time for n cubes to be $t_n \approx Cn^p$. For a complex with $8n$ cubes, the time would be $t_{8n} \approx C(8n)^p$ for the same constant C. Then $p \approx \frac{\log(t_{8n}/t_n)}{\log(8)}$ is an estimate for the order of growth.

In Table 3 we give times for full trees in the sense that every leaf, and hence every possible cube, is present. In this case the output is 1 vertex (0-cell) and no cells of any other dimension. Thus,

$$H_n \approx \begin{cases} \mathbb{Z} & \text{for } n = 0 \\ 0 & \text{for } n \geq 1 \end{cases}$$

which is correct for this complex.

In Table 3 we give times for a complex which is homotopic to $S^1 \vee S^2 \vee T^2$,

Table 1: Run times and order of growth for full tree.

Number of Levels	Number of Cubes	Run Time	Order of Growth
9	512	3.5 secs	
12	4,096	67.0 secs	1.42
15	32,768	1,998.0 secs	1.63
18	262,144	78,269.0 secs	1.76

which has homology

$$H_n \approx \begin{cases} \mathbb{Z} & \text{for } n = 0 \\ \mathbb{Z}^5 & \text{for } n = 1 \\ \mathbb{Z}^2 & \text{for } n = 2 \\ 0 & \text{for } n \geq 3 \end{cases} .$$

The code yields a result of 1 vertex (0-cell), 5 edges (1-cells), 2 faces (2-cells), and 0 cubes (3-cells). We only give times for 12 and 15 level trees as examples, since the time for an 18 level tree is similar to that for the full tree shown in Table 3.

Table 2: Run times and order of growth for $S^1 \vee S^2 \vee T^2$.

Number of Levels	Number of Cubes	Run Time	Order of Growth
12	4,020	66.4 secs	
15	32,644	1,903.4 secs	1.61

4 Computational Complexity in 2D

In this section we will use the following notation. The *even* levels in the binary tree T will be indexed by $l = 0, 1, \ldots, L$ and will be called *simplification levels*. Since we start at the bottom of the tree and move up, $l = 0$ will denote the bottom of the tree and $l = L$ the root node. There are $2L + 1$ total levels in the binary tree, and the indexed levels are where simplification of blocks is performed. Also N_l will denote the number of nodes in the tree at level l. Hence $N = N_0$ is the total number of 2-cells in the initial complex.

4.1 Simplify-Block

We begin by estimating the total number of operations needed to perform Simplify-Block on all nodes in a dense tree. To accomplish this we need to estimate the sum of the number of operations to update boundary and coboundary information taken over all simplifiable cells at a given level.

The next two lemmas describe a simplified block at level $l \geq 1$ and the number of operations on the boundary and coboundary lists required for simplification. The first lemma determines the worst case estimates on the numbers of cells in each dimension remaining after simplification. Each simplification step inputs four simplified blocks at level $l-1$ and outputs one simplified block at level l. In the following discussion, we will always use B to denote a block (possibly unsimplified) at the current level l.

Lemma 4.1 *After one simplification step at level $l \geq 1$, the resulting block B has at most 2^{l+2} vertices on the boundary and no interior vertices, B has at most 2^{l+2} edges on the boundary, and one of the following statements holds:*
i) *B contains no other cells, $|\delta v \cap B| \leq 2$ for any vertex $v \in B$, and $|\delta e \cap B| = 0$ for any edge $e \in B$.*
ii) *B contains no interior edges and one 2-cell c with $|\partial c| = 2^{l+2}$. Moreover for any vertex $v \in B$ we have $|\delta v \cap B| = 2$, and for any edge $e \in B$ we have $|\delta e \cap B| = 1$.*
iii) *B contains no 2-cells and at most $2^{2l-1} - 1$ interior edges. Moreover for any vertex $v \in B$ we have $|\delta v \cap B| \leq 2^{2l-1} + 1$ and the total $\sum_v |\delta v \cap B| \leq 2^{l+2} + 2^{2l} - 2$. For any edge $e \in B$ we have $|\delta e \cap B| = 0$.*

The proof of Lemma 4.1 proceeds by finite induction on l and requires consideration of several special cases. By keeping a careful count on the number of cells left after Simplify-Block exits, as well as on the lengths of the boundary and coboundary lists, we can arrive at the results stated.

The following is the main lemma of the complexity result, so we will provide a sketch of the proof.

