AN ALGORITHM ORIENTED MESH DATABASE [†]

Jean-François Remacle[*] and Mark S. Shephard

Scientific Computation Research Center, CII-7011, 110 8th Street, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, U.S.A.

SUMMARY

In this paper, we present a new point of view for efficiently managing general mesh representations. After reviewing some mesh representation basics, we introduce the new Algorithm Oriented Mesh Database (AOMD). Some hypothesis are taken in order to be able to manage any set of adjacencies. Then, we present the design of the AOMD in terms of classes and algorithms. The Standard Template Library (STL) is used for managing the AOMD. Finally, we present some results and discuss technical choices that were made in the AOMD design and implementation. AOMD is available as open source at `http://www.scorec.rpi.edu/AOMD`.

KEY WORDS: Mesh Database, Mesh Representation, C++ STL

1. INTRODUCTION

A mesh M is a geometrical discretization of a domain Ω that consist in

[*]Correspondence to: remacle@scorec.rpi.edu

[†]This work was supported by the ASCI Flash Center at the University of Chicago, under contract B341495 and DOE SciDAC program through agreement No. DE-FC02-01ER25460

- A collection of mesh entities M_i^d of controlled size and distribution;
- Topological relationships or adjacencies forming the graph of the mesh.

Maintaining a complete graph that would consists in all possible relations between all adjacent mesh entities is, of course, not acceptable in both terms of storage and algorithmic complexity. Some *ad hoc* data structures have been developed that only maintain specific sets of adjacencies that are able to fulfill the needs of specific algorithms: mesh generation [1–3], mesh refinement, [4–7] or solution process [8]. In modern adaptive simulation frameworks, mesh generation, partial differential equation solving, even post processing are to be inter-operable components [9]. In such a modern context, a mesh management component should be able to deal with algorithms that have different needs in terms of adjacencies. One way to achieve this goal is to use a so called "complete" set of adjacencies i.e. a mesh representation where every mesh entity can retrieve efficiently every possible set of adjacencies without having to do a global traversal of the graph of the mesh. Reference [10] proposed three complete representations and analyzed them in terms of both memory and efficiency. In this paper, we propose a new approach called an Algorithm Oriented Mesh Database (AOMD) [11,12]. Contrary to [10], the AOMD uses a dynamic mesh representation: the AOMD is able to shape itself to the needs of an algorithm by building an optimum representation and modifying it efficiently when needed.

In section §2, we discuss general issues of mesh representation: mesh entities, downward and upward adjacencies, high order accesses, association with geometric entities and complete representations. We then recall the one level, circular and reduced interior representations presented in [10] and we show how these compare with some *ad hoc* ones in terms of storage or/and efficiency.

In §3, we define basic hypothesis for building AOMD. Mesh entities are described using one set of ordered downward entities. Because AOMD can modify a representation by adding new mesh entities, we must be able to add, delete and ensure uniqueness of mesh entities present in the data structure. For that, we need to be able to compare mesh entities independently of their representation. We chose then a comparison operator for AOMD and explain how to apply it for efficiently supporting the needed operations.

In §4, we describe some software design aspects of AOMD. AOMD is written in `C++`. In recent years, Standard `C++` has added significant new features to the language. One of the most important is the Standard Template Library (STL) [13]. The STL demonstrates that `C++` can be used very efficiently for generic programming techniques [14]. In a mesh database, we need to build data structures for mesh entities and we need some kind of containers for storing and accessing them. In [15,16], authors describe practical implementations of classical containers like lists, binary trees or hash tables. The STL has all those data structures already included. We can use STL containers, algorithms and iterators in an efficient way with the data (i.e. mesh entities) in AOMD. AOMD uses Object Oriented Programming (OOP) for building a hierarchy of classes for mesh entities, M_i^d, and uses the generic paradigm for building STL-like efficient algorithms and iterators.

In §5, we present results of the use of AOMD. We show methods to support periodic boundaries, curved meshes and adaptivity.

2. BASICS OF MESH REPRESENTATION

2.1. Nomenclature

G	the geometric model
M	the mesh model
$\mathsf{T/H}$	tetrahedral/hexahedral mesh
Ω_{V}	domain associated with the model V, $\mathsf{V} = \mathsf{G}, \mathsf{M}$
V_i^d	the ith entity of dimension d in a model V
$\{\mathsf{V}^d\}$	unordered group of topological entities of dimension d in a model V
$[\mathsf{V}^d]$	ordered group of topological entities of dimension d in a model V
$\phi\{\mathsf{V}^d\}$	set of mesh entities of dimension d that are adjacent or contained in a model V. ϕ may be a single entity, a group of entities or a complete model.
$\mathsf{V}_i^d\{\mathsf{V}^q\}$	the unordered group of topological entities of dimension q that are adjacent to the entity V_i^d of model V
$\mathsf{V}_i^d\{\mathsf{V}^q\}_j$	the jth member of the adjacency list $\mathsf{V}_i^d\{\mathsf{V}^q\}$
$\dim \mathsf{V}_i^d\{\mathsf{V}^q\}$	the number of elements in the adjacency list $\mathsf{V}_i^d\{\mathsf{V}^q\}$
$\sqsubset$	classification symbol used to indicate the association of one or more mesh entities from the mesh model, M with an entity of the geometric model G.

