
AHF: Array-Based Half-Facet Data Structure for Mixed-Dimensional and Non-manifold Meshes

Vladimir Dyedov^{1,*}, Navamita Ray¹, Daniel Einstein²,
Xiangmin Jiao^{1,**}, and Timothy J. Tautges³

¹ Dept. of Applied Math. & Statistics, Stony Brook University, Stony Brook, NY 11744
jiao@ams.sunysb.edu

² Biological Sciences Division, Pacific Northwest National Laboratory,
Richland, WA 99352

³ Math. & Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

Summary. We present an *Array-based Half-Facet* mesh data structure, or *AHF*, for efficient mesh query and modification operations. The AHF extends the compact array-based half-edge and half-face data structures (T.J. Alumbaugh and X. Jiao, Compact array-based mesh data structures, IMR, 2005) to support mixed-dimensional and non-manifold meshes. The design goals of our data structure include generality to support such meshes, efficiency of neighborhood queries and mesh modification, compactness of memory footprint, and facilitation of interoperability of mesh-based application codes. To accomplish these goals, our data structure uses *sibling half-facets* as a core abstraction, coupled with other explicit and implicit representations of entities. A unique feature of our data structure is a comprehensive implementation in MATLAB, which allows rapid prototyping, debugging, testing, and deployment of meshing algorithms and other mesh-based numerical methods. We have also developed C++ implementation built on top of MOAB (T.J. Tautges, R. Meyers, and K. Merkley, MOAB: A Mesh-Oriented Database, Sandia National Laboratories, 2004). We present some comparisons of the memory requirements and computational costs, and also demonstrate its effectiveness with a few sample applications.

Keywords: mesh generation, data structure, non-manifold, mixed-dimensional meshes, sibling half-facets, MATLAB.

1 Introduction

Mesh data structures are an important technology underlying meshing algorithms (such as mesh generation and modification) and mesh-based numerical methods

* Current address: CD-adapco, Austin, TX 78750.

** Corresponding author.

(such as finite element and finite volume methods). While mesh data structure has been investigated and used since the inception of mesh generation and computational geometry [1, 2, 8, 10, 11, 13, 16, 17], the increasing complexities and the ever-changing demands of the application codes often pose new requirements to mesh data structures. Examples of such new demanding applications include coupled multiphysics simulations and multi-component systems, which may pose diverse requirements within each code as well as requirements on interoperability across different codes. Therefore, mesh data structure has continued to be an active research topic in recent years.

In light of the new challenges of meshing applications, we summarize a few requirements of mesh data structures, which will serve as our design goals:

Generality: Support mixed-dimensional, non-manifold (oriented or non-oriented) meshes in 1-D, 2-D, and 3-D, and be easily generalizable to higher dimensions.

Efficiency: Support all local adjacency queries in constant time, assuming the valence of each mesh entity is bounded by a small constant.

Simplicity: Be simple and intuitive, and be easy to implement.

Extensibility: Allow extensibility for performance and parallelization.

Interoperability: Facilitate interoperability with application codes, such as simulation codes, multigrid solvers, etc.

Compactness of memory footprint: Require minimal storage in addition to element connectivity.

In this paper, we strive to develop a mesh data structure that meets all the above requirements. We refer to our data structure as the *Array-based Half-Facet* data structure, or *AHF*. The AHF unifies the array-based half-edge and half-face data structures for surface and volume meshes [1], and further generalizes them to support mixed-dimensional and non-manifold meshes by introducing the concept of *sibling half-facets*. It can be used to perform efficient mesh queries and modification.

The key contributions of this paper are twofold: First, we introduce a simple data model for mixed-dimensional and non-manifold meshes. Our data model is easy to implement and is efficient in both memory and computational cost. In addition, some of its data fields can be created dynamically for fine-grain operations and be removed afterwards. Second, as an array-based data structure, AHF facilitates better interoperability across different application codes, different programming languages (such as MATLAB, C/C++, FORTRAN, etc.), and different hardware platforms. Our C++ implementation on top of MOAB [17] further improves its interoperability through the iMesh interface (<http://www.itaps.org/>). In addition, our data structure is unique in its comprehensive implementation in MATLAB, which allows rapid prototyping and deployment of meshing algorithms and other mesh-based numerical methods in a productive fashion.

The remainder of the paper is organized as follows. Section 2 reviews some background knowledge and related mesh data structures. Section 3 describes our data model for non-manifold and mixed-dimensional meshes. Section 4 describes the

algorithms for the construction, query, mesh modification operations, as well as their implementations in MATLAB. Section 5 describes the C++ implementation on top of MOAB. Section 6 presents some comparisons of AHF against others in terms of storage and computational cost. Section 7 concludes the paper with a discussion.

2 Background and Related Work

In this section, we review some terminology and other existing data structures, which will establish the foundation of our proposed data structure.

2.1 Terminology

We develop data structures for representing discrete geometric and topological objects in 1-D, 2-D, or 3-D, arising from numerical computations in engineering and scientific applications. These objects correspond to *curves*, *surfaces*, and *volumes*, respectively, typically embedded in two- or three-dimensional Euclidean spaces. Topologically, a d -dimensional object is a *manifold with boundary* if every point in it has a neighborhood homeomorphic to either a d -dimensional ball or half-ball, where the points whose neighborhood is homeomorphic to a half-ball are *boundary* (or *border*) *points*. In practice, it is quite common to have topological objects that are *non-manifold*, especially for curves and surfaces. These non-manifolds are typically composed of a union of a finite number of manifolds with boundaries, and sometimes embedded in a higher-dimensional manifold structure. In 3-D space, a surface is *oriented* if it is possible to make a consistent choice of surface normal vector at every point; otherwise it is *non-oriented*.

In our setting, a *mesh* is a simplicial complex representing discretely a geometric or topological object. We say a mesh is 1-D, 2-D, or 3-D if the object that it represents is topologically 1-D, 2-D, or 3-D, respectively. We say a mesh is a *manifold* or *non-manifold* if its geometric realization is a manifold or non-manifold, respectively. A *mesh* is composed of 0-D, 1-D, 2-D, and 3-D entities, which we refer to as *vertices*, *edges*, *faces*, and *cells*, respectively. Typically, a face is either a triangle or quadrilateral, and a cell is a tetrahedron, prism, pyramid, or hexahedron, especially for finite element methods, although general polygons and polyhedra are also often used in finite volume meshes. We focus on finite-element meshes, and will briefly describe the generalization to polyhedral meshes in Section 7.

