The Implementation of the Colored Abstract Simplicial Complex and its Application to Mesh Generation

CHRISTOPHER T. LEE*, JOHN B. MOODY*, ROMMIE E. AMARO, J. ANDREW MCCAMMON, and MICHAEL J. HOLST, University of California, San Diego, USA

We introduce CASC: a new, modern, and header-only C++ library which provides a data structure to represent arbitrary dimension abstract simplicial complexes (ASC) with user-defined classes stored directly on the simplices at each dimension. This is accomplished by using the latest C++ language features including variadic template parameters introduced in C++11 and automatic function return type deduction from C++14. Effectively CASC decouples the representation of the topology from the interactions of user data. We present the innovations and design principles of the data structure and related algorithms. This includes a meta-data aware decimation algorithm which is general for collapsing simplices of any dimension. We also present an example application of this library to represent an orientable surface mesh.

CCS Concepts: • **Mathematics of computing** → **Mathematical software**; **Mesh generation**; Combinatorics; • **Theory of computation** → **Data structures design and analysis**; *Computational geometry*; • **Applied computing** → Imaging; Molecular structural biology; • **Computing methodologies** → *Computer graphics*;

Additional Key Words and Phrases: Abstract Simplicial Complexes, Molecular Modeling, Mesh Generation, Mesh Decimation, Variadic Templates, C++ Library

ACM Reference Format:
Christopher T. Lee, John B. Moody, Rommie E. Amaro, J. Andrew McCammon, and Michael J. Holst. 2017. The Implementation of the Colored Abstract Simplicial Complex and its Application to Mesh Generation. *Journal* 0, 0, Article 0 (2017), 21 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

For problems in computational topology and geometry, it is often beneficial to use simple building blocks to represent complicated shapes. A popular block is the simplex, or the generalization of a triangle in any dimension. Due to the ease of manipulation and the coplanar property of triangles, triangulations have become commonplace in fields such as geometric modeling and visualization

*Contributed equally to this work.

This work is supported by the National Institutes of Health under grant numbers P41-GM103426 and RO1-GM31749. CTL also acknowledges support from the NIH Molecular Biophysics Training Grant T32-GM008326. MJH was supported in part by the National Science Foundation under awards DMS-CM1620366 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1620366) and DMS-FRG1262982 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1262982).
Author's addresses: C.T. Lee, R.E. Amaro, and J.A. McCammon, Department of Chemistry, University of California San Diego; M.J. Holst, Department of Mathematics, University of California San Diego; J.B. Moody (Current address) ViaSat, Inc. Carlsbad-Bldg 10-2063, 6155 El Camino Real, Carlsbad, CA 92009 .
Authors' address: Christopher T. Lee; John B. Moody; Rommie E. Amaro; J. Andrew McCammon; Michael J. Holst, mholst@ucsd.edu, University of California, San Diego, La Jolla, CA, 92037, USA.

0098-3500/2017/0-ART0 $15.00
https://doi.org/0000001.0000001

as well as topological analysis. The use of meshes has become increasingly popular even in the fields of computational biology and medicine[12].

As methods in structural biology improve and new datasets become available, there is interest in integrating experimental and structural data to build new predictive computer models [8]. A key barrier that modelers face is the generation of multi-scale, computable, geometric models from noisy datasets such as those from electron tomography (ET)[11]. This is typically achieved in at least two steps: (1) segmentation of relevant features, and (2) approximation of the geometry using meshes. Subsequently, numerical techniques such as Finite Elements Modeling or Monte Carlo can be used to investigate the transport and localization of molecules of interest.

While many have studied mesh generation in the fields of engineering and animation, few methods are suitable for biological datasets. This is largely due to noise introduced by limits in image resolution or contrast. Even while using state-of-the-art segmentation algorithms for ET datasets there are often unresolved or missed features. Due to these issues, the generated meshes often have holes and other non-manifolds which must be resolved prior to mathematical modeling. Another challenge is the interpretation of a voxel valued segmentation. The conversion of zig-zag boundaries into a mesh can lead to other problems such as extremely high aspect ratio triangles or, in general, poorly conditioned elements[11]. To remedy this, various smoothing and decimation algorithms must also be applied prior to simulation.

Previous work by us and others have introduced a meshing tool for biological models, `GAMer`, for building 3D tetrahedral meshes which obey internal and external constraints, such as matching embedding and/or enclosing molecular surfaces. It also provides the ability to use various mesh improvement algorithms for volume and surface meshes[6, 10]. `GAMer` uses the `Tetgen` library as the primary tetrahedral volume generator[9]. While the algorithms are sound, the specific implementation is prone to segmentation faults even for simple meshes. Careful analysis of the code has identified that the data structures used for the representation of the mesh is primarily at fault. This article will focus entirely on the representation of topology in very complex mesh generation codes. We note that the algorithms which handle geometric issues like shape regularity and local adaptivity are well understood[1, 7], among others. Similarly there is a large body of literature related to local mesh refinement and decimation[2, 3]. Our innovations serve to enable the implementation of these algorithms in the most general and robust way.

`GAMer` currently employs a neighbor list data structure which tracks the adjacency and orientation of simplices. Neighbor lists are quick to construct, however the representation of non-manifolds often leads to code instability. Algorithms must check for aberrant cases creating substantial overhead. Other data structures to represent simplicial complexes have also been developed. A comprehensive list is beyond the scope of this work, instead we refer the readers to a recent review of available simplicial complex data structures[5].

The data structures can be categorized by into two major types: those which represent meshes of a specific dimension (i.e., surface or volume meshes), or those which can represent meshes of any dimension. Of the former, many of these data structures further make assumptions about the embedding or information stored and thus, are neither general nor scalable in dimension. For the latter, the data structures typically store only the combinatorics. Information about simplex properties are often stored elsewhere and accessed by dictionary. While this feature may be convenient for applications such as graphical visualization where similar materials may be applied to several faces, for applications with heterogeneous and dense stored data the map introduces an additional look-up step. Furthermore, performing domain decomposition for parallel processing on such a decoupled data structure may be difficult.

To the best of our knowledge, there currently exists no data structure which supports meshes of arbitrary dimension with user-selected typed data stored directly on each simplex. This level of generalization and abstraction would be necessary for the implementation of algorithms to compute triangulations of manifolds with arbitrary topology[4]. For this reason we have developed a new simplicial mesh data structure capable of both criteria.

Here we describe the development of a scalable *colored abstract simplicial complex* data structure called CASC. Simplices are stored as nodes on a Hasse diagram. For ease of traversal all adjacency is stored at the node level. An additional data object can be stored at each node which is typed according to the simplex dimension at compile time. This means that, for example, for a mesh the 0-simplices can be assigned a vertex type while the 2-simplices can store some material property instead. Typing of each k-simplex is achieved using variadic templates introduced in C++11. CASC thus provides a natural separation between the combinatorics represented by the ASC from the underlying data types at each simplex dimension and their interactions. In §2 we briefly define an ASC and some relevant definitions followed by the introduction of the CASC data structure and it's construction in §3. We then demonstrate the use of CASC to represent a surface mesh and compute vertex tangents in §5.