2.2. Geometry-based analysis

The goal of an analysis is to solve a set of partial differential equations over a geometrical domain G. The most common way to describe G is to use a boundary-based scheme where the geometric domain is represented as a set of topological types together with adjacencies [17–20]. When the geometric domain G is a manifold, its representation is very simple: in a d-manifold

V, a $(d-1)$-dimensional entity V_i^{d-1} bounds a maximum of two d-dimensional entities. In other cases, the geometric model is not manifold because it's impossible to find a mapping from neighbor points to V_i^{d-1} to $\Re^d$. See, for example, the top image of Figure 1, where the neighborhood of a point, p, on an edge that bounds 3 faces cannot be uniquely isolated by a two-dimensional disk. In fact, it often happens that geometrical models used in the

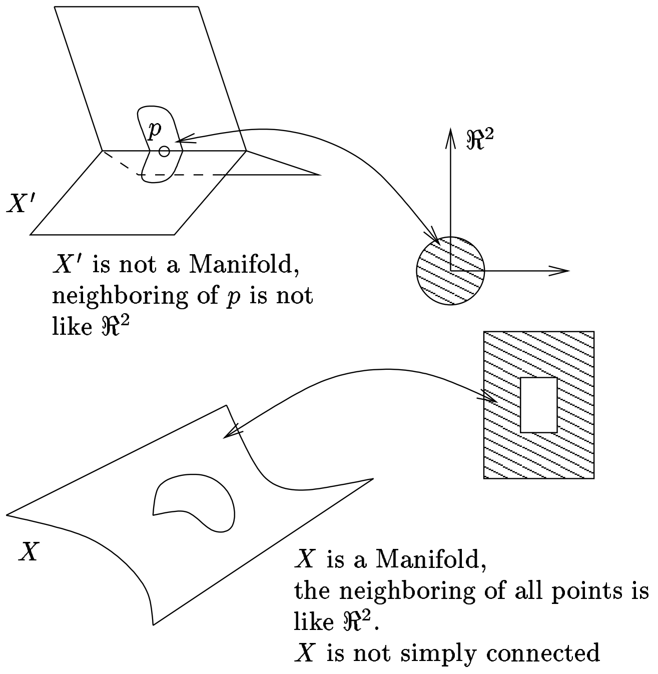

p
$\Re^2$
X'
X' is not a Manifold,
neighboring of p is not
like $\Re^2$
X
X is a Manifold,
the neighboring of all points is
like $\Re^2$.
X is not simply connected

Figure 1. A geometrical object X is a d-manifold when it can be mapped (or transformed) continuously into an open set of $\mathbb{R}^d$ [21]. This simply means that X is locally like $\mathbb{R}^d$. The dimension d of a manifold is a well defined quantity. X is a 2-manifold: all its point have a neighborhood that is like $\Re^2$. X' is not a 2-manifold: points like p have a neighborhood that is not like $\Re^2$. Note that in terms of topology, X is not simply connected.

real engineering world are not manifolds. There exists some non-manifold representation of geometric models [17,20] for representing the classes of relations between the boundary entities. Reference [17] does this by the introduction of *use* structures for the lowest four topological entities of the solid model topological entity hierarchy of region–shell–loop–face–edge–vertex (see upper box of Figure 2).

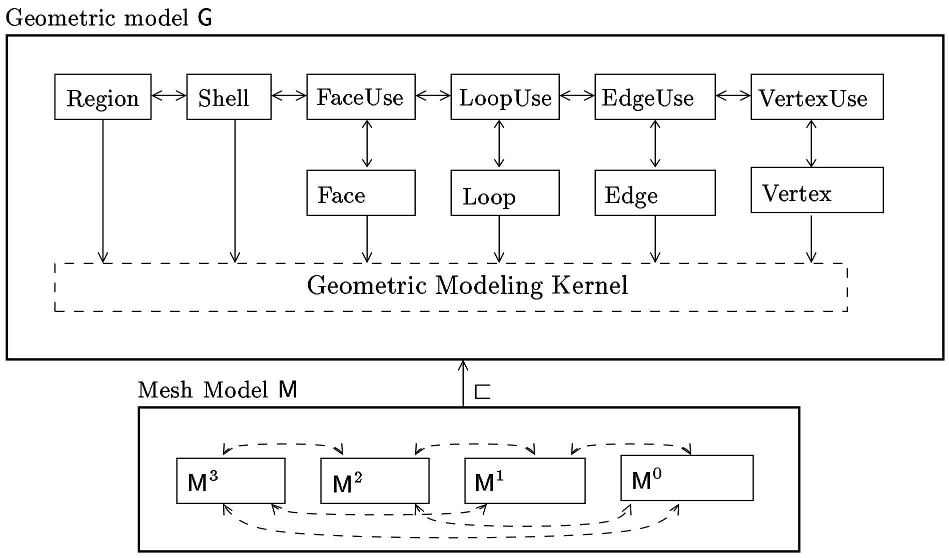


Figure 2. Geometric model G and Mesh model M. We use a version of the radial-edge data structure [17] for G where the topological adjacencies are implemented and interfaced with kernels of solid modelers (ACIS, CATIA, PARASOLID, ProE). Mesh model entities M^d are classified on model entities.