In a d -dimensional mesh, we refer to the d -dimensional entities as *elements*, and refer to the $(d - 1)$ -dimensional sub-entities as its *facets*. More specifically, the facets of a cell are its faces, the facets of a face are its edges, and a facet of an edge are its vertices. Each facet has an orientation with respect to the containing element. For example, each edge of a triangle has a direction, and all the edges form an oriented loop. Thus it makes sense to call the facets as *half-facets*. Each facet may have multiple incident elements, especially for non-manifold entities. We refer to all

such half-facets as *sibling half-facets*. A half-facet without any sibling is a *border half-facet*, and refer to vertices incident on a border half-facet as a *border vertex*. A mesh is said to be *conformal* if the pairwise intersection of any two entities is either another entity or is empty. In this paper, we consider only conformal meshes, which may be manifold or non-manifold. In the case of surface meshes, the mesh may be oriented or non-oriented.

In some engineering applications, especially in coupled or multi-component systems, the domain of interest may be composed of a union of topologically 1-D, 2-D, and 3-D objects, such as a mixture of cables, thin-shells, and solids. We refer to such a domain and its mesh as *mixed-dimensional*.¹ We refer to a subset of the mesh corresponding to a 1-D, 2-D, and 3-D object in the domain as a *sub-mesh*. It is common, although not required, for the sub-meshes to share some mesh entities, especially shared vertices. Our goal is to design data structures for consistent representations for mixed-dimensional meshes with shared entities.

2.2 Half-Edge Data Structure

The *half-edge data structure*, a.k.a. the *doubly-connected edge list (DCEL)*, is a popular data structure for 2-D and surface meshes (see e.g. [5]), especially oriented, manifold, polygonal surface meshes with or without boundary. The DCEL uses edges as the core object. The edge within each face is called a *directed edge* or *half-edge*. In an oriented manifold surface mesh, suppose the edges within each face can be ordered in counter-clockwise order with respect to outward normal (or upward normal for 2-D meshes). Each edge has two incident faces, and the two half-edges have opposite orientations and hence are said to be *opposite* or *twin* of each other. An edge on the boundary does not have a twin half-edge.

There are various implementations of DCEL. A typical implementation, such as those in CGAL [10, 7], OpenMesh [3] and Surface_Mesh [15], stores the mappings from each half-edge to its opposite half-edge, its previous and next half-edge within its face, its vertices, its incident face, as well as the mapping from each vertex and each face to an incident half-edge. More compact representations, such as [1], can be obtained by storing only the mapping between opposite half-edge, optionally the mapping from each vertex to an incident half-edge, along with the element connectivity. The above implementations do not support non-manifold or non-oriented meshes, which we overcome in the proposed data structure.

2.3 Half-Face Data Structure

A generalization of the concept of DCEL to volume meshes is the so-called the *half-face data structure* [1, 13]. Within each cell, suppose the edges of each face are oriented in counter-clockwise order with respect to the outward normal of the cell. We refer to the oriented faces as *half-faces*. For typical meshes in engineering

¹ These meshes are sometimes referred to as *mixed* or *hybrid meshes*, which we avoid here since they may also refer to meshes with mixed-types of elements of the same dimension.

applications, each face in the interior of a volume mesh has two corresponding half-faces with opposite orientations, which are said to be *opposite* or *twin* of each other.

The data structure in [1] was designed for 3-manifold with boundary composed of the standard elements. Although this is the typical case in applications, a volume mesh may also be non-manifold. For example, this can happen if the domain contains two objects that intersect at a single vertex or along an edge. For generality, we consider volume meshes that may be manifold or non-manifold, and allow them to be oriented or non-oriented. OpenVolumeMesh [13] was developed for general polytopal meshes, which may be non-manifold. However, the data model in OpenVolumeMesh is quite complicated, because unlike the edges in a cell, the faces within a cell do not have a natural topological order. A mesh data structure for non-manifold meshes in arbitrary dimensions was proposed in [4]. This data structure stores not only the adjacency information of the highest dimensional elements but also some adjacency information of intermediate elements, which incur extra storage overhead. In addition, in its proposed form, it could not handle mixed-dimensional meshes that contain submeshes of different dimensions. We seek a simpler and a more unified data model.

2.4 Pointer-Based Versus Array-Based Implementations

A mesh data structure may be implemented using either pointers or arrays. The pointer-based implementations are more common, since they are relatively easy to manipulate. For example, the DCEL implementation in CGAL is pointer-based. Other examples, which are not based on half-edges or half-faces, include FMDB [16], MSTK [8], libMesh [11], etc. In such an implementation, the entities are represented as “objects” explicitly, and pointers (or handles) are used to refer to these explicit objects.

In contrast, in an array-based implementation, we do not represent the entities in the mesh as objects. Instead, an attribute of all the entities of the same type are stored in one or a few arrays, and the attributes for a single entity may be distributed in different arrays. An entity may be referenced through an ID or “handle”, which can be mapped easily to array indices. The half-edge and half-face data structures in [1], MOAB [17], and OpenVolumeMesh [13] are array-based.

In our work, we choose to use array-based, pointer-free implementations for a number of reasons. First, in an array-based implementation, we can treat intermediate dimensional entities (such as half-facets) as implicit entities, and reference them without forming explicit objects. This can lead to significant savings in storage, especially on computers with 64-bit pointers. Second, using arrays can also lead to faster memory access and hence better efficiency. In addition, array-based implementations also offer better interoperability across application codes, different programming languages, and different hardware platforms (such as between GPUs and CPUs).

3 Data Model for Non-manifold and Mixed-Dimensional Meshes

The basic half-edge and half-face data structures are simple, but they were restricted to oriented, manifold meshes (with or without boundary) in 2-D and 3-D, respectively. In this section, we present a simple and unified generalization to mixed-dimensional meshes, which may be non-manifold and/or non-oriented.