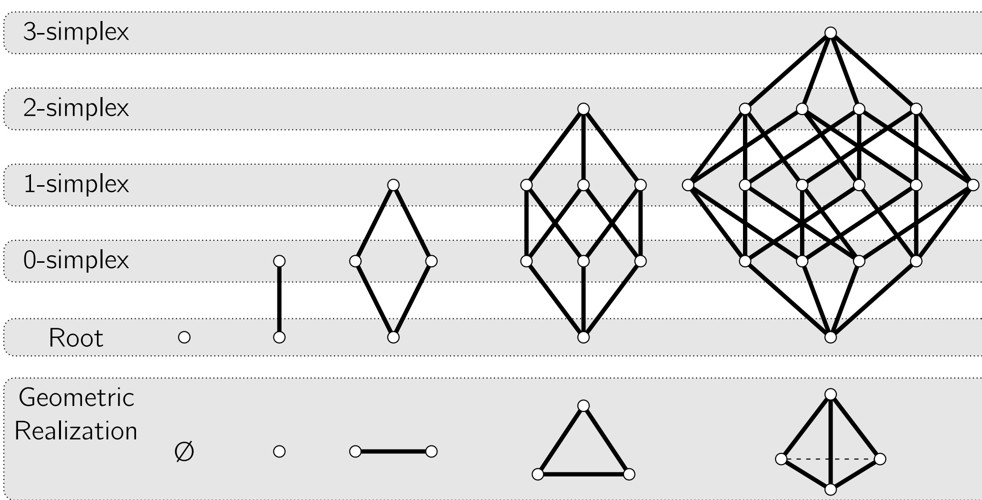


Fig. 1. Hasse diagrams of several ASCs and a geometric realization from left to right: the empty set, a vertex, an edge, a triangle, a tetrahedron.

2 BACKGROUND – ABSTRACT SIMPLICIAL COMPLEXES

An abstract simplicial complex (ASC) is a combinatorial structure which can be used to represent the connectivity of a simplicial mesh, independent of any geometric information. More formally, the definition of an ASC is as follows.

Definition 2.1. Given a vertex set V, an *abstract simplicial complex* $\mathcal{F}$ of V is a set of subsets of V with the following property: for every set $X \in \mathcal{F}$, every subset $Y \subset X$ is also a member of $\mathcal{F}$.

The sets $s \in \mathcal{F}$ are called a simplex or face of $\mathcal{F}$; similarly a face X is said to be a face of simplex s if $X \subset s$. Since X is a face of s, s is a coface of X. Each simplex has a dimension characterized by $\dim s = |s| - 1$, where $|s|$ is the cardinality of set s. A simplex of $\dim s = k$ is also called a k-simplex.

The dimension of the complex, $\dim(\mathcal{F})$, is defined by the largest dimension of any member face. Simplices of the largest dimension, $\dim(\mathcal{F})$ are referred to as the facets of the complex.

If one simplex is a face of another, they are incident. Every face of a k-simplex s with dimension $(k-1)$ is called a boundary face while each incident face with dimension $(k+1)$ is a coboundary face. Two k-simplices, f and s are considered adjacent if they share a common boundary face, or coboundary face. The boundary of simplex s, ∂s, is the sum of the boundary faces.

Having introduced the concept of an ASC, we can also define several operations useful when dealing with ASCs. A subcomplex is a subset that is a simplicial complex itself. The closure of a simplex, f, or some set of simplices $F \subseteq \mathcal{F}$ is the smallest simplicial subcomplex of $\mathcal{F}$ that contains F:

$$\mathrm{Cl}(f) = \{s \in \mathcal{F} \mid s \subseteq f\}; \qquad \mathrm{Cl}(F) = \bigcup_{f \in F} \mathrm{Cl}(f). \tag{1}$$

It is often useful to consider the local neighborhood of a simplex. The star of a simplex f is the set of all simplices that contain f:

$$\mathrm{St}(f) = \{s \in \mathcal{F} \mid f \subseteq s\}; \qquad \mathrm{St}(F) = \bigcup_{f \in F} \mathrm{St}(f). \tag{2}$$

The link of f consists of all faces of simplices in the closed star of f that do not intersect f:

$$\mathrm{Lk}(f) = \{s \in \mathrm{Cl} \circ \mathrm{St}(f) \mid s \cap f = \emptyset\} = \mathrm{Cl} \circ \mathrm{St}(f) - \mathrm{St} \circ \mathrm{Cl}(f). \tag{3}$$

For some algorithms, it is often useful to iterate over the set of all vertices or edges etc. We use the following notation for the horizontal "level" of an abstract simplicial complex.

$$\mathrm{Lvl}_k(\mathcal{F}) = \left\{s \in \mathcal{F} \mid \dim s = k\right\} \tag{4}$$

A subcomplex which contains all simplices $s \in \mathcal{F}$ where $\dim(s) \leq k$ is the k-skeleton of $\mathcal{F}$:

$$\mathcal{F}_k = \mathrm{Cl} \circ \mathrm{Lvl}_k(\mathcal{F}) = \bigcup_{i \leq k} \mathrm{Lvl}_i(\mathcal{F}). \tag{5}$$

By Definition 2.1, an ASC forms a partially ordered set, or poset. Posets are frequently represented by a Hasse diagram where sets are nodes and edges denote set inclusion. Several example simplicial complexes and their corresponding Hasse diagrams are shown in Fig. 1. Colloquially we will use *up* and *down* to refer to the boundary and coboundary of a simplex respectively. In Hasse diagrams, we follow a convention that simplices shown graphically on the same horizontal level have the same simplex dimension. Furthermore, simplices of greater dimension are drawn above lesser simplices.

3 COLORED ABSTRACT SIMPLICIAL COMPLEX

In this section we introduce the CASC data structure and its implementation. For a given simplicial complex, each simplex is represented by a node (`asc_Node`) in the Hasse diagram, and defined by a set of keys corresponding to the vertex(ices) which comprise the simplex. When a node is instantiated, we assign it a unique integer internal identifier (iID) for use in the development of CASC algorithms. The iID is constant and never exposed to the end-user except for debugging purposes. Instead nodes can be referenced by the user using the `SimplexID` which acts as a convenience wrapper around an `asc_Node*`, providing additional support for move semantics for fast data accession. All topological relations (i.e., edges of the Hasse diagram) are stored in each node as a dictionary which maps user specified keys to `SimplexID`s up and down. An example data structure diagram of triangle {1,2,3} is shown in Fig. 2. Based upon this example, if a user has the `SimplexID` of 1-simplex $\{1, 2\}$ and wishes to get 2-simplex $\{1, 2, 3\}$, they can look in the `Up` dictionary of `SimplexID`$\{1, 2\}$ for key 3 which maps to a `SimplexID`$\{1, 2, 3\}$. The vertex(ices) which constitute each simplex are not stored directly, but can be accessed by aggregating all keys in `Down`.