The mesh M is a discrete version of the domain. It consists of a set of mesh entities M_i^d together with adjacencies. There are more mesh entities M_i^d than entities of the geometric model G_i^d. On the other hand, the mesh entities that are employed have limited topological complexity. Mesh entities are topologically equivalent to the unit d-dimensional sphere $S^d = \{x \in \mathbb{R}^d; \|x\|_2 < 1\}$: they are made of one part, they are simply connected (a manifold is said to be simply connected if every closed curve can be smoothly shrunk to a point) and they have no holes. With such simplifications, mesh entities are of 4 distinct topological kinds: *vertices* $\mathsf{M}\{\mathsf{M}_i^0\}$ are the 0-dimensional topological entities, *edges* $\mathsf{M}\{\mathsf{M}_i^1\}$ are the 1-dimensional topological entities, *faces* $\mathsf{M}\{\mathsf{M}_i^2\}$ are 2-dimensional topological entities and *regions* $\mathsf{M}\{\mathsf{M}_i^3\}$ are the 3-dimensional topological entities. Adjacencies in a mesh are described by 4 distinct topological types while the geometrical model needs 6.

Any mesh entity M_i^d is a piece of the discretization of a geometric entity G_j^q, $d \leq q$. We call this association a *classification* of a mesh entity to a geometrical entity and we note it as $\mathsf{M}_i^d \sqsubset \mathsf{G}_j^q$ [10, 22, 23]. Simulation attributes like boundary conditions or material properties are naturally related to model entities and not to mesh entities. In mesh generation and mesh enrichment procedures, the classification information is critical for ensuring that the mesh is constructed so that it improves the geometric approximation of the domain when it is is refined. It is therefore necessary to maintain the classification of mesh entities through all our algorithms.

2.3. Mesh adjacencies

A mesh M is composed of a collection of mesh entities together with their adjacencies. Any mesh entity bounds and/or is bounded by other ones of higher and/or lower dimension. This adjacency information represents the graph of a mesh. For any mesh entity M_i^d, we distinguish two kind of adjacencies:

- Upward adjacencies $\mathsf{M}_i^d\{\mathsf{M}^q\}$ when $q > d$;
- Downward adjacencies $\mathsf{M}_i^d[\mathsf{M}^q]$ when $q < d$.

In 3-D, each mesh entity M_i^d has potentially 3 sets of adjacencies: d downward adjacencies and $3 - d$ upward adjacencies. All these adjacency sets do not need to be present in a given mesh representation. Moreover, some entities may simply not be present in a representation: "interior" edges or faces are not relevant in many situations.

In order to define the mesh representation, we introduce the following simple formalism. We define $\mathcal{I}$ a 4×4 matrix that we will call the *incidence matrix* of the mesh. Diagonal element $\mathcal{I}_{j,j}$ of $\mathcal{I}$ is equal to 1 if mesh entities of dimension j are present in the representation and is equal to 0 if not. Element $\mathcal{I}_{i,j}$ of $\mathcal{I}$ is equal to 1 if the adjacencies from entities of dimension i to entities of dimension j are present. Note that setting an $\mathcal{I}_{i,j} = 1$ is only meaningful when $\mathcal{I}_{i,i} = 1$ and $\mathcal{I}_{j,j} = 1$.

It is interesting at this point to gather some statistics about the average number of adjacencies per entity that occurs in usual three dimensional tetrahedral and hexahedral meshes. With these statistics, we will be able to compute the cost of a given representation i.e. its size in the memory of a computer. Let us call $N^d(\mathsf{M})$ the number of mesh entities of dimension d in a mesh M. Vertices are the entity which is the less numerous in meshes. For that reason, we will use the number of vertices $N^0(\mathsf{M})$ as the unit for evaluating the storage size $\mathcal{C}(\mathcal{I}, \mathsf{M})$ of a given representation $\mathcal{I}$ on a given mesh M. The average number of mesh entities in tetrahedral and hexahedral meshes are presented in Table I.

Tetrahedral Mesh T	Hexahedral Mesh H
$N^3(\mathsf{T}) = 6N^0(\mathsf{T})$	$N^3(\mathsf{H}) = N^0(\mathsf{H})$
$N^2(\mathsf{T}) = 12N^0(\mathsf{T})$	$N^2(\mathsf{H}) = 3N^0(\mathsf{H})$
$N^1(\mathsf{T}) = 7N^0(\mathsf{T})$	$N^1(\mathsf{H}) = 3N^0(\mathsf{H})$

Table I. Relation between number of entities in a mesh.

A second interesting set of statistics concerns the average number of mesh entities of dimension d adjacent to a mesh entity of dimension q. We call this $N^d(\mathsf{M}^q)$. These statistics are represented in Table II.

Tetrahedral Mesh T					Hexahedral Mesh H				
d	3	2	1	0	d	3	2	1	0
$N^3(\mathsf{M}^d)$	1	2	5	23	$N^d(\mathsf{M}^3)$	1	2	4	8
$N^2(\mathsf{M}^d)$	4	1	5	35	$N^d(\mathsf{M}^2)$	6	1	4	12
$N^1(\mathsf{M}^d)$	6	3	1	14	$N^d(\mathsf{M}^1)$	12	4	1	6
$N^0(\mathsf{M}^d)$	4	3	2	1	$N^d(\mathsf{M}^0)$	8	4	2	1

Table II. Average number of adjacencies per entity

We read these tables as follow: most of the tetrahedron meshes will contain, when they are sufficiently big, 6 times more tetrahedron than vertices. Every edge is connected, in average, to 5 tetrahedron. If we want to store all edge-to-tetrahedron adjacencies, we will need $(6\times5)N^0(\mathsf{T})$ pointers. Numbers in Table I and II were presented in Reference [10].