3.1 Unification of Half-Edges and Half-Faces

We unify the data models of half-edge and half-face data structures. In this unified data model, the key abstractions for a d -dimensional mesh are:

- vertices*: 0-dimensional entities (a.k.a. nodes);
- elements*: d -dimensional entities (faces and cells in 2-D and 3-D, respectively);
- half-facets*: $(d-1)$ -dimensional sub-entities of a d -dimensional entity (half-edges and half-faces of a 2-D and 3-D element).

In this data model, we assume that the element has a standard numbering convention for its vertices and its facets. For standard elements, we follow the convention of the CGNS (CFD General Notation System) [12, 18], as illustrated in Fig. 1. We do not

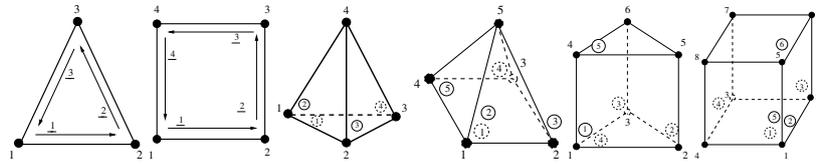


Fig. 1 CGNS numbering conventions for 2-D and 3-D elements. Underscored numbers correspond to local edge IDs, and circled ones correspond to local face IDs.

require explicit representation of intermediate dimensional entities between 1 and $d-1$. Instead, we treat the half-facet as an *implicit entity*, and refer to a half-facet using the element ID and its local ID within the element. We refer to this data model as the *half-facet data structure*.

The above unified model is not limited to $d=2$ and 3. In fact, it can be applied to any d , as long as the numbering convention is predefined for the vertices and its facets. In particular, for $d=1$, we refer to this representation for curves as the *half-vertex data structure*, which is a specialization of the half-facet data structure to curves. For $d \geq 2$, it provides a compact representation, since the intermediate dimensional entities are not stored but referenced implicitly instead. In addition, this data model can also be used for meshes with high-order elements (such as six-node triangles or 10-node tetrahedra), where the mid-edge, mid-face or mid-cell nodes do not affect the definition and identification of the half-facets.

We store the data structure similar to that in [1]. The element connectivity is stored in arrays in a manner similar to a typical finite element code. The mappings between sibling half-facets are stored in a 2-D array or in separate arrays, one per unique element type. Let $|E|$ and f denote the number of elements and the maximum number of facets in an element, respectively. We denote each half-facet by a two tuple $\langle eid, lfid \rangle$, where eid denotes the element ID, which starts from 1, and $lfid$ denotes the local facet ID, which starts from 0. For typical meshes, the two tuple can be encoded into a single 32-bit unsigned integer, by using first d bits to storing the local facet ID of a d -dimensional element, and using the remaining bits to storing the element ID. This allows up to about 500 million elements for volume meshes. For very large meshes, we assign a 32-bit unsigned integer to the element ID and an unsigned 8-bit integer to the local ID, and store them in separate arrays. This allows up to 4 billion elements with minimal extra storage overhead. Depending on whether the half-facet IDs are encoded in a single integer or in two integers, we can store the mappings in either a single array or two arrays, respectively, where each array is of size $|E| \times f$. In addition, the vertex to half-facet mapping is stored in a single 1-D array. For most meshes, we need to store only one incident half-facet. Some complications may arise for non-manifold vertices, which we address next.

3.2 Generalization to Non-manifold Meshes

In a non-manifold mesh, there can be more than two elements abutting the same facet, unlike a manifold mesh where there can be only up to two. We still refer to an oriented facet within an element as a *half-facet*, but it no longer has a “twin” or “opposite” half-facet in general. We refer to the half-facets corresponding to the same facet as *sibling half-facets*. The orientations of two sibling half-facets are not required to be opposite to each other, and therefore this generalization also allows representing non-oriented meshes, such as the Möbius strip.

An important issue is the storage for mapping between the sibling half-facets. Instead of *doubly-connected linked list* for twin half-facets, we use a cyclic linked list, which allows us to preserve the storage structure and also to traverse all the elements incident on a half-facet. Figure 2 shows an example of a non-manifold edge (edge joining vertex 2 and 3) present in the given triangulated mesh, as well as the element connectivity, sibling half-edge map and vertex to half-edge maps. Note that we do not necessarily need to sort the half-facets in any particular order. In fact, an ordering may not be well-defined in some cases. However, the data structure does not exclude the user from ordering the half-facets.

In a d -dimensional non-manifold mesh, when $d > 1$, there may exist a vertex whose neighborhood is not a d -dimensional ball or half-ball, but instead the union of two or more d -dimensional ball or half-balls that intersect at an entity of lower than $(d - 1)$ dimension (such as at the vertex in a surface mesh, or at a vertex or edge in a volume mesh). We refer to such a vertex as a *non-manifold vertex*. Some minor modification to the data structure is necessary in this setting. In particular, at a non-manifold vertex, we need to store a 1-to- n mapping instead of a 1-to-1 mapping to

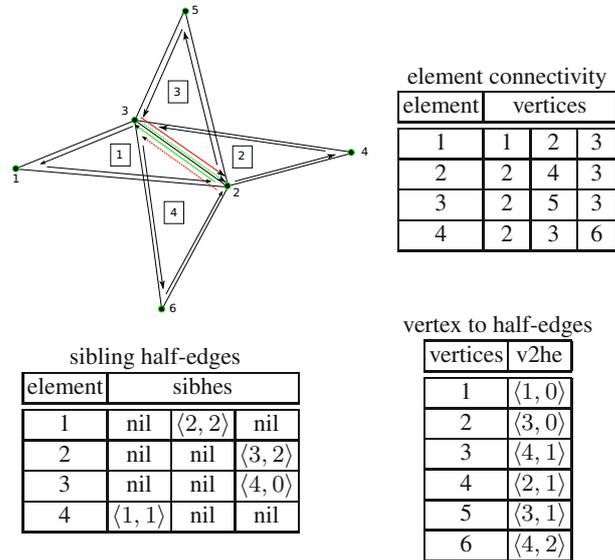


Fig. 2 Example of a non-manifold mesh, along with its element connectivity, sibling half-edges, and mapping from each vertex to an incident half-edge

its incident half-facets. Note that this also covers the case where there exist an edge in a volume mesh whose neighborhood is the union of two or more d -dimensional ball or half-balls that intersect only at the vertex.