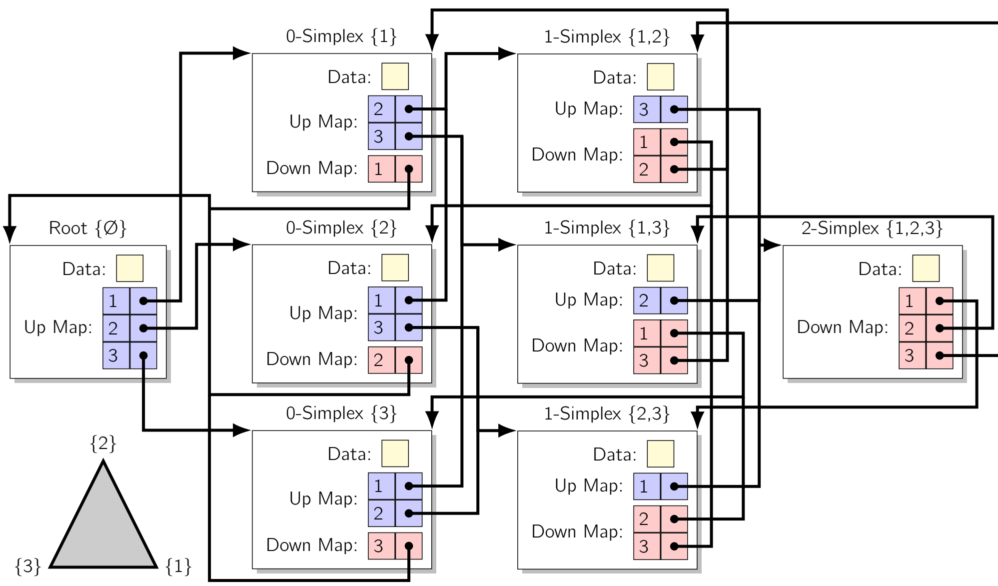


Fig. 2. Data structure diagram of the resulting CASC for a triangle. Each simplex is represented as a node containing a dictionary up and/or down which maps the vertex index to a pointer to the next simplex. Data can be stored at each node with type determined at compile time. Effectively each level can contain different meta-data as defined by the user, separating the interactions of user data from the representation of topology.

We note that while the representation of all topological relations is redundant and may not be memory optimal, it vastly simplifies the traversals across the complex. Furthermore, the associate algorithms and innovations using variable typing are general and thus compatible with other more condensed representations.

3.1 Variable Typing Per Simplex Dimension

We achieve coloring by allowing user-defined data to be stored at each node. The typical challenge for strongly typed languages such as C++ is that the types must be defined at compile time. Typical implementations would either hard code the type to be stored at each level or use a runtime generic type, such as `void*`. However, each of these have drawbacks. For the former, this requires writing a new node data structure for every simplicial complex we may wish to represent. For the latter, using `void*` adds an extra pointer dereference which defeats cache locality and may lead to code instability. Another possible implementation might be to require users of the library to derive their data types from a common class. This solution puts an unnecessary burden on users who may have preexisting class libraries, or simply wish to store a built in type, such as an `int`. To avoid this cumbersome step, we have employed the use of variadic templates introduced in C++11 to allow for unpacking and assignment of data types. The user specifies the types to be stored at each level in a list of templates to the object constructor, see Fig. 3.

The variadic templating allows CASC to represent complexes of any user-defined dimension. For example, suppose we have a 2-simplicial complex intended for visualization and wish to store locations of vertices and colors of faces. A suitable ASC can be constructed using the following template command:

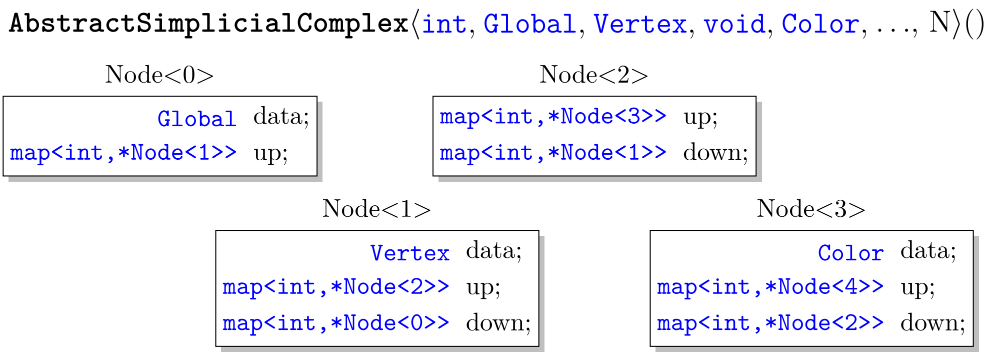


Fig. 3. Template arguments for ASC are unpacked and assigned to nodes accordingly. The first argument "int" is the key type for labeling vertices while the following arguments define node data types. Notably, Node<2> does not allocate memory for data as the corresponding template argument was "void". By passing in additional types, simplicial complexes of higher dimensions are instantiated.

```
auto mesh = AbstractSimplicialComplex<int, void, Vertex, void, Color>()
```

If we now wish to have represent tetrahedral mesh, instead of constructing a new data structure, we can simply adjust the command:

```
auto mesh = AbstractSimplicialComplex<int, void, Vertex, void, Color, Density>()
```

In both cases, the first template argument is the key type for referring to vertices followed by the data type for each k-simplex. Supposing that the user does not wish to store data on any given level, by passing "void" as the template argument, the compiler will optimize the node data type and no memory will be allocated to store data. In both cases, the 0- and 1-simplices will have no data.

By using variadic templates, we allow the user to specify both the dimension of the simplicial complex as well as the types stored at each level. Because the type deduction is performed at compile time, there is no runtime performance impact on user codes. There is, however, some additional code complexity introduced. If the user wishes to retrieve the data stored in a simplicial complex, they must know what level they are accessing at compile time. A consequence is that the exposed identifier object, SimplexID, is templated on the integral level, so that types can be deduced. This does not present a problem for simple use cases, such as:

```
// Create alias for 2-simplicial complex
using ASC = AbstractSimplicialComplex<int,int,int,int>;
ASC mesh = ASC();        // construct the mesh object
mesh.insert<2>({1,2},5); // insert edge {1,2} with data 5
ASC::SimplexID<2> s = mesh.get_simplex<2>({1,2}); // get the edge
// NOTE: the type is templated on the integral level
std::cout << *s << std::endl; // prints data "5"
```

Listing 1. Example instantiation of 2-simplicial complex using CASC.

However, when implementing algorithms intended to be generic on any simplicial complex, templated code must be written. We discuss the implementation of several such algorithms in the following section.

4 IMPLEMENTED ALGORITHMS

The following algorithms are provided with the CASC library:

- Basic Operations
  - Creating and deleting simplices (`insert`/`remove`)
  - Searching and traversing topological relations (`GetSimplexUp`/`GetSimplexDown`)
- Traversals
  - By level (`get_level`)
  - By adjacency (`neighbors_up`/`neighbors_down`)
  - Traversals across multiple node types (Visitor Design Pattern/double dispatch)
- Complex Operations
  - Star/Closure/Link
  - Meta-Data Aware Decimation

4.1 Basic Operations

4.1.1 Creating and deleting simplices. Since the CASC data structure maintains every simplex in the complex and all topological relations, inserting a simplex s into the complex means ensuring the existence of, and possibly creating, $2^{|s|}$ nodes and $|s| \cdot 2^{\dim(s)}$ edges. Fortunately, the combinatorial nature of simplicial complexes allows this to be performed recursively. A generalized recursive insertion operation for any dimensional complex and user specified types, is described in Algorithm 1. The insertion algorithm defines an insertion order such that all dependent simplices exist in the complex prior to the insertion of the next simplex.

As an illustrative example of the template code used in this library, Algorithm 1 is rewritten in C++ template function-like pseudocode shown in Algorithm 6. While the templated code is more complicated, it provides many optimizations. For example, since the looping and recursion are performed at compile time, for any k-simplex we wish to insert, any modern compiler should optimize the code into a series of `insertNode()` calls; the `setupForLoop()` and `forLoop()` function calls can be completely eliminated. As a result, the optimized templated code will exhibit superior run time performance. To illustrate the insertion operation, a graphical representation of inserting tetrahedron {1,2,3,4} by Algorithm 6 is shown in Fig. 4, and step-by-step in Fig. S6. In the example, new simplex $root \cup v$ is added sequentially to the complex and any missing topological relations are found by traversing the faces of *root* and backfilling.