More generally, if we consider that each adjacency has a unit storage, the total storage cost for a given representation can be computed as follow:

$$\mathcal{C}(\mathcal{I},\mathsf{M}) = \sum_{d=0}^{4}\sum_{q=0}^{4} N^d(\mathsf{M})\mathcal{I}_{d,q}N^d(\mathsf{M}^q) \qquad (1)$$

A mesh representation is said to be *complete* if any adjacencies can be retrieved for any mesh entity without a global traversal of the mesh. In the other case, the representations are termed *incomplete*. In a complete representation, any adjacency requires a number of operations that does not depend on the size of the mesh. In an incomplete representation, getting some adjacencies will require a complete traversal of the mesh containers (referred to as linear behavior). It is evident that we cannot afford this traversal each time we ask for an adjacency. Complete representations are then the only acceptable mesh representations if we are to work with a single $\mathcal{I}$ for all the mesh related algorithms.

We will say that a mesh entity M_i^d has a direct access to its adjacencies of dimension q, $\mathsf{M}_i^d\{\mathsf{M}^q\}$, if $\mathcal{I}_{d,q} = 1$. Indirect access to a set of adjacencies occurs when it is possible to use other sets of adjacencies to reach our goal. For example, if our goal is to know the edges of a given region $\mathsf{M}_i^3[\mathsf{M}^1]$, if $\mathcal{I}_{3,2} = 1$ and $\mathcal{I}_{2,1} = 1$, we can get the set of edges of the region by taking the union of the edges in $\mathsf{M}_j^2[\mathsf{M}^1]$ of each of the 4 faces M_j^2 that bounds region M_i^3 and maintaining an appropriate ordering. There is a computational overhead in getting indirect adjacencies. We say that we have a second order access when we have to use one set of intermediary adjacency for getting the information, and a third order access when two intermediary steps are needed.

Let us now describe storage and CPU costs of different usual mesh representations. We first consider a mesh representation that contains the whole set of adjacencies i.e. when the incidence matrix is

$$\mathcal{I}_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{2}$$

would require storage $\mathcal{C}(\mathcal{I}_1, \mathsf{T}) = 388N^0$ or $\mathcal{C}(\mathcal{I}_1, \mathsf{H}) = 120N^0$. This representation is the only one which allows access to all sets of adjacencies as direct adjacencies relations. If we consider that the size of an adjacency is the size of a pointer i.e. 4 bytes, a whole tetrahedron mesh of $100,000$ vertices would require 155.2 Megabytes (Mb) of storage while a whole hexahedron mesh of the same number of vertices would require 48 Mb.

Specific algorithms can have a preferred representation that fit exactly its adjacency information needs. If we consider classical fixed order, fixed mesh, Lagrangian finite elements, they only require element-node connectivity which gives the incidence matrix $\mathcal{I}_2$ of Table III. Storage requirements drop to $C(\mathcal{I}_2, \mathsf{T}) = 31N^0$, i.e., for the same mesh with $N^0 = 100,000$, 12.4 Mb. Of course, representation $\mathcal{I}_2$ cannot fill the needs of every single algorithm that mesh users usually deal with but it is still a very popular mesh representation in the finite element community. Hierarchical finite elements can take advantage of all downward entity sets to be able to define degrees of freedom over a variable order C^0 mesh [24]. In this case, the incidence matrix is $\mathcal{I}_3$ in Table III. We have $C(\mathcal{I}_3, \mathsf{T}) = 232N^0$ or 92.8 Mb for $N^0 = 100,000$.

$$\mathcal{I}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \mathcal{I}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \mathcal{I}'_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$C(\mathcal{I}_2, \mathsf{T}) = 31N^0 \quad C(\mathcal{I}_3, \mathsf{T}) = 232N^0 \quad C(\mathcal{I}'_3, \mathsf{T}) = 100N^0$$
$$C(\mathcal{I}_2, \mathsf{H}) = 10N^0 \quad C(\mathcal{I}_3, \mathsf{H}) = 64N^0 \quad C(\mathcal{I}'_3, \mathsf{H}) = 32N^0$$

$$\mathcal{I}_4 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \mathcal{I}_5 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \mathcal{I}_6 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$C(\mathcal{I}_4, \mathsf{T}) = 103N^0 \quad C(\mathcal{I}_5, \mathsf{T}) = 64N^0 \quad C(\mathcal{I}_6, \mathsf{T}) = 164N^0$$
$$C(\mathcal{I}_4, \mathsf{H}) = 24N^0 \quad C(\mathcal{I}_5, \mathsf{H}) = 18N^0 \quad C(\mathcal{I}_6, \mathsf{H}) = 35N^0$$

Table III. Some classical mesh adjacencies requirements: $\mathcal{I}_2$ for Lagrangian finite elements, $\mathcal{I}_3$ and $\mathcal{I}'_3$ for Hierarchical finite elements, $\mathcal{I}_4$ for Finite Volumes or Discontinuous Galerkin solvers, $\mathcal{I}_5$ for mesh smoothing and $\mathcal{I}_6$ for edge swapping.

