Application of Simplicial Sets to Computational Topology

Patrick Perry*

June 10, 2003

Abstract

This paper presents the concept of a Simplicial Set, which is a generalization of a Simplicial Complex, to the Computational Topology community. We show how to reduce the size of a simplicial set while preserving homotopy type, and also present an algorithm for computing the betti numbers of a simplicial set.

1 Introduction

Within the last ten years, there has been increasing interest in a new field of Computer Science called Computational Topology. The field involves taking ideas and constructions from Topology, and either finding efficient ways of computing them, or applying them to real-world problems As a field of computer science Topology is relatively young, but Mathematicians have been working on problems in Topology for hundreds of years. Thus, there is a wealth of knowledge to be gained by looking into the Mathematical literature and applying the work there to computational methods. In this paper we apply one such idea from the Mathematics, that of a Simplicial Set.

Currently, the standard way of representing a topological space in computational topology is as a simplicial complex. In a simplicial complex, we fix an ordering on our vertices, and define higher simplices as sets of vertices. The advantage of such a representation is that it is relatively simple, and thus very easy to work with. Unfortunately, the simplicial complex construction requires that each k-simplex contain *exactly* $(k+1)$ unique points, a restriction that turns out to be quite a hindrance. In a simplicial set, we do *not* have this restriction, and we are therefore allowed a great deal of flexibility that is absent in a simplicial complex.

In Section 2, we give the motivation for and introduce the concept of simplicial set. In Section 3 we show how to reduce the size of a simplicial set, thereby speeding up computations on the set—in particular betti-number computations.

*Department of Mathematics, Stanford University

Section 4 defines some useful data structures for dealing with a simplicial set, which we will rely upon for the rest of the paper. Section 5 gives a straightforward method of computing the homology group of any chain complex— this will be useful both as an algorithmic tool, and as a point of comparison. Lastly, Section 6 presents a new algorithm for computing the betti numbers of a simplicial set.

2 Simplicial Sets

The simplicial complex may be an adequate frameworks to work with when dealing with topological spaces arising from Euclidean objects, but it is too structured to admit such constructions as quotient spaces and identifications. We introduce the class of simplicial sets, which contains the class of simplicial complexes, and is flexible enough to allow constructions like those mentioned above.

2.1 Motivation

Two desirable features that are absent from the simplicial complex framework are quotients and identifications. We describe why such constructions may be desirable below.

2.1.1 Quotients of Spaces

We recall from Topology [6] that if X is a topological space and A is a subspace of X homotopic to a point, then X and the quotient space X/A are homotopy equivalent. For instance, in Figure 1, each edge of the triangle is homotopic to a point. We can quotient by two of the edges, one at a time, thereby simplifying the description of the space. This quotient operation will be known as a collapse, as it "collapses" a given subspace to a point while leaving the rest of the space unchanged.

After collapsing trivial simplices, we were able to reduce the representation of S^1 from 3 edges and 3 vertices to a representation needing only 1 simplex and 1 edge. Even though this is a small example, one can image that such collapses would be useful in dealing with large representations of topological spaces.

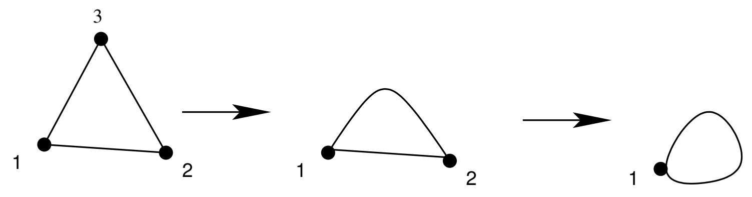

Figure 1: Simplifying a triangle through edge collapses

2.1.2 Identification Spaces

In an identification space, we take a topological space, and a disjoint partition of it, and create a new space whose points are the members of the partition [1]. For example, in the standard construction of the torus, we take a rectangle and identify opposite sides, preserving orientation.

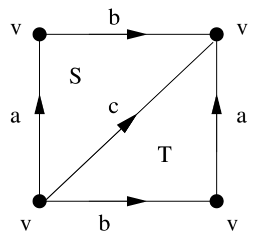

Figure 2: Triangulating a torus

To realize this construction in the context of a simplicial complex, we must triangulate a rectangle with its opposing sides identified. Unfortunately, the obvious first attempt, shown in Figure 2, is not a valid simplicial complex as both triangles contain only 1 unique vertex, which violates the rule that each n-simplex must contain $(n+1)$ vertices. In fact, a minimal legal triangulation of the torus requires at least 7 vertices in the simplicial complex setting [8].

In a simplicial set, however, n-simplices are allowed to have fewer than $(n+1)$ unique vertices. The situation in Figure 2 is completely valid. In general, by allowing such identifications, we are able to drastically reduce the size of a representation of a topological space.

2.2 Mathematical Framework

We now present the Mathematical basis for simplicial sets and their associated Homology groups. We also include the realization of quotients and identifications.

2.2.1 Formal Definition

The formal definition of a simplicial set, as given in [2] is as follows:

Definition 1 (Simplicial Set) *A Simplicial set K is a sequence of sets, $K = \{K_0, K_1, \ldots, K_n, \ldots\}$ together with functions:*

$$d_i : K_n \to K_{n-1},$$
$$s_i : K_n \to K_{n+1},$$

for each $0 \leq i \leq n$. These functions are required to satisfy the simplicial identities:

$$\begin{aligned}
d_i d_j &= d_{j-1} d_i && \textit{for } i \leq j, \\
d_i s_j &= \begin{cases} s_{j-1} d_i & \textit{for } i < j, \\ \textit{identity} & \textit{for } i = j,\ i = j+1, \\ s_j d_{i-1} & \textit{for } i > j+1, \end{cases} \\
s_i s_j &= s_{j+1} s_i && \textit{for } i \leq j.
\end{aligned}$$

d_i is called the ith face map, and s_i is called the ith degeneracy map.

Like in the simplicial complex setting, every n-simplex has $n+1$ faces. The difference is in the addition of so-called "degenerate" simplices, those simplices x which can be written $x = s_i y$ for some degeneracy map s_i.

2.2.2 The Simplicial Set Derived from a Simplicial Complex

Simplicial complexes fit very naturally into the simplicial set framework [2]. Let X be a simplicial complex and let the vertices of X be well-ordered. Then, we define K_0 to be the set of vertices of X, and for $n > 0$ we define

$$K_n = X_n \cup \left(\bigcup_{i=0}^{n} s_i(K_{n-1}) \right)$$

where X_n is the set of n-simplices of X. The term on the right is the set of degenerate simplices.

Now, the face map d_i acts on a simplex by deleting the ith vertex, and the degeneracy map s_i acts by repeating the ith vertex:

$$\begin{aligned}
d_i([v_0, \ldots, v_n]) &= [v_0, \ldots, \widehat{v_i}, \ldots, v_n], \\
s_i([v_0, \ldots, v_n]) &= [v_0, \ldots, v_i, v_i, \ldots, v_n].
\end{aligned}$$

It is easy to see that these maps satisfy the simplicial identities. Note that since X is a simplicial complex, all faces of a simplex are in the simplicial complex as well, so d_i is well-defined. The degeneracies of a simplex, however, are *not* in the simplicial complex, as they have repeated vertices.