The removal of any simplex is also performed using a recursive template function. When removing simplex s, in order to maintain the property of being a simplicial complex, all cofaces of s or $f \in \mathrm{St}(s)$ must also be removed. The implemented removal algorithm traverses up the complex and removes simplex s and all cofaces of s level by level.

4.1.2 Searching and traversing topological relations. The algorithms for retrieving a simplex as well as for basic traversals from one simplex to another across the data structure are the same. Given a starting simplex, and an array of keys up, the new simplex can be found recursively by Algorithm 2. Since all topological relations are stored, the traversal order across the array of keys does not matter. The same algorithm can be applied going down in dimension. For the retrieval of an arbitrary simplex, we start the search up from the root node of the complex.

4.2 Traversals

Thus far, we have presented algorithms for the creation of a simplicial complex as well as the basic traversal across faces and cofaces. For many applications, other traversals, such as by adjacency, may be more useful. We present several built-in traversal algorithms as well as the visitor design pattern for complicated operations.

ALGORITHM 1: Insertion of a new simplex

```
Input: keys[n]: Indices of n simplices to describe new simplex s
rootSimplex: The simplex to insert relative to (most commonly root)
Output: The new simplex s
Function insert(keys[n], rootSimplex)
{
    for (i = 0; i < n; i++)
    {
        newSimplex = createSimplex(rootSimplex ∪ keys[i])
        if (i > 0) /* Recurse to insert sub-simplices                      */
        {
            /* Pass only the first part of keys up to index i               */
            return insert(keys[0:i], newSimplex)
        }
        else /* Terminal conditional                                        */
        {
            return newSimplex
        }
    }
}
```

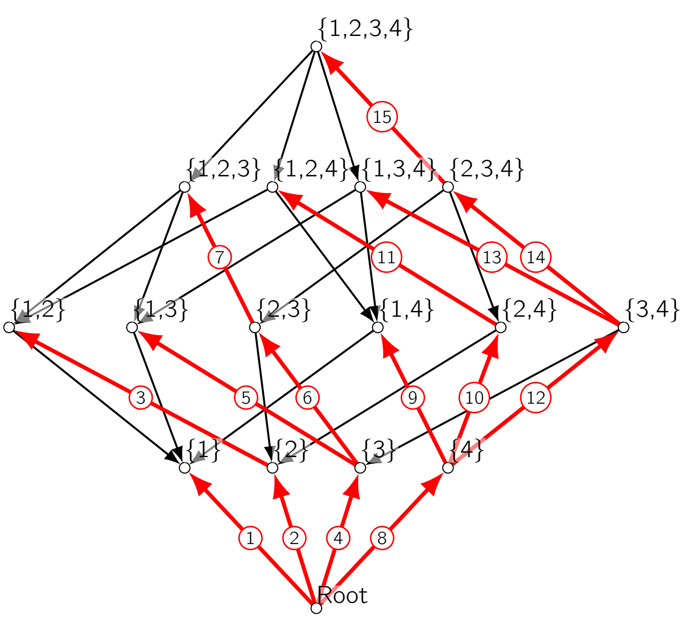


Fig. 4. Recursive insertion of tetrahedral simplex {1,2,3,4}. The order of node insertions is represented by the numbered red arrows. When each node is created, the black arrows to parent simplices are created by backfilling.

4.2.1 By level. It is often useful to have a traversal over all simplices of the same level. For example, iterating across all vertices to compute a center of mass. To support this in an efficient fashion, simplices of the same dimension are stored in a level specific map of iIDs to node pointers.

ALGORITHM 2: Searching for a new simplex

Input: *root*: the simplex to start the search from.
keys[*n*]: the relative name of the desired simplex *s*
Output: The simplex *s*

```
Function getSimplexUp(root, keys[n], index)
{
    if (index < n)
    {
        Find keys[index] in root.up /* Look for the edge up                    */
        if (Edge up is found)
        {
            return getSimplexUp(root ∪ keys[index], keys, index+1)
        }
        else /* Edge keys[index] doesn't exist                                 */
        {
            return null pointer
        }
    }
    else { return root } /* Terminal condition                                 */
}
```

Notably, the map for each level is instantiated with the correct user specified node type with respect to level at compile time. To achieve this, we again use variadic templates to generate a tuple of maps, where each tuple element corresponds to the map for a specific level's node type.

Since asc_nodes are templated on the integral level, we can use a template type map to map an integral sequence to the node pointer type,

$$tuple\langle 1, 2, 3, \ldots \rangle \xrightarrow{Node\langle k \rangle *} tuple\langle Node\langle 1 \rangle *, Node\langle 2 \rangle *, Node\langle 3 \rangle *, \ldots \rangle,$$

producing a tuple of integrally typed node pointers. Subsequently, we can map again to generate a tuple of maps,

$$tuple\langle Node\langle 1 \rangle *, Node\langle 2 \rangle *, \ldots \rangle \xrightarrow{map<int,T>} tuple\langle map\langle int, Node\langle 1 \rangle * \rangle, map\langle int, Node\langle 2 \rangle * \rangle, \ldots \rangle.$$

By using this variadic template mapping strategy we now have the correct typenames assigned. Any level of the tuple can be accessed by getting the integral level using functions in the C++ standard library. Variations of this mapping strategy are also used to construct the SimplexSet and SimplexMap structures below.

For end users, the implementation details are entirely abstracted away. Continuing from the example above, iteration over all vertices of simplicial complex, mesh, can be performed using the provided iterator adaptors as follows.

```
// Deduces the type of nid = ASC::SimplexID<1>
for(auto nid : mesh.get_level_id<1>()){
        std::cout << nid << std::endl;
}
```

Listing 2. Example use of iterator adaptors for traversal across vertices of mesh.

The function get_level_id<k>() retrieves level k from the tuple and returns an iterable range across the corresponding map.

ALGORITHM 3: Get the neighbors of a simplex, s, by inspecting the faces of s

Input: s: The simplex to get the neighbors of.
Output: List of neighbors
IfElseIfElseifelifelse

```
Function getNeighborsUp(s) /* Get the neighbors */
{
    Create an empty list of neighbors
    /* Follow all coboundary relations up from s. */
    for (a ∈ s.up)
    {
        SimplexID id = s ∪ a
        for (b ∈ id) /* Go down from id */
        {
            if (id \ b ≠ s)then { Add id \ b to neighbors } /* Do not add self to neighbors */
        }
    }
    return neighbors
}
```

4.2.2 By adjacency. Many geometric algorithms operate on the local neighborhood of a given simplex. Unlike other data structures such as the halfedge, CASC does not store the notion of the next simplex. Instead, adjacency is identified by searching for simplices with shared faces or cofaces in the complex. The algorithm for finding neighbors with shared faces is shown in Algorithm 3. We note that the set of simplices with shared faces may be different than the set of simplices with shared cofaces. Both adjacency definitions have been implemented and we leave it to the end user to select the relevant function. Once a neighbor list has been aggregated, it can be traversed using standard methods. While the additional adjacency lookup step is extra in comparison to other data structures, in many cases, the generation of neighbor lists need only be done once and cached. The trade off is that CASC offers facile manipulations of the topology without having to worry about reorganizing neighbor pointers.