3.3 Generalization to Mixed-Dimensional Meshes

The concept of *sibling half-facets* unifies the half-vertex, half-edge, and half-face data structures for 1-D, 2-D, and 3-D meshes, which may be manifold or non-manifold with boundary. In a mixed-dimensional mesh, the sub-meshes of different dimensions can share entities. In particular, it is most common for the meshes to share vertices. However, these entities may have different representations.

This unification allows an easy extension to support mixed-dimensional meshes, which may be composed of sub-meshes of 1-D, 2-D, and 3-D. Figure 3 shows a diagram of a typical half-facet data structure, where the half-vertices and half-edges are only required for explicit edges and faces in the mesh, respectively. We refer to this data structure for mixed-dimensional meshes as *Array-based Half-Facet data structure*, or *AHF*. This data structure is very simple and modular, as the individual sub-meshes of different dimensions are self-contained, and they can be maintained separately. They also allow us to traverse between multiple dimensions efficiently. The interactions of the different dimensions are all performed through the shared vertices.

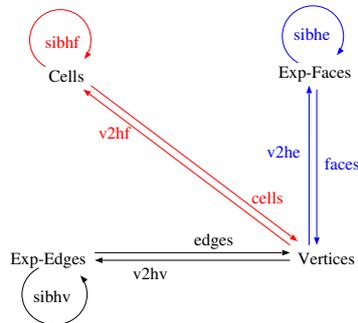


Fig. 3 Typical AHF for mixed-dimensional meshes is composed half-vertex (black, for explicit edges only), half-edge (blue, for explicit faces only), and half-face (red) data structures.

4 Construction, Query, and Modification of AHF

In this section, we describe some detailed algorithms for the construction of AHF, as well as some query and modification operations. Since AHF is array-based, these algorithms can be implemented in any programming languages, including MATLAB, C/C++, FORTRAN, etc. We will also describe our implementation in MATLAB.

4.1 Construction of AHF

In the half-facet data structure, there are two components: *sibhfs* (sibling half-facets) and *v2hf* (vertex to half-facet). The former is central to AHF, as nearly all adjacency queries require it. These sibling half-facets should map to each other and form a cycle. The latter array, *v2hf*, is optional for many operations, it can be created only when needed.

In general, we construct the AHF in two steps, which construct these two mappings, respectively. In a mixed-dimensional mesh, the AHF for the submesh of each each is constructed independently of each other. In the following, we describe the two steps in a manner independently of the dimension of the mesh.

Identification of Sibling Half-Facets

During the first step, we determine the sibling half-facets and construct a cyclic mapping between them. The key components of this step are two intermediate mappings:

v2hfs: a mapping from each vertex to its incident half-facets in which the vertex has largest ID;

v2adj: a mapping from each vertex to its adjacent vertices in each of the above incident half-facets.

Algorithm 1 outlines the procedure for the first stage, which is applicable to half-facets in arbitrary dimensions, and it is particularly efficient in 1 to 3 dimensions.

Algorithm 1. Determination of sibling half-facets.

Input: elems: element connectivity
Output: sibhfs: cyclic mappings of sibling half-facets

- 1: **for each** elements e in elems **do**
- 2: **for each** facet f in e **do**
- 3: $v \leftarrow$ vertex with largest ID within f ;
- 4: $us \leftarrow$ set of adjacent vertices of v in f ;
- 5: Append f into $v2hfs(v)$, and append us into $v2adj(v, f)$;
- 6: **end for**
- 7: **end for**
- 8: **for each** elements e in elems **do**
- 9: **for each** facet f in e **do**
- 10: **if** sibhfs(f) is not set **then**
- 11: $v \leftarrow$ vertex with largest ID within f , and $us = v2adj(v, f)$;
- 12: Find half-facets in $v2hfs(v)$ s.t. $v2adj(v, \cdot) = us$;
- 13: Form a cyclic mapping for these half-facets in sibhfs;
- 14: **end if**
- 15: **end for**
- 16: **end for**

The computational cost of Algorithm 1 is linear, assuming that the number of facets incident on a vertex is bounded by a small constant, say c . To analyze the storage requirement, let $|H|$ denote the number of half-facets in the mesh. The output requires approximately $|H|$ integers. The algorithm requires two intermediate maps $v2hf$ and $v2adj$, which require $c|H|$ integers total. This intermediate storage requirement can be further reduced by equally distributing the vertices into b buckets and process each bucket in a separate pass, which then would require only $|H|/b$ integers.

Construction of Incident Half-Facet of Vertex

During the second step, we construct a mapping from each vertex to an incident half-facet. This is done by utilizing the sibling half-facets obtained from the first step, as outlined in Algorithm 2.

When determining the incident half-facets, we give higher-priorities to border half-facets, so that from its output $v2hf$, we can determine whether a vertex v is a border vertex by simply checking whether $v2hf(v)$ is a border half-facet. In addition, a minor variant of Algorithm 2 can be used to construct a bitmap of vertices to determine whether all the vertices are border vertices without forming $v2hf$. These functions can be useful, for example when extracting the boundary of a mesh or when imposing boundary conditions in numerical computations. The computational cost of Algorithm 2 is also linear in the number of vertices plus the number of half-facets. It does not require any intermediate storage. In addition, the AHF can be used to extract the internal boundaries between different materials in a mesh.

Algorithm 2. Construction of mapping from vertex to an incident half-facet.