2.2.3 The Homology of a Simplicial Set

We form the chain complex of a simplicial set K in a similar way to the chain complex of a simplicial complex. We define the n-chain group C_n to be the free group on the non-degenerate n-simplices of K. The boundary map is defined as

$$\partial_n = \sum_{i=0}^{n} d_i$$

where degenerate faces are treated as 0. Now, ∂ has the property that $\partial_n \partial_{n+1} = 0$, as desired. The n-th Homology group is defined in the usual way as $H_n = \ker \partial_n / \operatorname{im} \partial_{n+1}$.

Notice that the simplicial set derived from a simplicial complex has an identical chain complex, and thus the same Homology as the simplicial complex.

2.2.4 Identifications

The spaces arising from identifications easily fit into the context of simplicial sets. For instance, the torus in Figure 2 is described by the following simplicial set:

The non-degenerate simplices are v, a, b, c, S, and T. These simplices, together with their degeneracies define the sets K_i. Now, the face maps are defined as:

$$\begin{array}{ccccccc} d_0(S) & = & d_0(T) & = & a, & & \\ d_1(S) & = & d_1(T) & = & c, & & \\ d_2(S) & = & d_2(T) & = & b, & & \\ d_i(a) & = & d_i(b) & = & d_i(c) & = & v \end{array}$$

and the degeneracy maps are defined to satisfy the simplicial axioms. This is a completely valid representation of the torus, and it is much more compact than any torus in the simplicial complex setting.

2.2.5 Quotients

Quotient spaces may be constructed explicitly, but the real power of a quotient is to modify an *existing* topological space, as in Figure 1. In particular, we are interested in "collapses"—identifications of a simplex to a single point. Such an operation fits very naturally within the context of a simplicial set.

To construct a simplicial set K' from collapsing the simplex x in K, i.e. $K' = K/x$, we introduce a new vertex x_0, and replace all instances of x—its faces, and its degeneracies—with degeneracies of x_0. Specifically, we perform the following construction:

Say x is a simplex of dimension n (i.e. $x \in K_n$). For $k \geq 0$, define the *non-degenerate face set of dimension k* of x, $F_{x,k}$ inductively as:

$$F_{x,k} = \begin{cases} \emptyset & \text{for } k > n \\ \{x\} & \text{for } k = n \\ \bigcup_{i=0}^{k+1} d_i(F_{x,k+1}) & \text{for } k < n \end{cases}$$

Now, for $k > 0$ we define the *complete face set of dimension k* of x as:

$$\bar{F}_{x,k} = F_{x,k} \cup \left(\bigcup_{i=0}^{k-1} s_i(\bar{F}_{x,k-1}) \right)$$

for $k > 0$, and say that $\bar{F}_{x,0} = F_{x,0}$. With these sets, we are ready to define the simplicial sets and maps for K'. Define the simplex set K'_k as

$$K'_k = \left(K_k \setminus \bar{F}_{x,k}\right) \cup \{s_0^k(x_0)\}$$

where s_0^k denotes applying the 0-th degeneracy operator k times. s_0^k is not already in the simplex set K'_k, so we must add it explicitly. This is the only degeneracy of x_0 we need to add; it is easy to check via the simplicial relations that since x_0 is a vertex, applying k arbitrary degeneracy operators to x_0 is the same as applying s_0 to x_0 k times.

Notice that we are just removing all faces and degeneracies of x, and we are adding a single new simplex to each simplex set, the only non-degenerate addition being x_0.

Now, we define the simplicial maps as:

$$\begin{aligned} d'_i(y) &= \begin{cases} d_i(y) & \text{if } d_i(y) \notin \bar{F}_{x,i-1} \\ s_0^{i-1}(x_0) & \text{otherwise} \end{cases} \\ s'_i(y) &= \begin{cases} s_i(y) & \text{if } s_i(y) \notin \bar{F}_{x,i+1} \\ s_0^{i+1}(x_0) & \text{otherwise} \end{cases} \end{aligned}$$

Again, we are just replacing instances of x with x_0 or one of its degeneracies.

The collapses in Figure 1 are exactly the geometric realizations of the result of collapsing two edges in the simplicial set representation. Such collapses would not be possible in a simplicial complex.

2.3 Combinatorial Representation

We see that there is a very natural way to perform quotients in a simplicial set. With an efficient combinatorial representation for a simplicial set and a collapse, we will, indeed, have a powerful tool.

2.3.1 Design Goals

We would like a data structure to store a simplicial set supporting a BOUNDARY operation, as well as a COLLAPSE operation, which replaces the simplicial set with a new simplicial set obtained by identifying a simplex with a single point. In addition, it will be useful for some algorithms to support an UNCOLLAPSE operation that will "undo" a collapse. Of course, we would also like to be able to iterate through the simplices of each dimension.

If there are n total simplices in the set, it is reasonable to request that the memory usage be $\Theta(n)$. Also, we would like the BOUNDARY operation to take constant time, and the COLLAPSE and UNCOLLAPSE operations to take amortized constant time.

It will not be necessary to store all of the degenerate simplices in the simplicial set (after all, there are infinitely many of them). However, a degenerate simplex *will* be important if it came from collapsing a non-degenerate simplex. We would like to test if a simplex is degenerate or not via a HAS-BEEN-COLLAPSED method.

2.3.2 Data Storage

An n-simplex, x, will have pointers to each of its $n+1$ faces. These pointers will be labeled explicitly by the face operator d_i they come from. In practice, this can be accomplished by storing an array of $n+1$ pointers, where the ith element of the array is $d_i(x)$. The dimension of the simplex, n, will be stored in a *dimension* field. Also, each simplex will need to know if it is degenerate or not, and which collapse operation caused it to be degenerate. To achieve this, the simplex will have a *collapsedTo* field, initially set to NIL, which may be replaced by a pointer to simplex that collapses it.

As collapsing a simplex can cause two or more vertices to be identified, we will need a different way of representing 0-simplices. Following the lead of [3], we choose to store the them as a disjoint partition of identified points. We use the Disjoint Set data structure, which supports the MAKE-SET, UNION, and FIND-SET operations, all of which run in amortized $O(\alpha(m,n))$ time, where n is the number of MAKE-SET operation, m is the total number of operations, and α is the inverse Ackermann function [7]. Initially, each point will reside in a separate set, but through identifications, we will take the unions of these sets.

In addition to the local simplex information, the global simplicial set data structure will maintain separate arrays for the simplices of each dimension. Iterating through all n-simplices will be a simple matter—just iterate through the appropriate array.

2.3.3 Operations

Boundary Given a simplex x, computing it's boundary involves iterating through its face array and returning a chain with the appropriate coefficients (positive or negative 1):

```
BOUNDARY(x)
1  return $\sum_{i=0}^{dimension[x]}(-1)^i d_i(x)$
```

The explicit chain representation will be discussed below, but one can imagine two arrays sufficing: one for pointers to simplices, and one for coefficients. The run time is $\Theta(k)$, where k is the dimension of the simplex.

Has-Been-Collapsed Testing to see whether or not a simplex has been collapsed, which is the same as testing for degeneracy for non-point simplices, only involves checking the *collapsedTo* attribute:

```
HAS-BEEN-COLLAPSED(x)
1  if collapsedTo[x] = NIL
2      then return FALSE
3      else return TRUE
```

The run time is constant.

Collapse As we would like COLLAPSE to be "undoable" via the UNCOLLAPSE, we must keep track of which COLLAPSE operation causes a simplex to be collapsed. To this end, as described above, each simplex is equipped with a *collapsedTo* field. When a simplex is to be collapsed, we call the COLLAPSE-TO(x) method for each of it's faces. For a non-degenerate face, this private method sets the *collapsedTo* field to x and recursively calls COLLAPSE-TO(f, x) for each face, f. For a degenerate face, the COLLAPSE-TO method does nothing.