4.2.3 Traversals over multiple node types. When performing more complicated traversals, such as iterating over the star of a simplex, multiple node types may be encountered. In order to avoid typename comparison based branch statements, we have implemented visitor design pattern-based breadth first searches (BFS). The visitor design pattern refers to a double dispatch strategy where a traversal function takes a visitor functor which implements an overloaded `visit()` function. At each node visited, the traversal function will call `visit()` on the current node. Since the functor overloads `visit()` per node type, the compiler can deduce which visit function to call. Example pseudocode is shown in Listing 3. This double dispatch strategy, eliminates the need for extensive runtime typename comparisons, and enables easy traversals over multiple node types. We provide breadth first traversals up and down the complex from a set of simplices. These visitor traversals are used extensively in the complex operations described below.

```
template <typename Complex>
struct Visitor{
        template <std::size_t k>
        using Simplex = typename Complex::template SimplexID<k>;
        // General template prototype
        template <std::size_t level>
        bool visit(Complex& F, Simplex<level> s){
```

```
                return true;
        }
        // Specialization for vertices
        bool visit(Complex& F, Simplex<1> s){
                s.position = s.position*2;
                return true;
        }
        // Specialization for faces
        bool visit(Complex& F, Simplex<3> s){
                s.color = 'green';
                return true;
        }
};

void BFS(ASC& mesh, Visitor&& v){
        // ... traversal code
        v.visit(mesh, currentSimplex);
        // NOTE: visit is overloaded and called based on function prototype.
}

void main(){
        // ... define simplicial complex traits for a surface mesh
        ASC mesh = ASC(); // construct the mesh object
        // ... insert some simplices etc.
        BFS(mesh, Visitor());
        }
}
```

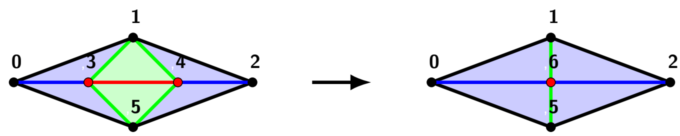
Listing 3. Example pseudocode of double dispatch to traverse the complex scaling the mesh by 2 and coloring the faces green.

4.3 Complex Operations

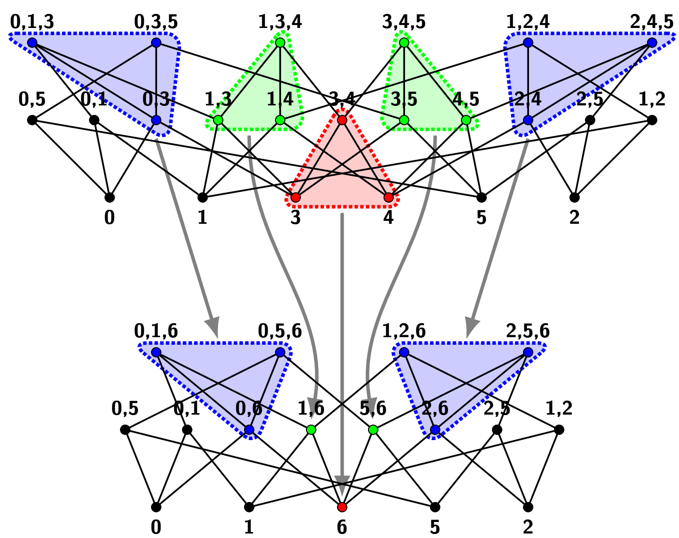
4.3.1 Star/Closure/Link. The star, link, and closure can be computed using the visitor breadth first traversals to collect simplices. These operations typically produce a set of simplices spanning multiple simplex dimensions, and thus simplex typenames, which cannot be stored in a traditional C++ set. We have implemented a multi-set data structure called the `SimplexSet`, which is effectively a tuple of typed sets corresponding to each level. The `SimplexSet` is constructed using the same mapping strategy as the tuple of maps used for the iteration across levels. For convenience, we provide functions for typical set operations such as insertion, removal, search, union, intersection, and difference. Using a combination of the star and closure functions with `SimplexSet` difference we can get the link by Eq. 3.

4.3.2 Meta-Data Aware Decimation. We have implemented a general, decimation algorithm which operates by collapsing simplices of higher dimensions into a vertex. Since simplices are being removed from the complex, user data may be lost. Our implementation is meta-data aware and allows the user to specify what data to keep post decimation. This is achieved by using a recursive algorithm to produce a map of removed simplices to new simplices. The user can use this mapping to define a function which maps the original stored data to the post decimation topology.

This decimation algorithm is a generalization of an edge collapse operation to arbitrary dimensions. It is formally defined as follows:



(a) A geometric realization of the complex before and after decimation.



(b) Explicitly drawn out Hasse diagrams for the constructed example where the TOP is before, and BOTTOM is after decimation. Grey arrows mark the relationships between sets of simplices before and after. Because there is always a mapping, users can define strategies to manage the stored data.

Fig. 5. The example decimation of edge $s = \{3, 4\}$ in a constructed example.

Definition 4.1. Given simplicial complex $\mathcal{F}$, simplex to decimate $s \in \mathcal{F}$, vertex set V of $\mathcal{F}$, and new vertex $p \notin V$, we define function,

$$\varphi(f) = \begin{cases} f & \text{if } f \cap s = \varnothing \\ p \cup (f \setminus s) & \text{if } f \cap s \neq \varnothing \end{cases}, \tag{6}$$

where f is any simplex $f \in \mathcal{F}$. We define the decimation of $\mathcal{F}$ by replacing s with p as $\varphi(\mathcal{F})$.

Note that decimation under this definition is not guaranteed to preserve the topology, as can by seen by decimating any edge or face of a tetrahedron.

Decimation of a simplicial complex must result in a valid triangulation. Here we show that decimation by Definition 4.1 produces a valid abstract simplicial complex.

Theorem 4.2. *$\varphi(\mathcal{F})$ is an abstract simplicial complex.*

Proof. Given simplices y and x, let $y \in \varphi(\mathcal{F})$ and let $x \subset y$. We will show that $x \in \varphi(\mathcal{F})$. There are two cases for $y \cap p$, as they can be disjoint or intersecting.

Considering the disjoint case where $y \cap p = \emptyset$. This implies that $y \in \mathcal{F}$ and $y \cap s = \emptyset$. Since $\mathcal{F}$ is a simplicial complex, $y \in \mathcal{F}$ implies that $x \in \mathcal{F}$. Furthermore since $y \cap s = \emptyset$ and $x \subset y$, then $x \cap s = \emptyset$ implying that $\varphi(x) = x$ and thus $x \in \varphi(\mathcal{F})$.

Alternately in the intersecting case where $y \cap p \neq \emptyset$, that is $y \cap p = p$. Since $x \subset y$ there are two sub-cases where x either contains p or does not.

Supposing that $x \cap p = \emptyset$. Then $x \subset (f \setminus s)$ implying that $x \in \mathcal{F}$ and $x \cap s = \emptyset$. Therefore by the disjoint case, $\varphi(x) = x$ implies that $x \in \varphi(\mathcal{F})$.

Supposing that $x \cap p = p$. We can rewrite any such simplex x as $x = w \cup p$ where $w \cap p = \emptyset$. Furthermore we can write that $y = x \cup r$ where $x \cap r = \emptyset$ and $r \cap p = \emptyset$ and thus $y = w \cup p \cup r$. There exists some set q such that $f = w \cup r \cup q$ and $q \subseteq s$ such that $\varphi(f) = p \cup ((w \cup r \cup q) \setminus s) = y$ Since $f \in \mathcal{F}$ then $w \cup q \in \mathcal{F}$. Therefore $\varphi(w \cup q) = p \cup ((w \cup q) \setminus s) = p \cup w = x$ and thus $x \in \varphi(\mathcal{F})$.