Input: elems: element connectivity
sibhfs: cyclic mappings of sibling half-facets
Output: v2hf: vertex to an incident half-facet

```

1: for each elements  $e$  in elems do
2:   for each vertex  $v$  of  $e$  do
3:     if v2hf( $v$ )==0 then
4:       v2hf( $v$ )← a facet incident on  $v$  in  $e$ 
5:     end if
6:   end for
   {Give border facets higher priorities}
7:   for each facet  $f$  in  $e$  do
8:     if sibhfs( $e, f$ )==0 then
9:       for each vertex of  $f$  do
10:        Set v2hf( $v$ ) to  $\langle e, f \rangle$ ;
11:      end for
12:     end if
13:   end for
14: end for

```

4.2 Algorithms of Adjacency Queries

We consider two classes of queries, which are representative:

1. Given an explicit d -dimensional entity, obtain neighbor d -dimensional entities that share a $(d - 1)$ -dimensional sub-entity with it.
2. Given an explicit d -dimensional entity, obtain the $(d+1)$ -dimensional entities incident on it.

We consider these operations for mixed-dimensional meshes with 1-D, 2-D, and 3-D explicit entities, and resulting in six operations total:

- 1a. for each edge, obtain vertex-connected neighbor edges;
- 1b. for each face, obtain edge-connected neighbor faces;
- 1c. for each cell, obtain face-connected neighbor cells;
- 2a. for each vertex, obtain incident edges;
- 2b. for each edge, obtain incident faces;
- 2c. for each face, obtain incident cells;

Figure 4 shows two examples of these operations, where the left figure panel shows the set of neighbor edges (in red) of a given (blue) edge, and the right panel shows the neighborhood faces (in red) of a given (blue) triangle.

We describe the procedures in a dimension-independent fashion. In the first class of operations, given a d -dimensional entity, we simply need to loop through its $(d - 1)$ -dimensional facets, for each of its facets obtain the sibling half-facets. Then by decoding the ID of the sibling half-facets, we obtain the neighbor d -dimensional entities. The algorithm takes time proportional to the output size, which is a constant.

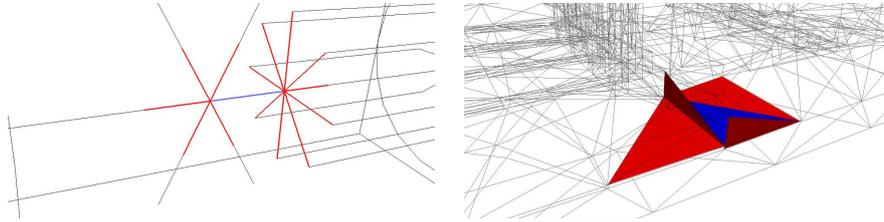


Fig. 4 Example adjacency queries for non-manifold meshes. Left panel shows neighbor edges (red) of a given blue edge. Right panel shows the neighbor faces of a given blue triangle.

The second class of operations are across the sub-meshes of different dimensions. The algorithm proceeds in two steps: 1) given an explicit d -dimensional entity, find a corresponding half-facet h in the $(d+1)$ -dimensional sub-mesh through the shared vertices; 2) find the sibling half-facets of this half-facet h , and decode the half-facet IDs to obtain all the adjacent $(d+1)$ -dimensional entities. In terms of computational cost, the first step takes time proportional to the number of incident entities of a vertex, and the second step takes time proportional to the size of the output. Both steps in general require constant time.

4.3 Mesh Modification Operations

Mesh-modification can be implemented relatively easily in AHF. For mixed dimensional meshes, the AHF is particularly attractive, because the adaptivity for different dimensions can be done nearly independently, and they only need to be synchronized at the shared vertices. This leads to very modular adaptivity strategies.

Within each dimension, the AHF can be modified either locally or globally. The local modification is performed through the mesh-modification primitives, such as edge flipping, edge splitting, and edge collapse, especially for triangular and tetrahedral meshes. As an example, Figure 5 illustrates the standard flipping operations for tetrahedra between a current n -complex of elements and a resulting m -complex, where the complex is the neighborhood of elements considered. We consider the standard operations, where the n - m combinations are 3-2, 2-3, 4-4 and 2-2. For the 4-4 and 2-2 flipping operations, both the numbers of vertices and of elements remain the same, and therefore the operation involves only updating the elems (element connectivity), sibhfs (sibling half-faces), and v2hf (vertex-to-half-face) mappings locally within the complex. For 3-2 flipping, and similarly for edge collapse, the number of elements decreases. This may result in a hole in the arrays elems and sibhfs. We fill the hole by swapping the element with the highest ID into the hole and updating the half-facets in sibhfs and v2hf, so that the element IDs remain consecutive. For 2-3 flipping, and similarly for edge splitting, the number of elements increases. This may require reallocating and copying the arrays. To avoid excessive memory copying, we expand the array by a small percentage (e.g. by 20%) each reallocation, so that the amortized cost for the local modifications is constant,

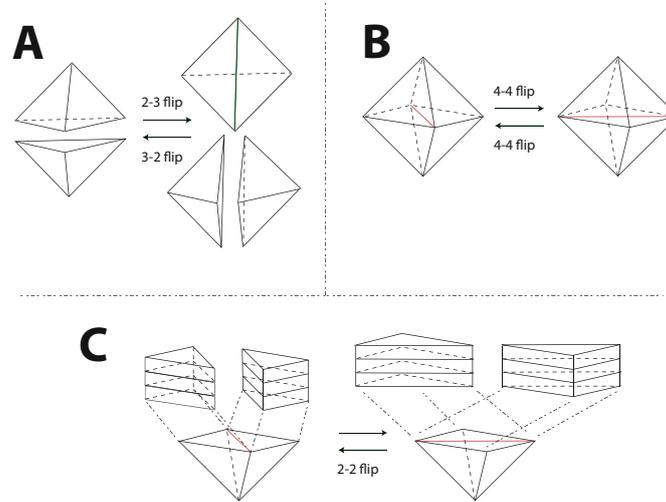


Fig. 5 Standard edge-flipping operations on a tetrahedral mesh. The 2-3 and 3-2 flips (A) increase or decrease the number of tetrahedra, respectively. In a manifold mesh, the 4-4 flip (B) can be applied in the interior. The 2-2 flip (C) is defined only on the boundary surface, or a tetrahedral mesh with prismatic boundary layers [6].

assuming the number of flipping and splitting operations are proportional to the size of the mesh. As an example, Figure 6 shows a tetrahedral mesh for a heart model, which was first generated by using TetGen [14], followed by applying flipping and smoothing implemented using AHF. The figure also shows the quality measures in terms of the orthogonality, skewness and uniformity [9], which are important measures for finite volume methods.