Handling points is different from other simplices, because when an edge connecting two points gets collapsed, the points must be marked as identified. This is done via the disjoint partition data structure that is stored with the simplicial set so identifying the points u and v only involves calling UNION(u, v).

The pseudocode for the collapsing functions is given here:

```
COLLAPSE-TO($f, x$)
1  $collapsedTo[f] \leftarrow x$
2  if $dimension[f] = 1$
3      then UNION($d_0(f)$, $d_1(f)$)
4      else for $i \leftarrow 0$ to $dimension[f]$
5          do if HAS-BEEN-COLLAPSED($d_i(f)$)
6                  then continue
7                  else COLLAPSE-TO($d_i(f), x$)
```

```
COLLAPSE($x$)
1  if HAS-BEEN-COLLAPSED($x$)
2      then return
3      else COLLAPSE-TO($x, x$)
```

Collapsing an entire simplicial set will cause COLLAPSE-TO(f, x) to get called exactly once for each simplex f of dimension greater than 0 in the set K. The time spent on this function call for 1-simplices is amortized $O(\alpha(n_1, n_0)$, where n_1 is the number of 1-simplices, n_0 is the number of 0-simplices, and α is the inverse Ackermann function. For other simplices, the run time is $\Theta(dimension[f])$. Thus, the amortized cost of each collapse operation when collapsing the entire simplicial set is $O(\max(k, \alpha(n_1, n_0)))$ (where k is the dimension of the simplicial set), which is essentially constant.

Uncollapse The inverse to the COLLAPSE(x) operation, UNCOLLAPSE(x) takes all faces of x which were collapsed as a result of a COLLAPSE(x) function call, and returns them to the prior state. The pseudocode for the function and its helper, UNCOLLAPSE-FROM(f, x) are given here:

```
UNCOLLAPSE-FROM($f, x$)
1  $collapsedTo[f] \leftarrow$ NIL
2  if $dimension[f] \neq 1$
3      then for $i \leftarrow 0$ to $dimension[f]$
3          do if $collapsedTo[d_i(f)] = x$
4              then UNCOLLAPSE-FROM($d_i(f), x$)
```

```
UNCOLLAPSE($x$)
1  if $collapsedTo[x] = x$
2      then UNCOLLAPSE-FROM($x, x$)
```

Notice that this is not a true inverse to the COLLAPSE function, as we are not undoing the point identifications. While such un-identifications may be desirable, the disjoint-set data structure does not support an "undo-union" function that runs in reasonable time.

The run-time analysis for uncollapsing a completely-collapsed simplicial set is essentially the same as that for collapsing the entire set, with the absence of the point-identification term. The amortized cost of a single UNCOLLAPSE when uncollapsing the entire simplicial set is thus $O(k)$ where k is the dimension of the simplicial set.

To allow for undoing point identifications, we provide the following function:

```
UPDATE-POINT-IDENTIFICATIONS()
1  for each $v \in K_0$
2      do MAKE-SET($v$)
3  for each $e \in K_1$
4      do if HAS-BEEN-COLLAPSED($e$)
5          then UNION($d_0(e), d_1(e)$)
```

Notice that the function throws away the previous point identification partition and builds another one from scratch. The run time for this function is thus $O(n_1\,\alpha(n_1, n_0))$, which is essentially linear, and therefore very expensive.

3 Simplifying Computation by Reducing Simplicial Set Size

As a first application of simplicial sets to computational topology, we show how how they can be applied to speed up an existing algorithm for computing an invariant of a simplicial complex. We assume we have an algorithm for computing the betti numbers $\{\beta_k\}$ of a chain complex $(\mathbf{C}_*, \partial_*)$. We also assume that the algorithm runs in $O(g(n))$ time, where n is the number of objects in the complex[1]. Now, say we are trying to compute an invariant using an algorithm that runs in time $\Omega(f(n))$. We show how to produce an algorithm that runs at least as fast as the existing algorithm, often with a substantial speedup.

3.1 Motivation

We know from topology [6] that if X is a cell complex and if A is a contractible subcomplex, then the quotient map $X \to X/A$ is a homotopy equivalence, and therefore X and X/A have the same homology.

[1]Such an algorithm is presented in Section 5

Now, in the context of the simplicial set, we know how to replace X with X/A. Also, to start with, every simplex of a simplicial complex has trivial homology. The idea, then, is to iteratively collapse simplices with trivial homology, thereby reducing the size n of the derived chain complex, and speeding up the homology computation.

As collapsing is a local operation, collapsing the simplex x affects the homology of the cofaces of the faces of x [2]. If only a subset of x's cofaces were affected (for instance, the cofaces of one dimension higher), then some intelligent bookkeeping might be able to keep track of which simplices are *a priori* trivial. Unfortunately, there is no obvious restriction, and as x and its faces have potentially exponentially many cofaces, this bookkeeping does not seem practical.

3.2 Algorithm

The algorithm is simple: look for s simplex with trivial homology, and then collapse it. We loop through all simplices a specified number of times, collapsing more simplices on each iteration.

```
SIMPLIFY-COMPLEX(K)
1  for $i \leftarrow 0$ to $MaxNumIterations$
2      do $noNewNonvialSimplices \leftarrow$ TRUE
3          for $x \in K$
4              do if $x$ has trivial homology
5                  then COLLAPSE($x$)
6                      $noNewNonvialSimplices \leftarrow$ FALSE
7          if $noNewNonvialSimplices =$ TRUE
8              then break
```

To test if a simplex, x is trivial, we use the existing homology algorithm, which will take $O(g(2^k))$ for a simplex of dimension k. Actually, since we are only interested in whether or not x has trivial has trivial homology and we are not actually computing the betti numbers of x, in practice we can improve on this worst-case bound. Testing triviality of 1- and 2-simplices can be done in constant time by counting the numbers of nontrivial faces. For example, any 2-simplex with 2 nontrivial edges and 1 nontrivial vertex must be nontrivial. The rest of the 1- and 2-simplices are shown in Figure 3.

As noted above, collapsing one simplex affects potentially exponentially other simplices. Thus, if we test a simplex once and find it to have nontrivial homology, collapsing another simplex later may make the original nontrivial simplex into a simplex with trivial homology. The question that arises,then is how many times are we to iterate through the simplices. The first answer may be to iterate until no more simplices can be collapsed. However, doing so may

[2] A coface is a simplex that has x as a face. More precisely, a simplex y is a coface of x if there is some sequence of face operators indexed by $i_1 \dots i_N$ such that $d_{i_N} d_{i_{N-1}} \cdots d_{i_1} y = x$.

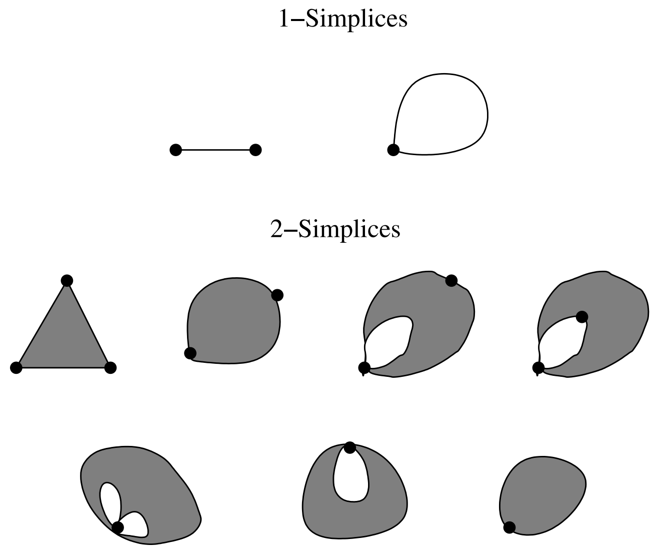

Figure 3: All possible 1- and 2-simplices in a simplicial set

take $O(n^2)$ iterations for certain configurations. For example, in Figure 4, the entire simplicial set has trivial homology. However, if we are iterating from right to left, we can only make 1 simplification per pass, and we must make n total passes. A better choice may be to limit the number of iterations to $cf(n)/g(2^k)$, where c is some constant and k is the dimension of the highest-dimensional simplex in K. By limiting our iterations as such, we are guaranteed that computing the betti numbers will still be $O(f(n))$.