For all cases and sub-cases we have shown that $x \in \varphi(\mathcal{F})$ therefore $\varphi(\mathcal{F})$ is an abstract simplicial complex. □

A pseudocode implementation for this decimation is provided in Algorithm 4. Given some simplicial complex $\mathcal{F}$ and simplex $s \in \mathcal{F}$ to decimate, this algorithm works in four steps. First, we compute the complete neighborhood, $nbhd = \mathrm{St}(\mathrm{Cl}(s))$, of s. Simplices not in the complete neighborhood will be invariant under φ and are ignored. Next, we use a nested set of breadth first searches to walk over the complete neighborhood and compute $p \cup f \setminus s$ for each simplex in the neighborhood. The results are inserted into a `SimplexMap` which maps $\varphi(f)$ to a `SimplexSet` of all f which map to $\varphi(f)$. Third, we iterate over the `SimplexMap` and run the user defined callback on each mapping to generate a list of new simplices and associated mapped data stored in `SimplexDataSet`. Finally, the algorithm removes all simplices in the complete neighborhood and inserts the new mapped simplices.

An example application of this decimation operation is shown in Figure 5. In Figure 5a we show the geometric realization of the complex before and after the decimation of simplex {3,4}. In Figure 5b we show the detailed Hasse diagrams for the constructed example. Note that there are two possible mapping situations. In one case, $f \in \mathrm{St}(\mathrm{Cl}(s)) \cap \mathrm{St}(\mathrm{Cl}(s))$, groups of simplices are merged. In the other case, simplices $f \in \mathrm{St}(\mathrm{Cl}(s)) \setminus \mathrm{St}(\mathrm{Cl}(s))$ only need to be reconnected to the new merged simplices. By carefully choosing the traversal order, some optimizations can be made.

We apply the decimation on the constructed example shown in Figure 5 and show the order of operations with respect to the current visited simplex for each visitor function in Table 1. Starting out, `MainVisitor` and `InnerVisitor` will visit {3,4}. At this point, `GrabVisitor` will search BFS down from {3,4} to grab the set {{3,4}, {3}, {4}} and remove it from the neighborhood, eliminating some future calculations at {3} and {4}. All simplices in this set will map to new simplex {6} after decimation. Continuing upwards, `InnerVisitor` will find simplex {1,3,4} and `GrabVisitor` will grab set {{1,3,4}, {1,3}, {1,4}}. Again this set of simplices will map to common simplex {1,6} post decimation. A similar case occurs with simplex {3,4,5}. At this point, all simplices in $\mathrm{St}(\mathrm{Cl}(s)) \cap \mathrm{St}(\mathrm{Cl}(s))$ have been visited and removed from the neighborhood and `MainVisitor` continues BFS down and finds {3} and calls BFS up (`InnerVisitor`). Note that since simplex {3} has already been grabbed, `InnerVisitor` will continue upwards and find {0,3}. Looking down there are no simplices which are faces of {0,3} in the neighborhood. So on and so forth.

To reiterate, `GrabVisitor` grabs the set of simplices which will be mapped to a common simplex. We show here that the order in which simplices are grabbed by Algorithm 4 will preserve that all simplices $f = w \cup q$ where $q \subseteq s$ will map to $\varphi(f) = w \cup p$.

When visiting any simplex $f \supsetneq q$ where $q \subseteq s$ and q corresponds to simplices visited by `MainVisitor`. We can write f as $f = w \cup q$ where the sets w and q are disjoint. Looking down from f all simplices fall into two cases: $g = v \cup q$ where $\emptyset \subseteq v \subsetneq w$ or $h = w \cup t$ where $t \subsetneq q$. All

ALGORITHM 4: Decimate a simplex by collapsing it into a vertex.

```
Input: F: the simplicial complex;
s: the simplex to decimate;
clbk: a callback function to handle the data mapping
Output:
Simplex np = F.newVertex() /* Create a dummy new vertex to map to */
SimplexSet N /* For the complete neighborhood */
SimplexDataSet data /* Data structure to store (simplex name, simplex data) pairs */
SimplexMap simplexMap /* Data structure to store New Simplex -> SimplexSet map */
/* Get the complete neighborhood (all simplices which are associated with s). */
for (vertex v of s)
{
    BFSup (simplex i ∈ St(v))
    {
        if (j ∉ N) { N.insert (j) }
    }
}
/* Backup the complete neighborhood. These simplices will be destroyed eventually. */
SimplexSet doomed = N
/* Generate the before-after mapping */
BFSdown (simplex i ∈ Cl(s)) /* MainVisitor */
{
    BFSup (simplex j ∈ N ∩ St(i)) /* InnerVisitor */
    {
        /* i maps to np so we need to connect j to np instead */
        Name newName = np ∪ j \ i
        SimplexSet grab
        BFSdown (simplex k ∈ N ∩ Cl(j)) /* GrabVisitor */
        {
            /* Grab dependent simplices which have not been grabbed yet. Grabbed simplices will
               map to simplex newName */
            N.remove(k)
            grab.insert(k)
        }
        if (newName ∉ simplexMap) { simplexMap.insert(pair(newName, grab)) }
        else { simplexMap[newName].insert(grab) }
    }
}
for ({newName, grabbed} ∈ simplexMap)
{
    /* Run the user's callback to map simplex data. */
    DataType mappedData = (*clbk)(F, name(j), newName, grabbed)
    data.insert({newName, mappedData})/* Insert a pair containing new simplex name and data */
}
/* Iterate over the complete neighborhood and remove simplices */
performRemoval(F, doomed)
/* Iterate over data and append mapped simplices and data */
performInsertion(F, data)
```

Order	MainVisitor	InnerVisitor	GrabVisitor	Maps to
1	{3,4}	{3,4}	{3,4}, {3}, {4}	{6}
2	{3,4}	{1,3,4}	{1,3,4}, {1,3}, {1,4}	{1,6}
3	{3,4}	{3,4,5}	{3,4,5}, {3,5}, {4,5}	{3,6}
4	{3}	{0,3}	{0,3}	{0,6}
5	{3}	{0,1,3}	{0,1,3}	{0,1,6}
6	{3}	{0,3,5}	{0,3,5}	{0,5,6}
7	{4}	{2,4}	{3,5}	{5,6}
8	{4}	{1,2,4}	{1,2,4}	{1,2,6}
9	{4}	{2,4,5}	{2,4,5}	{2,5,6}

Table 1. Traversal order of the visitors for the decimation shown in Figure 5a.

simplices of form g, at worst case, will been grabbed while InnerVisitor proceeded BFS up from q. Remaining simplices h can be grouped with f and correctly mapped to $w \cup p$.

We note that in some non-manifold cases GrabVisitor will not always grab set members in one visit. Supposing that we removed simplex {1,3,4} from the constructed example, in this case, InnerVisitor cannot visit {1,3,4} and simplices {1,3} and {1,4} will not be grouped. Instead {1,3} and {1,4} will be found individually when MainVisitor visits {3} then {4}. To catch this case and correctly map {1,3} and {1,4} to {1,6}, we use a SimplexMap to aggregate all maps prior to proceeding.

5 APPLICATION EXAMPLES

CASC is a general simplicial complex data structure which is suitable for use in mesh manipulation and processing. For example, we can use CASC as the underlying representation for an orientable surface mesh. Using a predefined Vertex class which is wrapped around a tensor library, and a class Orientable which wraps an integer, we can easily create a surface mesh embedded in $\mathbb{R}^3$.

```
using Vector = tensor<double,3,1>;
struct Vertex {
        Vector position;
        // ... other helpful vertex functions;
};
struct Orientable {
        int orientation;
};
struct complex_traits
{
    using KeyType = int;
    using NodeTypes = util::type_holder<void,Vertex,void,Orientable>;
    using EdgeTypes = util::type_holder<Orientable,Orientable,Orientable>;
};
using SurfaceMesh = simplicial_complex<complex_traits>;
```

In this case, 1-simplices will store a Vertex type while faces and all edges will store Orientable. Using SurfaceMesh we can easily create functions to load or write common mesh filetypes such as OFF as shown in the included library examples.