4.4 Implementations in MATLAB

Since our data structure is array-based and pointer-free, we can implement it conveniently in MATLAB. The MATLAB provides a user-friendly programming environment, so that our MATLAB implementation allows rapid prototyping and testing of sophisticated meshing algorithms and mesh-based numerical methods. In our MATLAB implementation, we support two ways to store the half-facet. The first way is to encode the two-tuple ID in a single integer, where d bits are reserved the local facet ID for a d -dimensional mesh. In the second way, we store the element ID and local facet ID into a 32-bit integer array and an 8-bit integer array, respectively, and they pack these two arrays into a single MATLAB struct. For a mixed dimensional mesh, the complete mesh is packed into a single struct composed of the arrays of different dimensions. Both of our implementations are compatible with MATLAB Coder, which allows generation of efficient and portable ANSI C code from our MATLAB implementation. The generated C code can be used as stand-alone libraries, or be compiled into MEX functions and be called in MATLAB or GNU Octave.

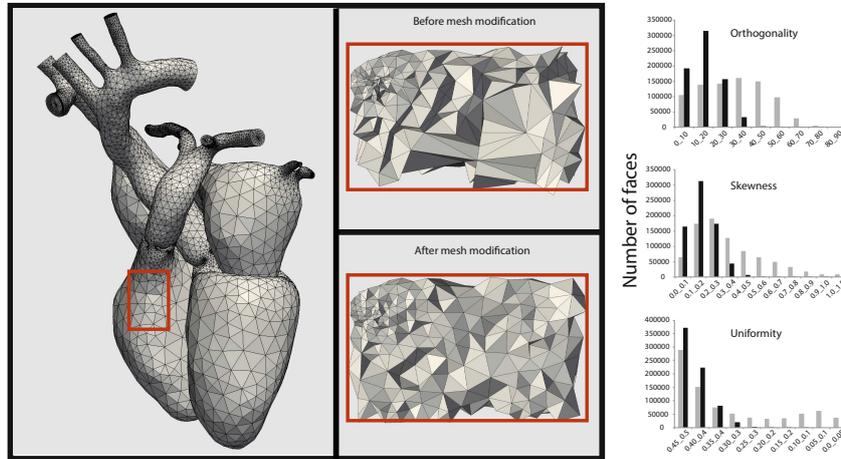


Fig. 6 An example tetrahedral mesh of a heart model optimized using mesh flipping and smoothing implemented using AHF. The histograms show the quality measures in terms of orthogonality, skewness and uniformity of the mesh before (gray) and after (black) optimization, where smaller values correspond to better qualities.

5 Integration of AHF into MOAB

The Mesh Oriented datABase (MOAB [17]) is a mesh data representation designed to support a range of mesh related operations, such as memory efficient mesh representation, mesh querying and representation of application specific data. The internal storage of MOAB is array-based and supports querying of adjacent entities by target dimension.

The non-vertex entity-to-entity adjacencies are created and stored only upon application request. Since most of the applications do require some kind of auxiliary entity adjacency information, MOAB may require significant storage in such situations. This imposes extra storage requirements on the application. The AHF comes handy precisely in such situations. It adds the flexibility of intermediate entity adjacency querying without explicitly storing them. In addition, MOAB does not support implicit entities, and does not store the neighboring information of entities of the same dimension. Therefore, it is desirable to incorporate AHF into MOAB. We refer to this implementation as *MOAB_AHF*, which can be created dynamically for some queries and be deallocated afterwards.

There are some important differences between a standalone AHF implementation and *MOAB_AHF*. In terms of storage, MOAB uses “tags” to store application specific data defined on mesh entities or entity sets. In *MOAB_AHF*, we store *sibhfs* and *v2hf* as tags, instead of standalone arrays. In addition, since MOAB references elements and other entities through handles, we store the *sibhfs* as two tags: one for the handles to the elements, and the other for local facet IDs. Another difference is that MOAB uses a different numbering convention of the local facet IDs than that

of CGNS. To be self-consistent, we use the MOAB’s own numbering convention in MOAB_AHF. Finally, in MOAB_AHF, we construct the sibhfs without forming the v2hfs and v2adj in Algorithm 1, and instead using MOAB’s built-in function “get_adjacency” to identify the elements adjacent to an entity.

6 Experimental Comparisons

In this section, we present some experimental studies of AHF, MOAB and MOAB_AHF in terms of storage requirements and computational cost.

6.1 Cost in Construction of Data Structure

We first compare the computational times in constructing the data structures. For AHF, we used the C code generated from our MATLAB implementation with MATLAB Coder 2.4 released with MATLAB R2013a. We compiled AHF, MOAB, and MOAB_AHF using gcc 4.4.3 with optimization enabled. All the tests were performed on a Linux computer with a 3.16GHz Intel Core 2 Duo processor and 4GB of RAM.

We use a set of six meshes, which are all mixed-dimensional tetrahedral meshes, containing explicit triangles and edges, courtesy of CST Computer Simulation Technology AG. Table 1 shows the sizes of these meshes as well as the run times taken by the standalone AHF and MOAB_AHF for constructing the mesh data structure. The overall cost is approximately linear in the number of vertices for both AHF and MOAB_AHF. However, our current implementation of MOAB_AHF takes about 6–7 times longer than AHF, because Algorithm 1 builds the intermediate arrays for v2hes and v2adj, which are more efficient than using MOAB’s built-in function “get_adjacency” to identify sibling half-facets. The cost of constructing MOAB_AHF can be optimized by using Algorithm 1.