If we decide to deal with second and third order access to adjacencies, we can reduce $\mathcal{I}_3$ to a lighter representation $\mathcal{I}'_3$ of Table III with $C(\mathcal{I}'_3, \mathsf{T}) = 100N^0$ or 40 Mb for $N^0 = 100,000$. Discontinuous Galerkin and/or finite volumes only use regions and faces for the calculation, the incidence matrix for such method is $\mathcal{I}_4$ in Table III. We have that $C(\mathcal{I}_4, \mathsf{T}) = 103N^0$ or 40.12 Mb for $N^0 = 100,000$.

Vertex repositioning can be implemented using different strategies. A simple Laplacian smoothing will reposition a vertex at the centroid of the cavity formed by its surrounding regions. In Table III, we have a suitable representation $\mathcal{I}_5$ associated to a Laplacian Smoothing with a storage cost $C(\mathcal{I}_5, \mathsf{T}) = 64N^0$ or 25.6 Mb for $N^0 = 100,000$. A common mesh modification algorithm is the edge swapping where edges have to be able to access regions upward adjacencies. Incidence matrix is then $\mathcal{I}_6$ with $C(\mathcal{I}_5, \mathsf{T}) = 164N^0$ or 65.6 Mb for $N^0 = 100,000$.

Algorithms have obviously very different needs in terms of mesh representation and storage. There are two ways for a mesh data structure to deal with all those algorithms. One first choice would be to use a complete representation. The second way to deal with general mesh representation is to have a mesh data structure that is able to fit to the needs of any algorithm *dynamically*. AOMD takes the second approach.

2.4. Complete representations

Reference [10] presents two complete representations. In the one-level complete representation $\mathcal{I}_{ol}$, each entity M_i^d is connected to entities of dimension $d-1$ and $d+1$. This leads to the incidence matrix $\mathcal{I}_{ol}$ presented in Table IV. The second one, the circular representation $\mathcal{I}_{circ}$, is also presented in Table IV.

$$\mathcal{I}_{ol} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \mathcal{I}_{circ} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$C(\mathcal{I}_{ol}, \mathsf{T}) = 163N^0 \qquad C(\mathcal{I}_{circ}, \mathsf{T}) = 123N^0$$
$$C(\mathcal{I}_{ol}, \mathsf{H}) = 56N^0 \qquad C(\mathcal{I}_{circ}, \mathsf{H}) = 33N^0$$

Table IV. Some complete representations.

Ad hoc representations like ones listed in Figure III are usually designed in order to avoid second and third order access to adjacencies that would be required to construct specific adjacencies, that may be heavily used in an algorithm, from complete representations like the one-level or circular. To understand the desirability of avoiding these second and third order adjacencies, a third set of statistics is introduced at this point. If we consider that the direct access to an adjacency has a unit computational cost, we can evaluate the over cost of second and third order accesses by counting how much time we have to access an intermediary adjacency in order to retrieve the information. For example, if we consider the circular representation, it is possible to know the set of vertices of a tetrahedron M_i^3. For that, we must first ask the 4 faces of the tetrahedron, for each face, ask its edges and for each edge, ask its vertices. The total computational cost is then $4 \times 3 \times 2 = 24$ which is 6 times more than the direct access. Note that we have not counted the time for making the list of vertices unique here. The design of our AOMD can make that operation efficient. These results are listed in Table V for the two previous complete representations. Complete representations have

$$\mathcal{O}(\mathcal{I}_{ol}, \mathsf{T}) = \begin{bmatrix} 1 & 1 & 2 & 6 \\ 1 & 1 & 1 & 2 \\ 3 & 1 & 1 & 1 \\ 6 & 2 & 1 & 1 \end{bmatrix} \quad \mathcal{O}(\mathcal{I}_{circ}, \mathsf{T}) = \begin{bmatrix} 1 & 69 & 9.2 & 1 \\ 1 & 1 & 36.8 & 2.6 \\ 3 & 1 & 1 & 19.7 \\ 6 & 2 & 1 & 1 \end{bmatrix}$$

Table V. Computational overhead due to second and third order adjacencies. $\mathcal{O}(\mathcal{I}, \mathsf{M})_{ij}$ is the ratio between the number of accesses needed for retrieving $\mathsf{M}_k^i\{\mathsf{M}^j\}$ using representation $\mathcal{I}$ and the number of accesses using a direct access i.e. $N^i(\mathsf{M}^j)$.

storage requirements that are larger than *ad hoc* representations of Table III. Table V show that complete mesh representations may also lead to non neglectible computational overheads, even for such a common adjacency query like getting vertices of a region.

2.5. Minimum information: a functionaly complete representation

An important question is what is the minimum amount of data we need to be able to build up all entities with their full set of adjacencies and classification. All vertices are to be present in all representations ($\mathcal{I}_{0,0} = 1$). A *sufficient minimum* of data is that any mesh entity *equally classified* has to be present in the representation. It means that all entities M_i^d for which we have $\mathsf{M}_i^d \sqsubset G_j^d$ are to be present and classified if we want the ability to properly construct a representation with any $\mathcal{I}_{i,j} = 1,\ i,j = 0,1,2,3$ for general non-manifold geometric domain. Note that all vertices are to be present in the representation but only ones that are classified on model vertices need to be classified. With the minimum of information we have defined, all the other ones may be classified using the following algorithm:

- Take all classified edges $\mathsf{M}_i^1 \sqsubset G_j^1$ and classify their unclassified vertices to G_j^1;
- Do the same for classified faces and regions and all vertices will be classified.