We can define a boundary morphism which applies on an ordered k-simplex,

$$\partial_i^k([a_0, \ldots, a_{k-1}]) = (-1)^i([a_0, \ldots, a_{k-1}] \setminus \{a_i\}), \tag{7}$$

ALGORITHM 5: Define the orientation of a topological relation.

Input: Simplices a and b where $b = a \cup v$ and v is a vertex.
Output: The orientation of edge $a \longrightarrow b$

```
int orient = 1
for (Vertex u : a) /* For each vertex u in simplex a */
{
    if (v>a) { orient *= -1; }
}
return orient;
```

where $a_i < a_{i+1}$. Using Algorithm 5, we can apply this morphism to assign a ±1 orientation to each topological relation in the complex. Subsequently, for orientable manifolds, we can compute orientations of faces f_1 and f_2 which share edge e such that,

$$\text{Orient}(e_1) \cdot \text{Orient}(f_1) + \text{Orient}(e_2) \cdot \text{Orient}(f_2) = 0, \tag{8}$$

where e_1 and e_2 correspond to the edge up from e to f_1 and f_2 respectively. Doing so, we create an oriented simplicial complex.

Supposing that we wish to compute the tangent of a vertex as defined by the weighted average tangent of incident faces. This is equivalent to computing the oriented wedge products of each incident face. This can be written generally as,

$$\text{Tangent}(v) = \frac{1}{N}\sum_{i=0}^{N} \text{Orient}(f_i) \cdot (\partial_j(f_i) \wedge \partial_k(f_i)) \tag{9}$$

$$= \frac{1}{N}\sum_{i=0}^{N} \text{Orient}(f_i) \cdot \frac{1}{2}(e_{i,j} \otimes e_{i,k} - e_{i,k} \otimes e_{i,j}), \tag{10}$$

where N is the number of incident faces, f_i is incident face i, j and k are indices of vertex members of f_i not equal to v, and $e_{i,j} = \partial_j(f_i)$. This can be easily computed using a templated recursive function.

```
// Terminal case
auto getTangentH(const SurfaceMesh& mesh,
                const Vector& origin,
                SurfaceMesh::SimplexID<SurfaceMesh::topLevel> curr){
    return (*curr).orientation;
}

template <std::size_t level, std::size_t dimension>
auto getTangentH(const SurfaceMesh& mesh,
                const Vector& origin,
                SurfaceMesh::SimplexID<level> curr){
    tensor<double, 3, SurfaceMesh::topLevel - level> rval;
    auto cover = mesh.get_cover(curr); // Lookup coboundary relations
    for(auto v : cover){
        auto edge = *mesh.get_edge_up(curr, v); // Get the edge object
        const auto& vtx = (*mesh.get_simplex_up({v})).position; // Vertex v
        auto next = mesh.get_simplex_up(curr,v); // Simplex curr union v
        rval += edge.orientation * (v-origin) * getTangentH(mesh, origin, next);
    }
    return rval/cover.size();
```

```
}
```

This demonstrates the ease using the CASC library as an underlying simplicial complex representation. Using the provided API, it is easy to traverse the complex to perform any computations.

6 CONCLUSIONS

CASC provides a general simplicial complex data structure which allows the storage of user defined types at each simplex level. The library comes with a full-featured API providing common simplicial complex operations, as well as support for complex traversals using a visitor. We also provide a meta-data aware decimation algorithm which allows users to collapse simplices of any dimension while preserving data according to a user defined mapping function. Our implementation of CASC using a strongly-typed language is only possible due to recent innovations in language tools. The CASC API abstracts away most of the complicated templating, allowing it to be both modern and easy to use. In the future we hope to incorporate parallelism into the CASC library.

A SUPPLEMENTARY MATERIALS

ACKNOWLEDGMENTS

The authors would like to thank...

This work is supported by the National Institutes of Health under grant numbers P41-GM103426 and RO1-GM31749. CTL also acknowledges support from the NIH Molecular Biophysics Training Grant T32-GM008326. MJH was supported in part by the National Science Foundation under awards DMS-CM1620366 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1620366) and DMS-FRG1262982 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1262982).

REFERENCES

[1] I. Babuška and A. K. Aziz. 1976. On the Angle Condition in the Finite Element Method. *SIAM J. Numer. Anal.* 13, 2 (apr 1976), 214–226. DOI: http://dx.doi.org/10.1137/0713021

[2] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. 1983. Refinement Algorithms and Data Structures for Regular Local Mesh Refinement. In *Scientific Computing: Applications of Methematics and Computing to the Physical Sciences*, R. S. Stepleman (Ed.). North-Holland, 3–17.

[3] Randolph E. Bank and Jinchao Xu. 1996. An algorithm for coarsening unstructured meshes. *Numer. Math.* 73, 1 (mar 1996), 1–36. DOI: http://dx.doi.org/10.1007/s002110050181

[4] Jean-Daniel Boissonnat, Ramsay Dyer, and Arijit Ghosh. 2017. Delaunay Triangulation of Manifolds. *Foundations of Computational Mathematics* (feb 2017). DOI: http://dx.doi.org/10.1007/s10208-017-9344-1

[5] Leila De Floriani and Annie Hui. 2005. Data Structures for Simplicial Complexes: An Analysis and a Comparison. In *Proceedings of the Third Eurographics Symposium on Geometry Processing*. Eurographics Association, Vienna, 119. http://dl.acm.org/citation.cfm?id=1281920.1281940

[6] Zhanheng Gao, Zeyun Yu, and Michael Holst. 2013. Feature-preserving surface mesh smoothing via suboptimal Delaunay triangulation. *Graphical Models* 75, 1 (jan 2013), 23–38. DOI: http://dx.doi.org/10.1016/j.gmod.2012.10.007

[7] A. Liu and B. Joe. 1994. Relationship between tetrahedron shape measures. *BIT* 34, 2 (jun 1994), 268–287. DOI: http://dx.doi.org/10.1007/BF01955874

[8] Elijah Roberts. 2014. Cellular and molecular structure as a unifying framework for whole-cell modeling. *Current Opinion in Structural Biology* 25 (apr 2014), 86–91. DOI: http://dx.doi.org/10.1016/j.sbi.2014.01.005

[9] Hang Si. 2015. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Software* 41, 2 (feb 2015), 1–36. DOI: http://dx.doi.org/10.1145/2629697

[10] Zeyun Yu, Michael J. Holst, and J. Andrew McCammon. 2008a. High-fidelity geometric modeling for biomedical applications. *Finite Elements in Analysis and Design* 44, 11 (jul 2008), 715–723. DOI: http://dx.doi.org/10.1016/j.finel.2008.03.004

[11] Zeyun Yu, Michael J Holst, Takeharu Hayashi, Chandrajit L Bajaj, Mark H Ellisman, J Andrew McCammon, and Masahiko Hoshijima. 2008b. Three-dimensional geometric modeling of membrane-bound organelles in ventricular myocytes: Bridging the gap between microscopic imaging and mathematical simulation. *Journal of Structural Biology* 164, 3 (dec 2008), 304–313. DOI: http://dx.doi.org/10.1016/j.jsb.2008.09.004

[12] Yongjie Zhang. 2016. *Geometric Modeling and Mesh Generation from Scanned Images.* Chapman and Hall/CRC. DOI: http://dx.doi.org/10.1201/b19466

A.1 Templated Insertion Algorithm

ALGORITHM 6: Templated pseudocode implementation of Algorithm 1.