3.3 Analysis

Unfortunately, there are no asymptotic guarantees as to how much we can simplify a simplicial set. However, empirical evidence suggests that this simplifications is a worthwhile endeavor. In the case of the triangulated torus in Figure 5, after two iterations the simplifying process terminates, reducing 18 faces, 27 edges, and 9 vertices to 10 faces, 11 edges, and 1 vertex, reducing the size of the complex by over 50 percent. With such a drastic reduction in such a small complex, it is expected that simplifying a large simplicial set will have similarly positive results.

4 Supporting Data Structures

In dealing with simplicial sets, it will be convenient to have some data structures to simplify operations like homology computation. As a bare minimum, we will

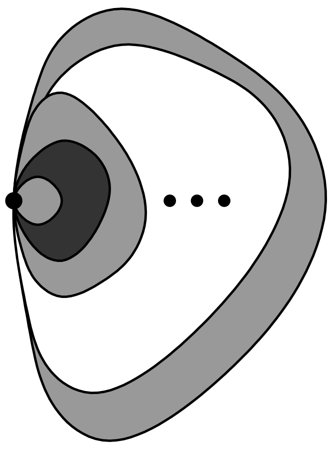

Figure 4: A worst-case situation for simplifying a simplicial set

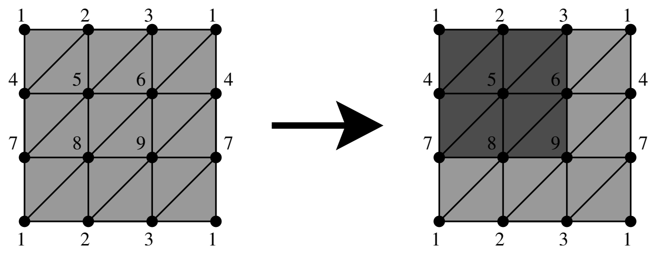

Figure 5: Simplifying a Torus (darker areas denote collapsed simplices)

need a structure for storing a chain on a simplicial set, as the BOUNDARY operation needs to return a chain. We suggest a simple structure for storing a chain that supports basic arithmetic operations, and then build more complicated structures built around the chain.

We recall that in defining homology, we first define the Chain Group, which is the free module with the simplices of our set as a basis. In the most general case, this construction is over a Euclidean Domain (ED). Typically, the integers ($\mathbb{Z}$) are used, but it is often more convenient to use a field—either the rational numbers ($\mathbb{Q}$), or the finite field with two elements ($\mathbb{F}_2$). In a field, in addition to the standard ring operations of addition, subtraction, and multiplication, we are also allowed to divide by an element. An ED does not allow division, but does allow us to use the Euclidean Algorithm to find the greatest common divisor (gcd) of two elements and write the gcd as a linear combination of the two elements [4].

In the data structures defined below, we focus primarily on the $\mathbb{F}_2$ case, but we will refer to the ring in the abstract as R.

4.1 The Chain Data Structure

The Chain is the fundamental object when studying homology. It is defined as a member of the Chain Group, the free R-module on the simplices. As a member of an R-module, a chain must support addition and subtraction of another chain, in addition to multiplication by scalars. It will also be convenient to support getting the coefficient of a given simplex, as will as a "leading term" construction, which is entirely artificial, but can be imposed by ordering the basis elements (the simplices). This ordering can be arbitrary, and order of creation or order by memory address are suitable choices.

4.1.1 Design Goals

We would like the storage for a chain to be linear in the number of basis elements. It is also reasonable to expect that arithmetic operations be linear in the number of basis elements, as well. Lastly, we expect that GET-LEADING-TERM be a constant-time operation.

4.1.2 Data

We store the chain as an ordered NIL-terminated singly-linked list [7]. *firstNode*[*c*] will be the first node in the linked list, and *nextNode*[*node*] will be the node pointed to by *node*.

Each node of the list represents a nonzero scalar from R and a pointer to a simplex. *coefficient*[*node*] will be the scalar from R, and *simplex*[*node*] will be the simplex.

We force each node to point to a unique simplex, and we also force that the list be ordered by the ordering induced from a simplex ordering. By storing our chain in this way, a chain involving m basis elements requires $2m$ pointers and m coefficients from R, so the total space is $O(m)$.

4.1.3 Operations

Leading Term The leading term of a chain will prove to be a useful for the structures we define below. We provide two methods, LEADING-COEFFICIENT, and LEADING-SIMPLEX for accessing the leading terms. Both are trivial operations, and we provide code only for LEADING-COEFFICIENT.

```
LEADING-COEFFICIENT(c)
1  return coefficient[firstNode[c]]
```

Simplex Coefficient We will often need the coefficient of a given simplex in the chain. If the simplex is not part of the chain, the coefficient is zero.

```
SIMPLEX-COEFFICIENT$(c, x)$
1  $node \leftarrow firstNode[c]$
2  while $node \neq$ NIL
3      do if $simplex[node] = x$
4             then return $coefficient[node]$
5             else $node \leftarrow nextNode[node]$
6  return 0
```

The run time is linear in the size of the chain. This operation could be made constant-time by using a hash table [7] instead of a linked list, but it would be at the expense of making the arithmetic operations more costly for sparse chains. As most chains are typically sparse, we choose to stick with the liked list by default.

Arithmetic The arithmetic operations are all relatively simple. We present the code for addition, and the reader can easily see how scalar multiplication and subtraction are performed. It is easy to see that all arithmetic operations can be performed in time linear in the length of the chain.

```
SUM$(a, b)$
1  $c \leftarrow$ LIST-COPY$(a)$
2  $p_c \leftarrow firstNode[c]$
3  $p_b \leftarrow firstNode[b]$
4  while $p_b \neq$ NIL
5      do while $p_c \neq$ NIL and $simplex[p_c] < simplex[p_b]$
6             do $p_c \leftarrow nextNode[p_c]$
7         if $simplex[p_c] = simplex[p_b]$
8             then $coefficient[p_c] \leftarrow coefficient[p_c] + coefficient[p_b]$
9                 if $coefficient[p_c] = 0$
10                    then LIST-REMOVE$(p_c)$
11            else LIST-INSERT-HERE$(p_c, p_b)$
12        $p_b \leftarrow nextNode[p_b]$
13 return $c$
```

4.2 The Boundary Cycle Data Structure

A Boundary Cycle is a special type of chain, an n-chain that is the boundary of some $(n+1)$-chain. We store such a cycle as a pair of chains, $(\beta, \partial_{n+1}(\beta))$. We allow access to the individual parts via *boundaryPart* and *boundingCycle* attributes. We also support arithmetic on boundary cycles by performing the operations on the individual components, as in:

$$(\alpha, \partial(\alpha)) + (\beta, \partial(\beta)) = (\alpha + \beta, \partial(\alpha) + \partial(\beta))$$

for addition. Since ∂ is an R-homomorphism, $\partial(\alpha) + \partial(\beta) = \partial(\alpha + \beta)$, and $r\partial(\alpha) = \partial(r\alpha)$, so these operation are valid.