Table 1 Times taken to construct data structures by standalone AHF and MOAB_AHF

mesh	#verts	#edges	#tris	#tets	AHF	MOAB_AHF
1	345	121	378	1357	0.002231	0.00969
2	447	137	678	1503	0.002376	0.01433
3	1443	225	1824	6794	0.008991	0.04659
4	1724	2081	3688	8177	0.01218	0.05989
5	2151	282	2556	9746	0.0126219	0.06112
6	119960	27215	42476	711014	0.935569	4.31953

6.2 Storage Costs

We compare the storage requirements of AHF and MOAB. Let C , F_{exp} , E_{exp} and V represent the set of cells, explicit faces, explicit edges and vertices of the given

mesh, and let $|\cdot|$ denote the number of items in a set. AHF stores three maps, which require the following number of entities:

$$\begin{aligned} \text{element connectivity: } n_c &= 2|E_{exp}| + v_f|F_{exp}| + v_c|C| \\ \text{sibling half-facet map: } n_s &= 2|E_{exp}| + s_f|F_{exp}| + f_c|C| \\ \text{vertex to half-facet map: } n_v &= 3|V| \end{aligned}$$

where v_f , v_c , s_f and f_c are the numbers of vertices per face, vertices per cell, edges (sides) per face, and faces per cell, respectively.

Note that for the half-facet ID $\langle eid, lfid \rangle$, we can encode it in a 32-bit integer or store eid and lfid in a 32-bit and a 8-bit integer, respectively. The storage required by the former in bytes is

$$S_{AHF1} = 4(n_c + n_s + n_v) = 16|E_{exp}| + 4(v_f + s_f)|F_{exp}| + 4(v_c + f_c)|C| + 12|V|,$$

whereas the latter requires

$$S_{AHF2} = 4n_c + 5n_s + 5n_v,$$

which is about 12% larger than S_{AHF1} . If the element connectivity is required, the extra storage required by AHF is only about 60% of these.

The storage requirement of MOAB is higher than AHF, as it stores both the connectivity and upward adjacencies. The upward adjacencies are created and stored the first time a query requiring the adjacency is performed. MOAB_AHF may reduce the storage requirement.

To put this into perspective, we also compare its storage against OpenVolumeMesh [13]. In OpenVolumeMesh, all the top-down and bottom-up incidence relations are stored explicitly using integer handles. Let E and F denote the set of all edges and faces (including the implicit edges and faces) of the given mesh. The number of required handles to encode top-down and bottom-up incidences are

$$n_{OVM} = f_c|C| + (v_f + 2)|F| + (s_f + 2)|E| + s_e|V|,$$

where v_f , v_c , s_c and f_c are the average numbers of vertices per face, vertices per cell, edges (sides) per cells, and faces per cell, respectively. Assuming each integer handle is 32-bit, then the storage will be about $S_{OVM} = 4n_{OVM}$. Table 2 shows the storage requirements for the last three meshes in Table 1. As it can be seen from the table, AHF requires about half the amount storage required by OpenVolumeMesh

Table 2 Storage requirements in kilobytes of AHF, MOAB, and OpenVolumeMesh for three largest meshes in Table 1

mesh	AHF		MOAB	MOAB_AHF	OpenVolumeMesh
	integer	struct			
4	435.094	486.955	1039.31	897.78	897.176
5	444.496	496.907	1041.69	904.60	1055.26
6	27857.3	31163.7	64514.35	56889.90	71074.8

and MOAB. MOAB_AHF has reduced the storage of MOAB slightly, but further reduction is still possible.

6.3 Computational Costs of Adjacency Queries

In this subsection, we report the times for the six queries described in Section 4.2. We report the performance results of AHF, MOAB, and MOAB_AHF. We omit OpenVolumeMesh, as it could not load the mixed-dimensional meshes. These

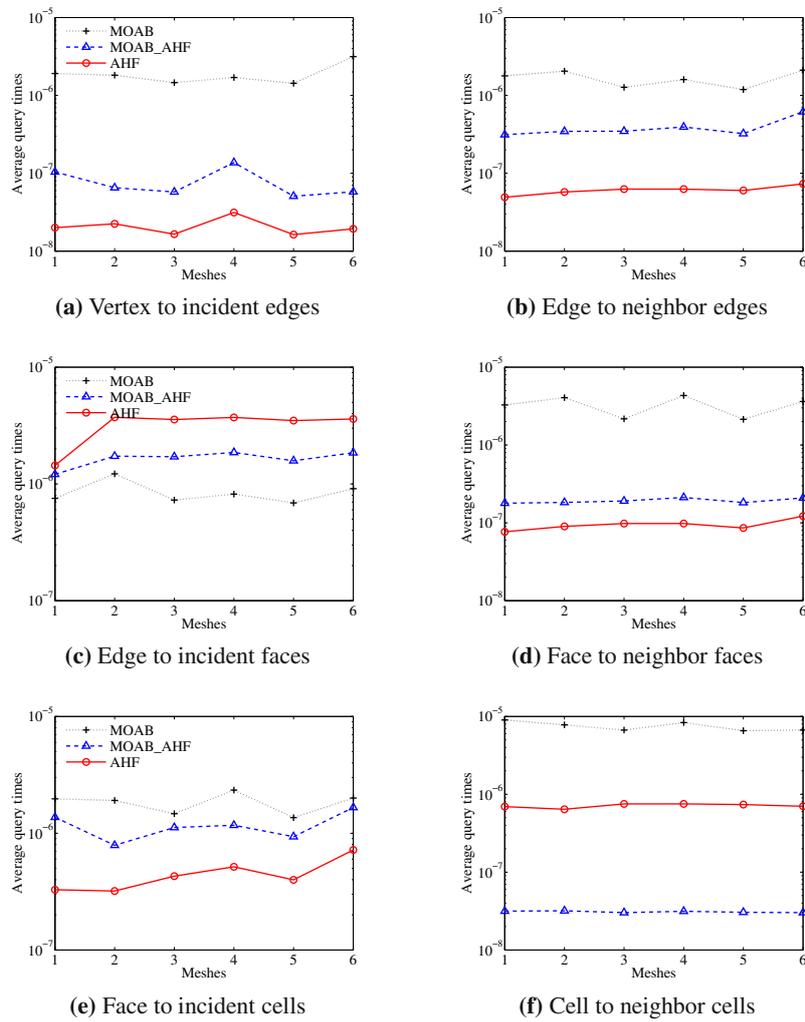


Fig. 7 Average times (in seconds and logarithmic scale) to perform queries in 1-D (a, b), 2-D (c, d) and 3-D (e, f)

queries are performed over explicit entities, and they return explicit entities only. Figure 7 shows the average time taken to perform the incident and neighborhood queries for 1-D, 2-D, and 3-D entities, respectively. The average time is measured by the total elapsed time of the algorithm for all the entities divided by the number of the entities. The results confirm that all the mesh query operations take approximately constant time, regardless of mesh sizes. We observe that AHF and MOAB_AHF have comparable performance, whereas MOAB_AHF significantly outperformed MOAB in nearly all cases (except for the case of Figure 7(c)). This result indicates that the AHF data model can significantly improve the MOAB data model for adjacency queries. Further code optimization of MOAB_AHF can lead to even better performance.