Lemma 4.2 *If T is a dense tree of N leaves, then to simplify T the total number of operations performed by* Simplify-Block *over all nodes is $O(N \log^2 N)$.*

Sketch of Proof: Let b be a simplifiable cell in B, and $c \in \delta b$ as in lines 5-6 of Simplify-Block. First we analyze the number of operations performed on the boundary and coboundary lists in lines 9-11. Let X_b and Y_b be the coboundary and boundary lists respectively which are defined in lines 7-8 of Simplify-Block. Since insertion into a list can be done in logarithmic time, in general the number of operations is estimated by $S(b) = K(|X_b| \log |Y_b| + |Y_b| \log |X_b|)$, where K is a constant which can change at each occurrence but is independent of l, the lengths of the lists, and the number of cells. However, in two dimensions if b is a vertex the estimate is $S(b) = K|X_b|$ and if b is an edge the estimate is $S(b) = K|Y_b|$, since the lengths of both the boundary list and coboundary list of an edge are at most two in $\mathbb{R}^2$.

Let $\mathcal{B}_l$ be the set of all blocks at level l, and let $\mathcal{C}_B^p$ be the set of all p-dimensional simplifiable cells in B. Note that $\mathcal{C}_B^p$ is contained in the central cross of B since the interiors of the subblocks are completely simplified. The total number of operations performed by SIMPLIFY-BLOCK in lines 3-11 over all blocks at all levels is

$$\sum_{l=1}^{L} \sum_{B \in \mathcal{B}_l} \sum_{p=0}^{1} \sum_{b \in \mathcal{C}_B^p} S(b) \tag{4.1}$$

It is important to notice that the terms in the first sum in (4.1) can change as each cell $b \in \mathcal{C}_B^p$ is simplified and hence the order of the cells in $\mathcal{C}_B^p$ is important. Thus we will implement the list of cells in $\mathcal{C}_B^p$ as a priority queue with $|X_b|$ as the priority when $p = 0$ and $|Y_b|$ as the priority when $p = 1$. Here the highest priority is given to the cells with the smallest boundary or coboundary lists. Inserting and extracting from a priority queue can be done logarithmically in the length of the queue.

Using the results of Lemma 4.1 we can then show that

$$\max \sum_{b \in \mathcal{C}_B^p} S(b) \leq K \cdot 2^{2l} \cdot l \tag{4.2}$$

where the maximum is taken over all possible unsimplified blocks B which can be obtained from the original complex. This can then be used to obtain the estimate

$$\sum_{l=1}^{L} \sum_{B \in \mathcal{B}_l} \sum_{p=0}^{1} \sum_{b \in \mathcal{C}_B^p} S(b) \leq K \cdot 2^{2L} \cdot L^2 \tag{4.3}$$

on the total number of operations. Since the tree is dense, $N \sim 2^{2L}$ and $\log N \sim L$. Therefore the number of operations is at most proportional to $N \log^2 N$ as desired.

$\square$

By again keeping a careful accounting on the number of cells stored at each node, we can obtain the following lemma.

Lemma 4.3 *To simplify a dense tree of N leaves, the total number of operations performed by* CONCATENATE-BLOCKS *over all nodes is $O(N \log^2 N)$.*

4.2 Overall Complexity

The procedures SIMPLIFY-BLOCK and CONCATENATE-BLOCKS, discussed in the previous subsection, are the most costly procedures and determine the computational complexity of the entire algorithm. In this section we will briefly discuss the other procedures described in Section 2 and give the overall complexity result.

CREATE-CUBE is of order $O(N)$ since it requires a constant number of operations for each cube. As remarked in Section 2, SIMPLIFY-BRANCH can be implemented iteratively so that a constant number of operations are performed at each branch node and not at every node as in the recursive version. Since the number of branch nodes is proportional to the number of leaves, no matter how many levels are in the tree, the cost is $O(N)$. We assume in this section that SIMPLIFY-BRANCH is implemented iteratively. We now proceed to analyze MAIN and state the overall computational complexity.

Recall that in Section 1.1 we defined a *dense tree* to be one in which the only nodes in the tree with exactly one child are at level $M-1$, where M is the depth of the tree. We now define a *dense subtree* T_d of the input binary tree T to be a connected subset which is itself a dense tree whose root and bottom levels are both at simplification levels in T. We define a *strut* in T to be a connected subset of T which has no branch points and which begins and ends at a simplification level of T. The *top* and *bottom* of the strut are the nodes closest to the root and leaves of T respectively. A *thinned subtree* T_t is a subtree which (a) has root and bottom levels at simplification levels of T; (b) is not dense; and (c) contains no struts.