4.3 The Homology Group Data Structure

We present a data structure for storing the homology group of a simplicial set. This data structure supports some sophisticated operations that a true homology group does not have, making our data structure more like the cycle group decomposition $Z_n = H_n \oplus B_n$. Nonetheless, the extra operations are quite useful in many situations involving homology groups.

4.3.1 Design Goals

In addition to knowing the rank of the homology group, we would like to be able to get a cycle's projection into the group. What's more, we ask that given a chain, $z \in C_n$ with boundary 0, we be able to write the chain as $z = \eta + \partial_{n+1}(b)$, where η is the projection of z into H_n, and $b \in C_{n+1}$ is an $(n+1)$-chain. Note that, as ∂_{n+1} is not necessarily injective, the choice of b will not, in general, be unique. However, since the cycle group decomposes as $Z_n = H_n \oplus B_n$, $\eta \in H_n$ and $\partial_{n+1}(b) \in B_n$ are uniquely determined by z.

The other basic operations required are iterating through a basis for the homology group, as well as iterating through the chains that form a basis for the boundary group, B_n.

4.3.2 Reduced Echelon Form

To perform our projection and decomposition operations, we will need to store bases for H_n and B_n in a convenient form. We call this form *Reduced Echelon Form*, and define it here:

Definition 2 (Reduced Echelon Form) *A set of chains $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\} \subset C_n$ is said to be in reduced-echelon form if:*

1. *As elements of the Chain R-module, $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\}$ are R-linearly independent, and*
2. *For each $\mathbf{c}_i$, if x_i is the leading simplex of $\mathbf{c}_i$, then for $j \neq i$, the coefficient of x_i in $\mathbf{c}_j$ is 0.*

Why such a form would be desirable is answered by the following lemma:

Lemma 3 (Gaussian Elimination) *If $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\} \subset C_n$ are in reduced echelon form, and $\mathbf{c}$ is in the span of $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\}$, then expressing $\mathbf{c}$ as an R-linear combination*

$$\mathbf{c} = \sum_{i=0}^{N} r_i \mathbf{c}_i \quad \text{where } r_i \in R$$

can be done with $O(N)$ divisions in R and $O(N)$ additions and scalar multiplications of chains from $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\} \cup \{\mathbf{c}\}$.

It is an easy result from linear algebra that given any set of chains, we can find a set of chains in reduced echelon form that spans the given set.

Proposition 4 *Given a set of chains $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\} \subset C_n$, we can find another set of chains $\{\mathbf{d}_1, \ldots, \mathbf{d}_M\} \subset C_n$ in reduced echelon form spanning our original set. Moreover, if R is a field, we can find such a set with $O(N^2)$ divisions in R and $O(N^2)$ additions and scalar multiplications of vectors from $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\}$. If R is an ED, we do through $O(N^2)$ instances of the Euclidean Algorithm, and $O(N^3)$ additions and scalar multiplications of vectors from $\{\mathbf{c}_1, \ldots, \mathbf{c}_N\}$.*

4.3.3 Data

To data for a homology group consists of a basis for H_n along with a basis for B_n, stored in two arrays, *homologyBasis* and *boundaryBasis*. The basis for H_n will be a set of chains, and the basis of B_n will be a set of cycles. We require that the set $\{H_n, B_n\}$ be in reduced echelon form.

We also build a balanced binary trees *leadingHCoefficients*, and *leadingZ-Coefficients* (Red-Black trees [7] are suitable candidates). The first tree stores the *homologyBasis* chains and the second tree stores both the *homologyBasis* chains and the *boundaryBasis* cycles, with an additional attribute in each node, *type*, indicating whether or not the given chain is a homology cycle or a boundary cycle. In both trees, the leading simplices of the chains as used as the search keys. Since the bases are in reduced echelon form, the search keys are unique. Building the trees takes $O(n \log n)$ time, and the SEARCH operations take $O(\log n)$.

4.3.4 Operations

Rank The rank of the homology group (otherwise known as the betti number) is just the number of H_n basis elements. It can be returned in constant time.

Projections Since the basis for the cycle group decomposition $Z_n = H_n \oplus B_n$ is in reduced echelon form, finding the representative element of a cycle can be done in time linear in the number of basis elements of B_n:

```
PROJECTION($H, z$)
1  projection ← 0
2  node ← firstNode[$z$]
3  while node ≠ NIL
4       do $h$ ← SEARCH(leadingHCoefficients[$H$], simplex[node])
5          if $h$ ≠ NIL
6             then projection ← (projection + $h$)
7          node ← nextNode[node]
8  return projection
```

This method requires $|z|$ SEARCH operations, and at most $|H_n|$ chain additions. If l is the longest homology basis chain length, m is the input chain length, and there are n homology basis vectors, then the total runtime is $O(m \log n + l\,m)$.

Cycle Decomposition If we know that a chain, z has boundary 0, we can query the homology group for its cycle decomposition $z = \eta + \partial b$, where $\eta \in H$ and $\beta = \partial b \in B$. The cycle decomposition is carried out as follows:

```
CYCLE-DECOMPOSITION($H, z$)
1  $h \leftarrow 0$
2  $(b, \beta) \leftarrow (0, 0)$
3  $node \leftarrow firstNode[z]$
4  while $node \neq$ NIL
5       do $\gamma \leftarrow$ SEARCH($leadingZCoefficients[H], simplex[node]$)
6           if $type[\gamma] =$ "homology"
7               then $h \leftarrow (h + \gamma)$
8               else $(b, \beta) \leftarrow ((b, \beta) + \gamma)$
9           $node \leftarrow nextNode[node]$
10 return $(h, (b, \beta))$
```

This procedure takes $|z|$ SEARCH operations, and $O(|H| + |B|)$ chain additions. Letting m be the length of z, and n be the size of Z, i.e. $n = |H| + |B|$, if l is the longest cycle basis element. Then the total runtime is $O(m \log n + m\,l)$. Note that since, in general, $|B|$ is much larger than $|H|$, the CYCLE-DECOMPOSITION function is considerably slower than the PROJECTION operation.

5 A Straightforward Approach to Computing Homology

In this section we present a method for computing the homology group of a simplicial set, K, using the data structure defined above. The algorithm is somewhat naïve in that it does not take advantage of the underlying structure in the boundary maps, but it serves both as a building block for more sophisticated algorithms, and as a point of comparison.

5.1 Problem Constraints

We are assigned with the task of computing $H_n = \ker \partial_n / \operatorname{im} \partial_{n+1}$. The constraints of our particular problem are as follows:

1. We must basis for H_n that is in reduced echelon form.
2. In addition to the basis for H_n, we must find a basis for $B_n = \operatorname{im} \partial_{n+1}$. The union of the bases $H_n \cup B_n$ must be in reduced echelon form.
3. For each basis chain $\beta \in B_n$, we must find a corresponding $b \in C_{n+1}$ such that $\partial_{n+1}(b) = \beta$.
4. Although we do not have an explicit matrix representation of $\partial_n : C_n \to C_{n-1}$, we have a list of all the n-simplices in K_n for each n, and for each simplex, $x \in K_n$ we can compute the chain $\partial_n(x)$ in time $O(n)$.