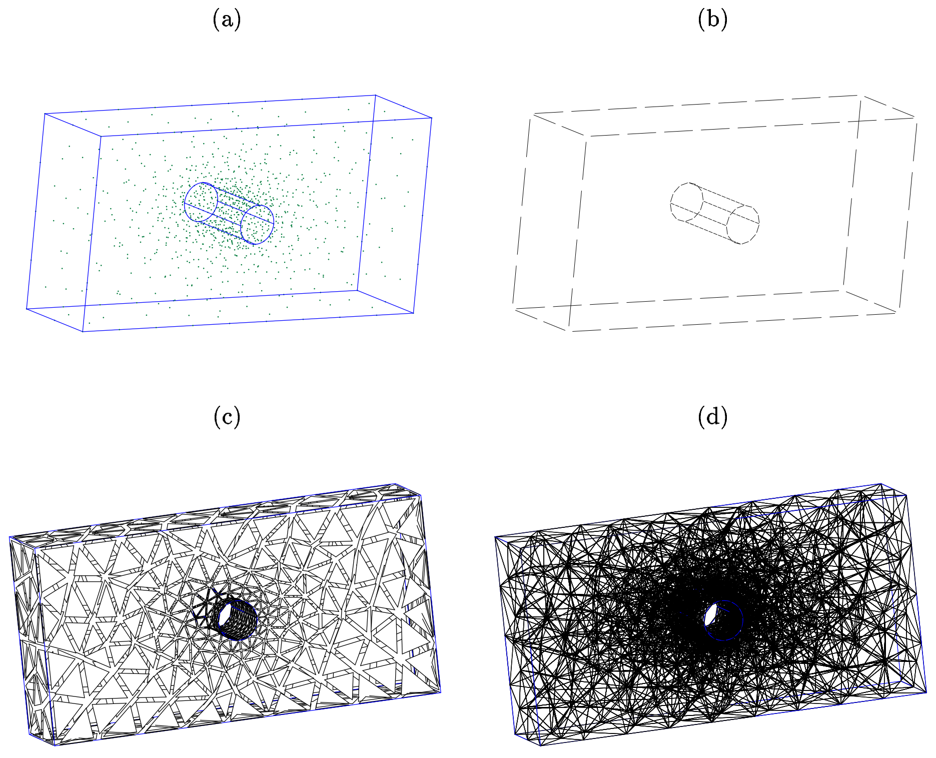


Figure 3. Minimum representation: all vertices (a), all edges classified on model edges (b), all faces classified on model faces (c) and all regions (d).

The algorithm for building up all entities is straightforward. For a $3-D$ mesh,

- Take all regions $\mathsf{M}_i^3 \sqsubset G_j^3$, create faces $\mathsf{M}_i^3[\mathsf{M}^2]$ using regions and classify all new faces to G_j^3;
- Then, take all faces $\mathsf{M}_i^2 \sqsubset G_j^d$, create faces edges $\mathsf{M}_i^2[\mathsf{M}^1]$ using faces and classify new edges to G_j^d;
- Finally, take all regions $\mathsf{M}_i^3 \sqsubset G_j^3$, create region edges $\mathsf{M}_i^2[\mathsf{M}^1]$ using regions and classify

new edges to G^3_j.

If the minimum information is not available, it is possible to recover classification information using geometrical criteria. However, these geometric operations require comparisons that can provide non-unique answers, especially if the underlying geometric model is non-manifold. Therefore, there is always a risk of obtaining incorrect classifications when using such checks. Our goal is to avoid this problem and only use topological information.

Figure 3 gives a graphical depiction of the minimum information. Figure 3a shows the model edges, G^1, and the mesh vertices M^0. Figure 3b shows the mesh edges, M^1 classified on model edges, Figure 3c shows the mesh faces, M^1 classified on model faces and Figure 3d shows the mesh regions.

It is possible to construct a functionally complete representation by storing appropriate adjacencies with the entities of the minimal set of information. The term *functionally complete* is introduced since interior entities $\mathsf{M}^d_i \sqsubset \mathsf{G}^q_j,\ 0 < d < q$ are not present, but can be uniquely determined and operated on within the algorithms by keying them from higher order entities. The reduced interior representation of Reference [10] is an example of a *functionally complete* representation that maintains slightly more than the minimum set (the $\mathsf{M}^1_i \sqsubset \mathsf{G}^2_j$ are also maintained in that representation).

3. MESH REPRESENTATION IN AOMD

The aim of the AOMD is to support the specific set of adjacencies, complete or not, needed by each application. For that aim, we need to be able to start from the minimum mesh representation (see §2.5 above) and generate specific adjacencies for the current application.

This implies that we want to be able to create mesh entities “from scratch”, to add some non-existant adjacencies lists to any mesh entity or to delete unneeded ones.