Input: $keys[n]$: Indices of n simplices to describe new simplex s,
$\mathcal{F}$: simplicial complex
Output: The new simplex s

```
/* User function to insert simplex {keys}                                        */
Function insert<n>(keys[n]){
    return setupForLoop<0, n>(root, keys) /* 'root' is the root node             */
}
// The following are private library functions...
/* Array slice operation. Algorithm 1: keys[0:i]                                 */
Function setupForLoop<level, n>(root, keys){ /* General template                 */
    return forLoop<level, n, n>(root, keys) /* Setup the recursive for loop      */
}
Function setupForLoop<level, 0>(root, keys){ /* Terminal condition n = 0         */
    return root
}
/* Templated for loop. Algorithm 1: for (i = 0; i < n; i++)                      */
Function forLoop<level, antistep, n>(root, keys){ /* For loop for antistep       */
    /* n − antistep defines next key to add to root                              */
    insertNode<level, n-antistep>(root, keys)
    return forLoop<level, antistep-1, n>(root, keys)
}
Function forLoop<level, 1, n>(root, keys){ /* Stop when antistep = 1             */
    return insertNode<level, n-1>(root, keys)
}
Function insertNode<level, n>(root, keys){ /* Insert a new node                  */
    v = keys[n]
    if (root ∪ v ∈ F){
        newNode= root.up[v]
    }
    else{ /* Add simplex root ∪ v                                                */
        newNode = createSimplex<n>() /* Create a new node, n-simplex newNode     */
        newNode.down[v] = root /* Connect boundary relation                      */
        root.up[v] = newNode /* Connect coboundary relation                      */
        backfill (root, newNode, v)/* Backfill other topological relations       */
    }
    /* Recurse to insert any cofaces of newNode. Algorithm 1: insert(keys[0:i], newSimplex) */
    return setupForLoop<level+1, n>(newNode, keys)
}
Function backfill<level>(root, newNode, value){ /* Backfilling pointers to other parents */
    for (currentNode in root.down){
        childNode = currentNode.up[value] /* Get simplex currentNode ∪ value     */
        newNode.down[value] = child /* Connect boundary relation                 */
        child.up[value] = newNode /* Connect coboundary relation                 */
    }
}
```

Pseudocode for the algorithms presented in this manuscript have been vastly simplified in order to facilitate understanding. For example, Algorithm 1, while the non-templated version is appears straightforward. It is impossible to be implemented in C++ directly, due to several typing related issues. First, the function prototype for `insert()` requires the `rootSimplex` as the second argument. Simplices at different levels have different types and `insert()` must be overloaded. Similarly the variable `newSimplex` and function `createSimplex()` must know the type of simplex which will be created at compile time.

The actual implementation uses variadic templates to resolve the typing issues. As an example, templated pseudocode for simplex insertion (Algorithm 1) is shown in Algorithm 6. Not only does the templated code automatically build the correct overloaded functions, but it provides many optimizations.

The step-by-step insertion of tetrahedron {1,2,3,4} is shown in Figure 6. Numbered red lines correspond to `newNode` and `root` in function `insertNode()`. Skinny black lines are the topological relations inserted by `backfill()`.

Received 0; revised 0; accepted 0

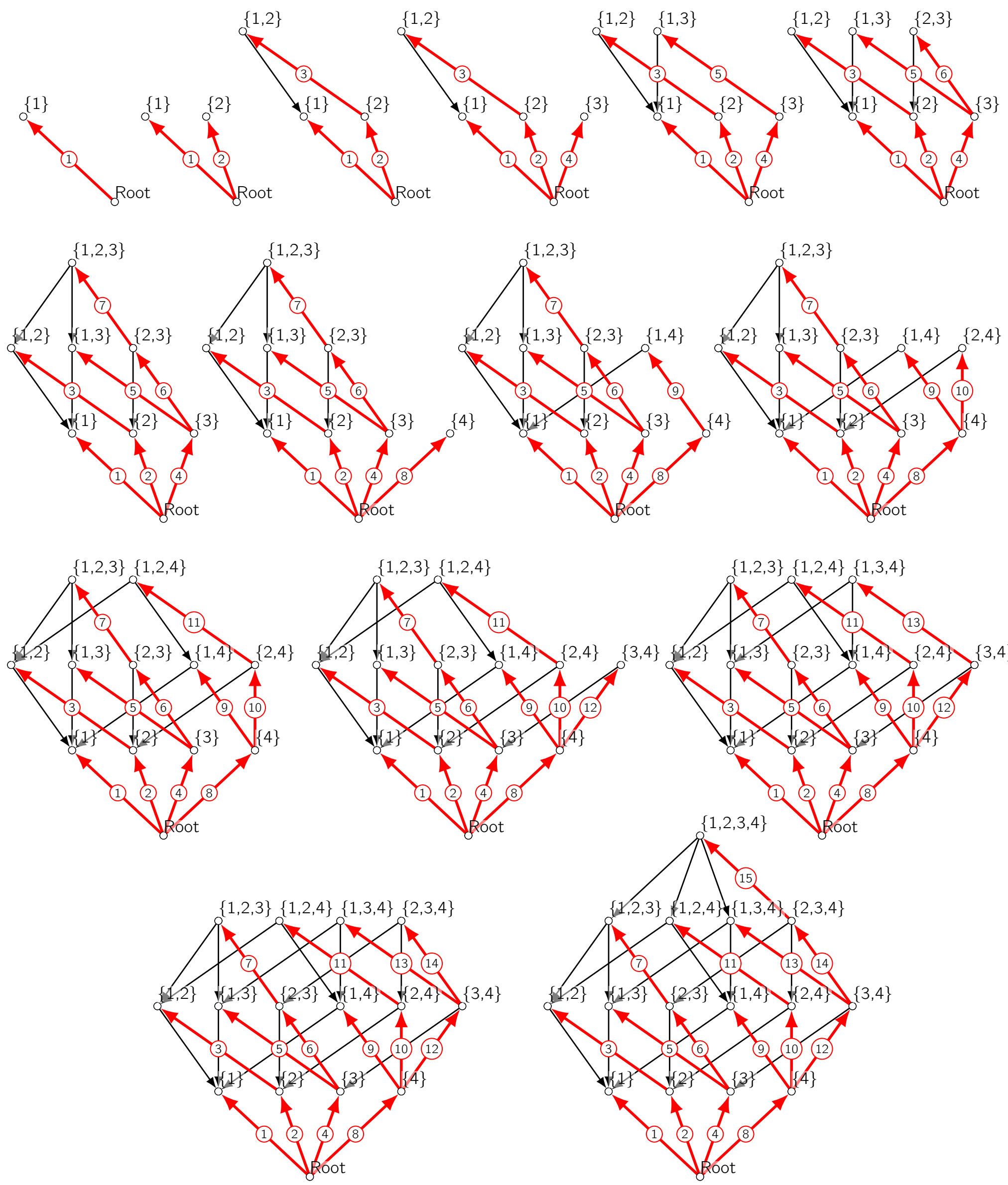


Fig. 6. The Hasse diagrams for the step-by-step insertion of tetrahedron {1,2,3,4} by Algorithm 1. Red lines represent the order of creation for each simplex. The skinny black lines represent where connections to parent simplices are backfilled.