Note that as our boundaries are stored in Chain data structures, taking the transpose of ∂_n, or even performing row operations on the matrix ∂_n can be costly. We therefore limit ourselves to performing "column" operations only.

5.2 Solution

As we must compute both the kernel and the image of every boundary map, and both involve Gaussian Elimination, it would be ideal to perform Gaussian Elimination on each boundary matrix only once, computing the kernel and image simultaneously. We propose the following scheme:

Let l be the size of K_n, and let m be the size of K_{n-1}. Then, (∂_n) is an $m \times l$ matrix. To find the kernel and image, we augment (∂_n) vertically with the $l \times l$ identity matrix I_l as:

$$\left(\begin{array}{c} \partial_n \\ \hline I_l \end{array}\right)$$

Now, via column operations, we put the top half of the matrix in column-echelon form. We are left with the block-triangular matrix:

$$\left(\begin{array}{ccc|ccc} \beta_1 & \cdots & \beta_r & 0 & \cdots & 0 \\ \hline \mathbf{b}_1 & \cdots & \mathbf{b}_r & \zeta_1 & \cdots & \zeta_{l-r} \end{array}\right)$$

We have that $\{\beta_1, \ldots, \beta_r\}$ is a basis for $\operatorname{im} \partial_n = B_{n-1}$ with $\partial_n(\mathbf{b}_i) = \beta_i$. Also, we have that $\{\zeta_1, \ldots, \zeta_{l-r}\}$, span $\ker \partial_n = Z_n$, although the chains $\{\zeta_i\}$ are not necessarily linearly independent.

Now, to find a basis for $H_n = Z_n/B_n$, we take our basis for B_n, $\{\beta_1, \ldots, \beta_r\}$ from our column reduction of (∂_{n+1}), and we take our spanning set for Z_n, $\{\zeta_1, \ldots, \zeta_{l-r}\}$ from the column reduction of (∂_n). We put all the chains in a matrix:

$$\left(\begin{array}{ccc|ccc} \beta_1 & \cdots & \beta_r & \zeta_1 & \cdots & \zeta_s \end{array}\right)$$

and, using column operations, we put the matrix into reduced-column echelon form *without* modifying $\{\beta_1, \ldots, \beta_r\}$. We will have:

$$\left(\begin{array}{ccc|cccccc} \beta_1 & \cdots & \beta_r & \eta_1 & \cdots & \eta_t & 0 & \cdots & 0 \end{array}\right)$$

where $\{\eta_1, \ldots, \eta_t\}$ form are in reduced echelon form and form a basis for H_n.

At this point, we have all that is necessary to build our Homology Group data structure. Also, all of these computations can be implemented using the Cycle and Chain data structures.

5.3 Analysis

Column-reducing the first matrix (∂_n augmented with I_l) requires $O(l^2)$ column operations. Each column starts out with at most $(n+2)$ nonzero entries, but at the worst case, has $l+m$ nonzero entries. We thus estimate the first Gaussian Elimination stage to take time $O(l^2(l+m))$. That is, determining the image and kernel of ∂_n is approximately $O(|K_n|^3)$.

The next stage, computing $\ker \partial_n / \operatorname{im} \partial_{n+1}$ via the second Gaussian Elimination takes at most $O(\dim_R(Z_n) \dim_R(B_n))$ column operations. Since $B_n \subset Z_n \subset C_n$, and since each column has at most $|K_n|$ nonzero entries, a bound for this stage is $O(|K_n|^3)$.

Thus computing the n-th homology group, H_n takes $O(|K_n|^3)$ operations. Computing all homology groups of a simplicial set K takes $O(\sum_{i=0}^{N} |K_n|^3)$, or $O(|K|^3)$

In this analysis, we have not taken advantage of the sparse structure of the boundary map. It is important to note that more sophisticated methods may give betters bounds for these computations..

6 Computing Betti Numbers Using Simplicial Sets

We now present an algorithm for computing the betti numbers of a set that takes advantage of the quotient capabilities of the simplicial set. The algorithm relies heavily on the Long Exact Sequence, which we review below.

6.1 The Long Exact Sequence

It is a standard result from Topology that if X is a topological space, and $A \subset X$ is a subspace, then the homology groups of X, A, and X/A are very closely related. In particular, we have the following theorem as given in [6]:

Theorem 5 *If X is a space and A is a nonempty closed subspace that is a deformation retract of some neighborhood in X, then there is an exact sequence*

$$\cdots \longrightarrow \tilde{H}_n(A) \xrightarrow{i_*} \tilde{H}_n(X) \xrightarrow{j_*} \tilde{H}_n(X/A) \xrightarrow{\partial_n} \tilde{H}_{n-1}(A) \xrightarrow{i_*} \tilde{H}_{n-1}(X) \longrightarrow \cdots$$
$$\cdots \longrightarrow \tilde{H}_0(X/A) \longrightarrow 0$$

where i is the inclusion $A \hookrightarrow X$ and j is the quotient map $X \to X/A$.

In the statement of the theorem, i_* and j_* denote the induced maps on homology from the maps on topological spaces. $\tilde{H}_n$ denotes the *reduced* homology group.

The reduced homology group $\tilde{H}$ differs from the regular homology group in that the boundary map ∂_0 is defined to take the value $\emptyset$ for all points in X_0. This has the effect of making $H_n \cong \tilde{H}_n$ for $n > 0$ and $H_0 \cong \tilde{H}_0 \oplus \mathbb{Z}$.

An "exact" sequence is so-called because the kernel of one map is equal to the image of the map that precedes it. For example, $\ker j_* = \operatorname{im} i_*$.

The maps i_*, j_*, and ∂_n fit naturally within the context of our simplicial set data structures. As chains defined on a subcomplex of X point to the same simplices as the global chains in X, the inclusion map i_* is essentially trivial. j_* corresponds to replacing all simplices in A with degenerate ones, exactly what happens to a chain after our COLLAPSE operation. Lastly, ∂_n is the standard boundary construction.

The Long Exact Sequence provides a convenient way of seeing how a local homology computation relates the the global homology.

6.2 Outline of The Homology Computation Algorithm

Our algorithm for computing homology is essentially a direct implementation of the Long Exact Sequence. To compute the homology of the simplicial set X, we use the following algorithm:

```
Compute-Homology($X$)
1  $A \leftarrow$ Choose-Subcomplex($X$)
2  $\tilde{H}_*(A) \leftarrow$ Compute-Full-Homology-Group($A$)
3  Collapse($A$)
4  $\tilde{H}_*(X/A) \leftarrow$ Compute-Homology($X/A$)
5  Uncollapse($A$)
6  $\tilde{H}_*(X) \leftarrow$ Piece-Together($\tilde{H}_*(X/A), \tilde{H}_*(A)$)
7  return $\tilde{H}_*(X)$
```

First, we choose a subspace $A \subset X$. The exact details of the selection are given in Section 6.5. Next we compute the full homology group of A (including the boundaries) using an existing homology algorithm, such as the one presented in 5. Now, we collapse A and recursively compute a basis for the homology group of X/A. Lastly, we "piece together" the homology of X from the homology of X/A and A using the properties of exactness.

The "piecing together" works by observing that $\tilde{H}_n(X)$ splits as

$$\tilde{H}_n(X) \cong \ker \partial_n|_{\tilde{H}_n(X/A)} \oplus (H_n(A)/\operatorname{im} \partial_{n+1}|_{\tilde{H}_{n+1}(X/A)})$$

when our homology coefficients are from a field. This limits the algorithm to computing field homology only, but as the betti numbers are the same for integer and field homology, it is an acceptable limitation.