7 Conclusion and Discussions

In this paper, we presented a simple but general array-based half-facet mesh data structure, called *AHF*, for efficient queries and modification of mixed-dimensional and potentially non-manifold meshes. AHF unifies and extends the compact array-based half-edge and half-face data structures [1]. We described our implementation in MATLAB, which allows rapid prototyping, testing, and deployment of meshing algorithms and mesh-based numerical methods, as well as an implementation in C++, called MOAB_AHF, built on top of MOAB. We demonstrated that AHF is efficient in terms of both storage and computational cost. Our preliminary results show that the MOAB_AHF can significantly improve the efficiency of adjacency queries in MOAB while reducing its storage requirements. We plan to release MOAB_AHF as an open-source add-on to MOAB after further improvements and optimization.

Besides its generality and efficiency, AHF has a number of other advantages. In particular, due to its array-based nature, AHF is well suited for parallel computations and is easier to port onto GPUs. In addition, it facilitates easier interoperability with application codes. We plan to explore these issues in our future research.

In its current form, AHF also has some limitations. First, we did not consider polyhedral meshes. However, AHF can be easily extended to support polyhedral meshes by treating them as face-based non-manifold meshes. This is because the faces are composed of polygons, which have a natural numbering convention of their edges. This approach is consistent with the applications of polyhedral meshes, which are typically finite volume methods that require computing the fluxes on the faces. Secondly, we considered only conformal meshes. We plan to explore the extension of AHF to non-conformal meshes in our future work.

Acknowledgements. This work was funded in part under the auspices of the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program of the Office of Nuclear Energy, and the Scientific Discovery through Advanced Computing (SciDAC) program funded by the Office of Science, Advanced Scientific Computing Research, both for the U.S. Department of Energy, under Contract DE-AC02-06CH11357, through a subcontract to Stony

Brook University from Argonne National Laboratory. We acknowledge CST Computer Simulation Technology AG for their generous gift to Stony Brook University and thank Drs. Oleg Skipa and Manuel Baptista of CST AG for providing the meshes used in our experimental tests. Daniel Einstein's contribution was supported by Award Number R01HL073598 from the National Heart, Lung, and Blood Institute

References

1. Alumbaugh, T., Jiao, X.: Compact array-based mesh data structures. In: Proceedings of 14th International Meshing Roundtable, pp. 485–504 (2005)
2. Beall, M.W., Shephard, M.S.: A general topology-based mesh data structure. *Int. J. Numer. Meth. Engrg.* 40, 1573–1596 (1997)
3. Bischoff, B.S., Botsch, M., Steinberg, S., Bischoff, S., Kobbelt, L., Aachen, R.: OpenMesh – a generic and efficient polygon mesh data structure. In: OpenSG Symposium (2002)
4. Canino, D., Floriani, L.D., Weiss, K.: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions. In: *Computer & Graphics Proc. of SMI Conf.*, vol. 35, pp. 747–753 (2011)
5. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: *Computational Geometry: Algorithms and Applications*, 3rd edn. Springer (2008)
6. Dyedov, V., Einstein, D.R., Jiao, X., Kuprat, A.P., Carson, J.P., del Pin, F.: Variational generation of prismatic boundary-layer meshes for biomedical computing. *International Journal for Numerical Methods in Engineering* 79, 907–945 (2009)
7. Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., Schönherr, S.: On the design of CGAL, a computational geometry algorithms library. *Softw. – Pract. Exp.* 30, 1167–1202 (2000); Special Issue on Discrete Algorithm Engineering
8. Garimella, R.V.: MSTK – a flexible infrastructure library for developing mesh based applications. In: Proceedings of 13th International Meshing Roundtable, pp. 213–220 (2004)
9. Juretic, F., Gossman, A.D.: Error analysis of the finite volume method with respect to mesh type. *Numerical Heat Transfer, Part B* 57, 414–439 (2010)
10. Kettner, L.: Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theo. Appl.* 13, 65–90 (1999)
11. Kirk, B.S., Peterson, J.W., Stogner, R.H., Carey, G.F.: libMesh: A c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 22, 237–254 (2006)
12. Kremer, M., Bommers, D., Kobbelt, L.: OpenVolumeMesh – A versatile index-based data structure for 3D polytopal complexes. In: Jiao, X., Weill, J.-C. (eds.) Proceedings of the 21st International Meshing Roundtable, vol. 123, pp. 531–548. Springer, Heidelberg (2013)
13. Poirier, D., Allmaras, S.R., McCarthy, D.R., Smith, M.F., Enomoto, F.Y.: The CGNS system, AIAA Paper 98-3007 (1998)
14. Seol, E.S.: FMDB: Flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis. Ph. D. thesis, Rensselaer Polytechnic Institute (2005)
15. Si, H.: TetGen, a quality tetrahedral mesh generator and three-dimensional Delaunay triangulator v1.4 (2006)

16. Sieger, D., Botsch, M.: Design, implementation, and evaluation of the `surface_mesh` data structure. In: Quadros, W.R. (ed.) Proceedings of the 20th International Meshing Roundtable, vol. 90, pp. 533–550. Springer, Heidelberg (2011)
17. Tautges, T., Meyers, R., Merkle, K.: MOAB: A mesh-oriented database. Technical report, Sandia National Laboratories (2004)
18. The CGNS Steering Sub-committee. The CFD General Notation System Standard Interface Data Structures. AIAA (2002)