3.1. Mesh Entity Description

Any mesh entity has to be described by an ordered set of mesh entities of lower dimension, $\mathsf{M}_i^d[\mathsf{M}^k]$ with $k < d$. Regions may be defined by either faces, edges or vertices, faces by edges or vertices, and edges by vertices. The vertex is then the atomistic, *self consistent* entity. To be able to differentiate vertices, we do not use coordinates because a mesh is considered here as a purely topological object. We attribute an unique `iD` to each vertex for differentiating them. We denote $\mathrm{id}(\mathsf{M}_i^0)$ the function that takes a vertex as a parameter and that returns its `iD`.

3.2. Mesh Entities Comparison

Hypothesis 3.1 says that entities are to be represented using at least one set of entities of lower dimensions. We use this hypothesis to build up an equal operator for mesh entities that will remain valid in any mesh entity representation. This operator is critical to perform queries in a data structure. For vertices, we have already dealt with this issue: two vertices M_i^0 and M_j^0 are equal if $\mathrm{id}(\mathsf{M}_i^0) = \mathrm{id}(\mathsf{M}_j^0)$. The second hypothesis is that *two entities that have the same vertices are equal.* Because mesh entities are always defined using lower order entities, it is always possible to obtain their representation in terms of vertices. For example, if a region is defined using its faces, faces are defined using either vertices or edges. If the faces are defined using edges, these edges are always defined using vertices so that we can always access vertices from any representation. The current equal operator has an additional restriction relative to the more flexible possibility of [10]. Figure 4 shows the case of a circle

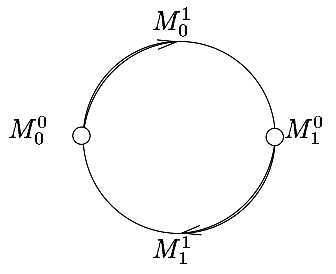
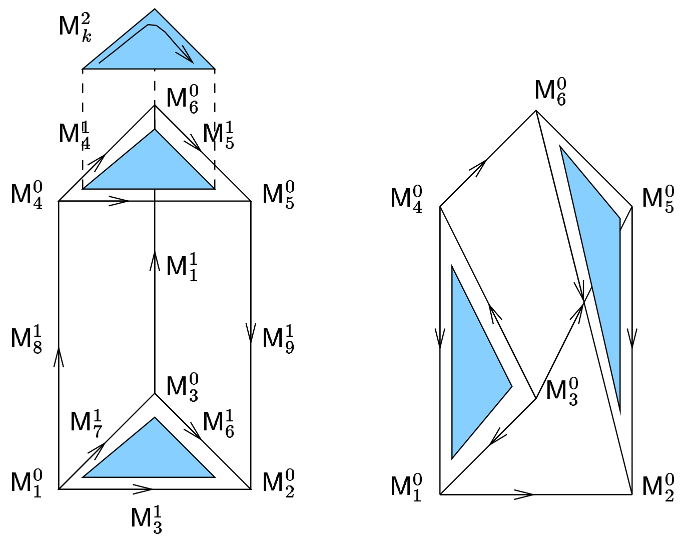


Figure 4. Pathological case of two mesh edges for the discretization of a periodic model edge (circle)

meshed using two curved edges M_0^1 and M_1^1 that should be topologically distinct but, in our hypothesis, are equal because they are bounded by the same vertices. If we restrict ourselves to meshes that enforce having at least three mesh edges for any closed curve and for any loop, our hypothesis is valid. We can have the same similar issue at face level, two faces could share same edges (imagine a sphere bounded by only two mesh faces). Note that this equal operator has the very important advantage that two mesh entities can alway be compared, even if their representations are different. For example we are able to compare two regions, one defined by edges and one defined by faces.

3.3. Downward Adjacencies Ordering: Templates

The boundary of a mesh entity M_i^d is defined as its adjacency set of dimension $d-1$. When a mesh entity M_i^d has an access to its boundary like in $\mathcal{I}_{ol}$ or $\mathcal{I}_{circ}$, it is possible to retrieve the rest of downward adjacencies without ambiguity. With AOMD, we use a weaker hypothesis: a mesh entity M_i^d may be described using any set of downward adjacencies $\mathsf{M}_i^d\{\mathsf{M}^q\}$, $q < d$. This information does not always lead to a unique mesh entity unless the set $\mathsf{M}_i^d\{\mathsf{M}^q\}$ enforces a predefined *ordering* of $\mathsf{M}_i^d\{\mathsf{M}^q\}$. As an illustration, we consider the wedges of figure 5. These two mesh regions are different while having the same set of vertices.



$$\mathsf{M}^3_i[\mathsf{M}^0] = [\mathsf{M}^0_3, \mathsf{M}^0_1, \mathsf{M}^0_2, \mathsf{M}^0_6, \mathsf{M}^0_4, \mathsf{M}^0_5] \qquad \mathsf{M}^3_j[\mathsf{M}^0] = [\mathsf{M}^0_3, \mathsf{M}^0_4, \mathsf{M}^0_1, \mathsf{M}^0_5, \mathsf{M}^0_6, \mathsf{M}^0_2]$$

Figure 5. Two prisms with vertices that are ordered differently. The "edge-vertex" template described below leads to the definition of different set of edges (and faces) with same set of vertices. Opposite triangular faces of the prism are drawn in both cases.