Calculating both of the summands is relatively easy using the structure we have already established. However, we must also find an explicit isomorphism to compute a basis for $\tilde{H}_n$. For this reason, we need to compute the full homology group of A, as described in section 4.3.

6.3 Piecing Together X from X/A and A

The real work of the algorithm is in the piecing together step. It will be useful to keep the following diagram in mind as we go over the details of how our procedure works.

$$\begin{tikzcd} \tilde{H}_n(X) \arrow[r, "j_*"] & \tilde{H}_n(X/A) \arrow[r, "\partial_n"] & \tilde{H}_{n-1}(A) \arrow[r, "i_*"] & \tilde{H}_{n-1}(X) \end{tikzcd}$$

The Piece-Together function uncollapses $A \subset X$, and computes the homology of X from the homology of X and X/A. The inputs to the function are $\tilde{H}_*(X/A)$, a basis for the set of homology groups of X/A, and $\tilde{H}_*(A)$, the complete set of homology groups of A. $\tilde{H}_*(A)$ is stored in our data structure described in 4.3 and $\tilde{H}_*(X/A)$ is just an array of Chains. We use collections $\ker \partial_n$ and $\operatorname{im} \partial_n$, which are collections supporting constant time insertion, and linear time to iterate through all elements (a linked list meets these requirements). d is assumed to be the dimension of the complex X. We also use the convention that $\tilde{H}_{-1} = \{\emptyset\}$.

```
PIECE-TOGETHER($\tilde{H}_*(X/A), \tilde{H}_*(A)$)
1  for $n \leftarrow 0$ to $d$
2       do $\ker \partial_n \leftarrow \emptyset$
3          $\operatorname{im} \partial_n \leftarrow \emptyset$
4          for each $\eta \in$ homologyBasis[$\tilde{H}_n(X/A)$]
5               do $(h, (b, \beta)) \leftarrow$ CYCLE-DECOMPOSITION($\tilde{H}_{n-1}(A), \partial_n(\eta)$)
6                  if $h = 0$
7                     then INSERT($\ker \partial_n, \eta - \beta$)
8                     else INSERT($\operatorname{im} \partial_n, h$)
9          $\tilde{H}_{n-1}(A)/\operatorname{im} \partial_n \leftarrow$ QUOTIENT(homologyBasis[$\tilde{H}_{n-1}(A)$], $\operatorname{im} \partial_n$)
10         $\tilde{H}_{n-1}(X) \leftarrow (\tilde{H}_{n-1}(A)/\operatorname{im} \partial_n) \cup \ker \partial_{n-1}$
11 return $\tilde{H}_*(X)$
```

Lines 4 and 5 compute the boundary of each basis cycle of $\tilde{H}_n(X/A)$, which is guaranteed to be in $\tilde{H}_{n-1}(A)$ by the long exact sequence. If the boundary is 0, then, the cycle is in $\ker \partial_n$, by definition. We lift the cycle $\eta \in X/A$ to a cycle in X by subtracting β from it. The boundary of $\eta - \beta$ in X is $b - b = 0$, so this is a valid lifting. When the boundary of η is not 0 in A, line 8 assigns $\partial_n(\eta)$ to $\operatorname{im} \partial_n$.

By exactness, $\operatorname{im} j_* \cong \ker \partial_{n-1}$. As the sequence is split, $\tilde{H}_{n-1}(A)/\operatorname{im} \partial_n = \ker j_*$. Thus, line 10 correctly computes a basis for $\tilde{H}_{n-1}(X)$ as $\ker j_* \cup \operatorname{im} j_*$.

6.4 Analysis

It is hard to perform a precise run-time analysis of the COMPUTE-HOMOLOGY algorithm, as it is mostly determined by the sizes of the homology groups of X/A after various collapses. These groups depend heavily on our choice of A, and are not easily given tight bounds on their sizes. We perform an exact analysis in terms of the sizes of the intermediate complexes X/A and their homology groups, and then in the next section discuss the choice of A and approximate conjectured run-time bases on this choice.

At any particular stage of the algorithm, we will have an A and an X/A. To distinguish them over the course of the algorithm, at the i-th level of the recursive algorithm we label them as A^i and $(X/A)^i$. We also let N be the number of collapses we are allowed to make before $(X/A)^i$ is a point set, which is the same as the number of levels deep the algorithm gets.

Over the course of the processing, we completely collapse, and then completely uncollapse X. The total cost of the N collapses is linear in the size of X.

We assume that line 1 of the COMPUTE-HOMOLOGY algorithm is a constant-time operation. Over the lifetime of the algorithm, this line takes time $O(N)$.

Line 2, computing the full homology groups of A^i, is cubic in the size of A^i. More precisely, the i-th instance of line 2 takes time $O(\sum_{n=0}^{d} |A^i_n|^3)$.

We now have that the run time T of COMPUTE-HOMOLOGY(X) is

$$\begin{aligned} T &= O(|X|) + O(N) + \sum_{i=1}^{N} O(\sum_{n=0}^{d} |A^i_n|^3) + \\ &\quad \sum_{i=1}^{N} \textsc{Time}(\textsc{Piece-Together}(\tilde{H}_*((X/A)^i), \tilde{H}_*(A^i))) \\ &= \sum_{i=1}^{N} \left[O(\sum_{n=0}^{d} |A^i_n|^3) + \textsc{Time}(\textsc{Piece-Together}(\tilde{H}_*((X/A)^i), \tilde{H}_*(A^i))) \right] \end{aligned}$$

The simplification is allowed since $N \leq |X|$ and, assuming there are no "lone points"—that is, each point is a face of at least one edge—we have that $|X| = \sum_i |A_i|$. We devote the rest of this discussion to analyzing the performance of PIECE-TOGETHER.

In each iteration of the main loop, other than constant-time collection insertion operations, the only work is computing the cycle decomposition of the boundary of each homology basis cycle in lines 4 and 5, and then computing the quotient $\tilde{H}_{n-1}(A^i)/\mathrm{im}\,\partial_n$.

Computing the decomposition of the n-cycle $\partial_{n+1}(\eta)$ takes time

$$O(|\eta|(\log |A^i_n| + |A^i_n|)) = O(|(X/A)^i_{n+1}|\,|A^i_n|)$$

Thus, over a full homology computation, this line takes time

$$\sum_{i=1}^{N} \sum_{n=0}^{d} |\tilde{H}_n((X/A)^i)|\, O(|(X/A)^i_{n+1}|\,|A^i_n|)$$

which, the reader will notice, is a degree 3 term. It will be crucial to choose our A^i such that this term does not get out of hand.

The quotient operation of line 9 can be carried out with $|\tilde{H}_{n-1}(A^i)|\,|\mathrm{im}\,\partial_n|$ chain additions. As each chain is of length $O(|A^i_{n-1}|)$, and as $\mathrm{im}\,\partial_n$ is of size $O(|\tilde{H}_n((X/A)^i)|)$, over the course of the algorithm line 9 takes time

$$\sum_{i=1}^{N} \sum_{n=0}^{d} O(|A^i_{n-1}|\,|\tilde{H}(A^i_{n-1})|\,|\tilde{H}_n((X/A)^i)|)$$

Again, this is a degree 3 term.