Lemma 4.2 immediately applies to dense subtrees which occur at the bottom of the entire tree T. However, dense and thinned subtrees can occur anywhere in T, not just at the bottom. Note that the number of cells in a block at any node of T at any step in the simplification process is at most $K(d)N_s$ where N_s is the number of leaves at the bottom of T below the subtree. Combining this fact with Lemma 4.2 implies that the number of operations to simplify a dense subtree is $O(N_s \log^2(N_s))$ no matter where it is located in T. Also the number of leaves in a thinned subtree is proportional to 2^{2L} where L is the number of simplification levels in the thinned subtree, and hence Lemma 4.2 applies to thinned as well as dense subtrees.

Note that struts must connect two of the following: (a) a dense subtree; (b) a thinned subtree; (c) the root node of T; or (d) a leaf of T. With SIMPLIFY-BRANCH implemented iteratively, the cost of simplifying a strut is constant since we can assume that all simplification has already taken place within the root node of the subtree attached at the bottom of the strut. If there is a leaf at the bottom of the strut, then the number of cells (of any dimension) moved to the top of the strut will be constant and the subtree at the top of the strut can still be simplified with $O(N_s \log^2(N_s))$ operations. Thus, struts don't change the bounds on the number of operations performed by SIMPLIFY-BLOCK.

We now partition T into connected components which are either dense subtrees, struts, or thinned subtrees. We identify the components in the order mentioned and we choose each to be as deep as possible. The intersections between any two components will be a single node in the original tree. It is possible for one or more of the components to be empty so that, for example, there may not be any struts. Thus, we will view T as a collection of dense or thinned subtrees which are connected by struts and step 4 of MAIN will always be simplifying either a subtree or a strut.

The following technical lemma is straightforward to prove.

Lemma 4.4 *If $x_i \geq 1$ for $i = 1 \ldots P$, then*

$$\sum_{i=1}^{P} x_i \log^2(x_i) \leq \left(\sum_{i=1}^{P} x_i\right) \log^2 \left(\sum_{i=1}^{P} x_i\right) \tag{4.4}$$

The final complexity bound is obtained by estimating the number of operations required to simplify all of the dense and thinned subtrees in T as a sum of terms of the form $n \log^2 n$. Suppose T has N leaves (i.e., there are N cubes in the original complex). The number of dense or thinned subtrees in T which lie above a given leaf B is less than the number of simplification levels, not counting struts, which are above B. When the struts are removed from T, the number of simplification levels from the root node down to any leaf is at most $K \log N$. Combining this estimate with Lemma 4.4 yields the following theorem.

Theorem 4.5 *Let $\mathcal{C}$ be a cubical complex in $\mathbb{R}^2$ which contains N cubes and is stored in a binary tree as described in Section 1.1. The homology type $H_*(\mathcal{C})$ can be computed in $O(N \log^3(N))$ operations.*

5 Conclusion

The goal of this paper is to describe a reduction algorithm which can be used to compute the homology of a cubical cell complex, and the main application we have in mind is the numerical analysis of flows using the Conley index. This type of analysis has the greatest potential benefit in dimensions greater than three where visualization is difficult, and its practical implementation requires an efficient computation of homology in both time and space. Here we briefly comment on the important issues in the higher-dimensional case and the overall application to Conley index computations.

5.1 Higher Dimensions

There are several ways in which the two-dimensional version of the algorithm described above is a special case, but we conjecture that similiar subquadratic complexity bounds as in Theorem 4.5 are valid in the higher-dimensional case, as observed in some of the preliminary numerics in three dimensions shown in Section 3. As noted in Section 2.3, the incidence numbers produced in the two-dimensional algorithm can only be 0 or ± 1. This is due to the fact that the simplification operations on boundary and coboundary lists can be identified topologically as simple homotopies, and that complexes composed of rectangles in the plane are reduced to one-dimensional graphs, i.e. H_n is trivial for $n > 1$. Hence, the end result of the reduction algorithm is a chain complex containing only generators as in [16], and no other compuations are needed to obtain the homology type of the complex. It is well known of course that the homology type of such a complex can be computed in linear time, and our complexity estimate is superlinear (but subquadratic). However, in the above analysis, the additional simplification which arises from collapsing cells on the boundary of the complex is not taken into account. It is possible that a linear estimate could be obtained in this way; this is precisely the key to the linear estimate for the two-dimensional case in [16]. However, we are interested in the two-dimensional case only as a first step in the analysis for arbitrary dimension.