We now have that the run time of COMPUTE-HOMOLOGY(X) is

$$O(\sum_{i=1}^{N} \sum_{n=0}^{d} \Big[|A^i_n|^3 + |\tilde{H}_n((X/A)^i)| \, \Big(|(X/A)^i_{n+1}|\,|A^i_n| + |A^i_{n-1}|\,|\tilde{H}_{n-1}(A^i)| \Big) \Big])$$

Rearranging terms, we get see the run time to be

$$O(\sum_{i=1}^{N}\sum_{n=0}^{d}|A_n^i|\left[|A_n^i|^2+|\tilde{H}_n((X/A)^i)|\,|(X/A)_{n+1}^i|+|\tilde{H}_{n+1}((X/A)^i)|\,|\tilde{H}_n(A^i)|\right])$$

This is definitely an improvement over the straightforward computation, which takes time $O(\sum_{n=0}^{d}|X_n|^3)$. We now give discussion on how the A^i are to be chosen, and approximate run-times based on these choices.

6.5 Choice of Subcomplex

There are three obvious choices for the subcomplex $A \subset X$. In traditionally divide-and-conquer algorithms, we often divide our problem into two equal parts. Thus, the first candidate for the choice of A is a subcomplex of half the size of X. The second choice would attempt to minimize $|A|$. specifically, we can always choose an A of size 2 or 3 by choosing the lowest-dimensional simplex in the simplicial set. Lastly, we can choose the biggest possible simplex, the highest-dimensional one, thereby hoping to minimize $|\tilde{H}_n(X/A)|$.

Half of Complex We quickly see that even when ignoring issues of how to partition a complex into two equal parts, choosing A so that $|A| = \frac{1}{2}|X|$ is an improvement over Gaussian Elimination, but still leaves us with a cubic run-time. We can bound the run-time of the algorithm to be:

$$\begin{aligned} O(\sum_{i=1}^{N}\sum_{n=0}^{d}|A_n^i|^3) &= O(\sum_{i=1}^{N}\sum_{n=0}^{d}(\frac{|X_n|}{2^i})^3) \\ &= O(\sum_{n=0}^{d}|X_n|^3\sum_{i=1}^{N}(\frac{1}{8})^i) \\ &= O(\frac{1}{7}\sum_{n=0}^{d}|X_n|^3), \end{aligned}$$

which is asymptotically the same as Gaussian Elimination, but may be faster in practice.

Although we have not discussed the specifics of choosing half of the complex, one can imagine a greedy partitioning algorithm giving a choice of A to be roughly half of X, and adding a linear term to the run-time.

Smallest Simplex The other extreme from choosing A as half of the complex is choosing A to be the smallest simplex in X. First, this will mean choosing edges, which will have size 2 or 3 depending on how many unique endpoints there are. After all edges are gone, we proceed to choosing 2-simplexes, which will be of size 2, and then 3-simplexes, and so on until we exhaust the highest-dimensional simplexes. In this form, the algorithm is similar to the one presented in [3], but there are some significant differences.

As we collapse each simplex individually, N in this case is equal to $|X|$. Computing the cycle group decomposition of A will now be bounded by $O(3^2) = O(1)$. Likewise, if A is a k-simplex, computing the kernel of the boundary map only requires searching for those cycles in $\tilde{H}_{k+1}(X/A)$ which do not have A in their boundary. So far, we have been using linked lists to represent chains, but here, if we switch to hash tables for the homology basis cycles, this step can be performed in constant time. Consequently, at each stage computing $\ker \partial_n$ will either take time $O(\tilde{H}_n(X/A))$ or it will not be necessary. The last remaining piece, computing $\tilde{H}_n(A)/\operatorname{im} \partial_{n+1}$, is also trivial and can be done in constant time. The total run-time is now seen to be

$$O(\sum_{i=1}^{|X|} \sum_{n=0}^{d} |\tilde{H}_n((X/A)^i)|)$$

.

Intuitively, choosing A^i in this way will cause $\tilde{H}_n((X/A)^i)$ to get very large. The highest $\tilde{H}_n((X/A)^i)$ can possibly get is $O(|X_n|)$. Choosing A in this way gives, like the way describe above, an approximately quadratic run-time.

Largest Simplex The last remaining method for choosing our subcomplexes, A^i is to choose the highest-dimensional simplex in X. This will result in larger cycle decomposition and quotient computation times, but will, intuitively, keep $\tilde{H}_*((X/A)^i)$ to be a much smaller quantity.

Now, A^i has at most $\binom{d}{k}$ k-simplices, and $|A^i|$ is of size at most 2^d. Bounding $|(X/A)^i_n|$ by $|X_n|$ and $\tilde{H}_n(A)$ by $\binom{d}{n}$, we see the run-time to be

$$O(\sum_{n=0}^{d} |X_n| \operatorname{E}\left[\binom{d}{n}^2 + |\tilde{H}_n((X/A)^i)|\,|X_{n+1}| + |\tilde{H}_{n+1}((X/A)^i)|\binom{d}{n}\right])$$

where E denotes expectation over $i = 1, \ldots, N$. This sum is seen to be an

$$\Omega(\sum_{n=0}^{d} |X_n|^2 \operatorname{E}\left[|\tilde{H}_n((X/A)^i)|\right])$$

quantity, and is thus worse than the case of choosing A^i to be the smallest simplex.

7 Conclusions

We have introduced the simplicial set construction in a computational setting, and shown how the flexibility the construction provides can be useful in the context of Computational Topology. In particular, we have shown how to reduce the size of a simplicial set without affecting its homological type. We have also presented an algorithm for computing the betti numbers of a simplicial set that runs in approximately cubic time for dimensions higher than 3, which is an improvement over current methods.

It is still necessary to implement the algorithms described above to test them in a practical context, especially since some of the results above have asymptotically the same run times as existing algorithms. The performance of these algorithms depend heavily on constant factors, which have yet to be determined.

The notion of topological persistence presented in [5] has yet to be applied to the homology computation algorithm presented above. A direct translation of the idea of topological persistence may not be possible, but we believe a similar notion may be applicable.

Acknowledgments This thesis is the culmination of a year of weekly discussions with my advisor, Professor Gunnar Carlsson. I'd like to thank Professor Carlsson for introducing me to the field of Computational Topology in the first place, and also for patiently teaching me the requisite mathematical foundations. This last year has been a struggle, trying to fit my often naïve intuitions into a formal framework, but Professor Carlsson has always been patient with me, and has always kept me on the right path.

I'd also like to thank Afra Zomorodian and Vin de Silva for their early instruction and encouragement. Afra introduced me to the Computer Science end of Computational Topology, and Vin has always been extremely helpful when I've needed guidance or clarification.

References

[1] M.A. Armstrong, *Basic topology*, Springer-Verlag, New York, 1983.

[2] Edward B. Curtis, *Simplicial homotopy theory*, Advances in Mathematics **6** (1971), 107–209.

[3] C. J. A. Delfinado and H. Edelsbrunner, *An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere*, Comput. Aided Geom. Design **12** (1995), 771–784.

[4] D. Dummit and R. Foote, *Abstract algebra*, 2 ed., John Wiley and Sons, Inc., New York, 1999.

[5] H Edelsbrunner, D. Letscher, and A. Zomorodian, *Topological persistence and simplification*, 41st Symposium on Foundations of Computer Science (Redondo Beach, CA), 2001.

[6] Allen Hatcher, *Algebraic topology*, Cambridge University Press, Cambridge, England, 2002.

[7] Thomas H.Corman, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The MIT Press, Cambridge, MA, 1990.

[8] M. Jungerman and G. Ringel, *Minimal triangulations on orientable surfaces*, Acta Math. **145** (1980), 121–154.