First we note that computations over $\mathbb{Z}$ can yield non-invertible incidence numbers other than 0, and that the simplification operations do not correspond to simple homotopies in all situations, see Section 2.3. Therefore, we cannot conclude that the end result of the computation is a fully reduced chain complex which is isomorphic to the homology groups unless computations are done in a field such as in $\mathbb{Z}_p$, as in Corollary 1 of [16]. Also the complexity bound obtained above is in general a bound on the number of operations on lists not bit operations. In the two-dimensional case or the case of coefficients in $\mathbb{Z}_p$, it is also a valid estimate of bit operations, but we cannot conclude this for the higher-dimensional case over $\mathbb{Z}$. However, it is hoped that in practice the resulting complex is sufficiently reduced that the usual matrix operations can be performed afterward without difficulty, and that the incidence numbers (and hence the number of bit operations) do not grow too large. We will defer further discussion of these issues to a later paper, and the reader is referred to the introduction for references to the literature.

5.2 Conley Index

The numerical analysis of flows generated by ODE's using the Conley index has basically three steps: (1) approximating the flow by multivalued map on a finite cellular complex, (2) identifying index pairs, and (3) homology computation. For the first step, our algorithm can accept as input the cubical complexes produced by GAIO, but there are other strategies, cf. Eidenschink [10]. The second step is done by generating a graph from the multivalued map whose nodes are the cells of the complex; the recurrent sets in the graph are candidates for isolating neighborhoods and can be computed easily using depth-first search algorithms, see Eidenschink [10]. These steps have been coded independent of dimension, but there is still much experimentation required to determine whether this scheme produces good index pairs in practice, especially in higher-dimensions. At this point, the homology computation is needed as a diagnostic tool as well. Since the entire process is far from automated, the homology computation must be efficient enough to allow for experimentation with several computations for the same problem. We are currently testing computations in three dimensions as well as developing a dimension independent code following the procedures outlined in this paper.

References

[1] T. J. Chou and G. E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Comput.*, 11:687–708, 1982.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1992.

[3] Cecil Jose A. Delfinado and Herbert Edelsbrunner. An incremental algorithm for Betti numbers of simplicial complexes. In *Proc. 9th Ann, Symp. Comput. Geom.*, pages 232–239, 1993.

[4] Cecil Jose A. Delfinado and Herbert Edelsbrunner. An incremental algorithm for Betti numbers of simplicial complexes on the 3-sphere. *Comp. Aided Geom. Design*, 12:771–784, 1995.

[5] Michael Dellnitz and Andreas Hohmann. The computation of unstable manifolds using subdivision and continuation. In H. W. Broer, S. A. van Gils, I. Hoveijn, and F. Takens, editors, *Nonlinear Dynamical Systems and Chaos*, PNLDE 19, pages 449–459. Birkhäuser, 1996.

[6] Michael Dellnitz and Andreas Hohmann. A subdivision algorithm for the computation of unstable manifolds and global attractors. *Numer. Math.*, 75:293–317, 1997.

[7] Michael Dellnitz, Andreas Hohmann, Oliver Junge, and Martin Rumpf. Exploring invariant sets and invariant measures. To appear in *Chaos. An Interdisciplinary Journal of Nonlinear Science*, 1997.

[8] Tamal K. Dey, Herbert Edelsbrunner, and Sumanta Guha. Computational topology. Preprint, 1997.

[9] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[10] Michael Eidenschink. *Exploring Global Dynamics: A Numerical Algorithm Based on the Conley Index Theory*. PhD thesis, Georgia Institute of Technology, 1995.

[11] Xin Gui Fang and George Havas. On the worst-case complexity of integer gaussian elimination. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation ISSAC 97*, pages 28–31. ACM Press, 1997.

[12] Joel Friedman. Computing Betti numbers via combinatorial Laplacians. Preprint, 1995.

[13] Rabbijah Guder, Michael Dellnitz, and Edwin Kreuzer. An adaptive method for the approximation of the generalized cell mapping. *Chaos, Solitons and Fractals*, 8(4):525–534, 1997.

[14] Costas S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix. *SIAM J. Comput.*, 18:658–669, 1989.

[15] Tomasz Kaczyński, Howard Karloff, Konstantin Mischaikow, and Marian Mrozek. Introduction to algebraic topology: A computational perspective. Book in progress.

[16] Tomasz Kaczyński, Marian Mrozek, and Maciej Ślusarek. Homology computation by reduction of chain complexes. Preprint, 1997.

[17] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM J. Comput.*, 8:499–507, 1979.

[18] James R. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[19] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[20] H. J. S. Smith. On systems of indeterminate equations and congruences. *Philos. Trans. Roy. Soc. London*, 151:295–326, 1861.

[21] Arne Storjohann. Near optimal algorithms for computing smith normal forms of integral matrices. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation ISSAC 96*, pages 267–274. ACM Press, 1996.