Templates A convention is then definied for ensuring uniqueness of the representation. We define a set of tables that describes local relationships between downward entities of a same mesh entity. Two tables are needed for each distinct mesh entity type. The first table describes all the edges of a mesh entity in terms of its vertices. This table is called the "edge-vertex template". The second table describes faces of an entity in terms of edges, it is the "face-edge template". Another useful template is the "face-vertex template" which is, by definition, the composition of the two previous ones. The "edge-vertex template" of the prism can be written in the matrix form as :

$$T^{ev} = \begin{bmatrix} 1 & 1 & 2 & 4 & 4 & 5 & 1 & 2 & 3 \\ 2 & 3 & 3 & 5 & 6 & 6 & 4 & 5 & 6 \end{bmatrix}^T$$

Elements $T^{ev}_{i,1}$ and $T^{ev}_{i,2}$ gives vertices indices of the ith edge of the wedge. These indices are local, they represent a position in the vertex adjacency list of the current wedge. For example, the second edge of wedge M^3_i of Figure 5 is defined by the first and third vertices: $\mathsf{M}^i_3[\mathsf{M}^0]_1 = \mathsf{M}^0_3$ and $\mathsf{M}^i_3[\mathsf{M}^0]_3 = \mathsf{M}^0_2$ so that $\mathsf{M}^i_3[\mathsf{M}^1]_2 = \mathsf{M}^1_6$. In wedge M^3_j, there is no edge connecting M^0_3 and M^0_2. Note that the 9th edge of M^3_i is connecting its 3rd and 6th vertices i.e. M^0_2 and M^0_5. This edge M^1_9 exists but is connecting M^0_5 and M^0_2 which means that wedge M^3_i *uses* edge M^1_9 negatively. This use of orientation information is critical in algorithms such as higher order hierarchical finite elements [24]. The use of templates allow the computation of orientations on demand. This is an advantage when we compare to [10] where authors are storing uses in their data structures. The "face-edge template" can be written in the matrix form as :

$$T^{fe} = \begin{bmatrix} 1 & 2 & 3 & \times \\ 4 & 5 & 6 & \times \\ 7 & 4 & 8 & 1 \\ 8 & 6 & 9 & 3 \\ 7 & 2 & 9 & 5 \end{bmatrix} \tag{4}$$

In a prism, two of the faces are triangles. That's why some elements of T^{fe} are noted with $\times$ when they are not relevant. The "face-vertex template" T^{fv} of the prism is computed as the composition of T^{ev} and T^{fe}:

$$T^{fv} = \begin{bmatrix} 1 & 2 & 3 & \times \\ 4 & 5 & 6 & \times \\ 1 & 4 & 5 & 2 \\ 2 & 5 & 6 & 3 \\ 1 & 3 & 6 & 4 \end{bmatrix} \qquad (5)$$

For example, the second face of wedge M_i^3 of Figure 5 is defined by the *4th*, *5th* and *6th* vertices i.e. $\mathsf{M}_3^i[\mathsf{M}^0]_4 = \mathsf{M}_6^0$, $\mathsf{M}_3^i[\mathsf{M}^0]_5 = \mathsf{M}_4^0$ and $\mathsf{M}_3^i[\mathsf{M}^0]_6 = \mathsf{M}_5^0$ so that $\mathsf{M}_3^i[\mathsf{M}^2]_2 = \mathsf{M}_k^2$ which ordered set of vertices is $[\mathsf{M}_4^0, \mathsf{M}_6^0, \mathsf{M}_5^0]$. M_i^3 *uses* this face negatively because we need an odd permutation of the set $[\mathsf{M}_6^0, \mathsf{M}_4^0, \mathsf{M}_5^0]$ to get M_k^2. We can also use templates to compute face orientations.

Inverse templates We define now the useful notion of inverse templates. Inverse templates are inverse mapping of templates. The first inverse template table describes all the vertices of a mesh entity in terms of its edges pair. This table is called the "vertex-edge template". For the wedge, it is written:

$$T^{ve} = \begin{bmatrix} 1 & 1 & 2 & 4 & 4 & 5 \\ 2 & 3 & 3 & 5 & 6 & 6 \end{bmatrix}^T \qquad (6)$$

It means, for example, that the first vertex of the prism is the one common to its first and second edges. Because of the hypothesis of §3.2, two edges can have only one vertex in common so that speaking about the only common vertex of two edges is meaningful. That's also why inverse templates are not unique: the first vertex of the prism is also the one common to edges 1 and 7. Similarly, we can define a "edge-face template" that describes pairs of faces sharing an edge:

$$T^{ef} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 4 \\ 3 & 5 & 4 & 3 & 5 & 5 \end{bmatrix}^T \tag{7}$$

Finally, we define the "vertex-face" template T^{vf} which is the composition of the T^{ve} and T^{ef}. We know by (6) that vertex 1 is common to edges 1 and 2. By (7), we know that edge 1 is common to faces 1 and 3 and edge 2 is common to faces 1 and 5. This implies that vertex 1 is common to faces 1, 2 and 5. We can continue that reasoning for all vertices to obtain:

$$T^{vf} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & 3 & 4 & 3 & 3 & 4 \\ 5 & 4 & 5 & 5 & 4 & 5 \end{bmatrix}^T$$

4. DESIGN OF AOMD

One important aspect of building a mesh database is its software design. The language used here is C++. The C++ language is evolving and one of its recent major new feature is the Standard Template Library (STL) [13]. The STL demonstrates that C++ can be used very efficiently for generic programming techniques. Generic programming is "programming with concepts", where a concept is defined as a family of *abstractions* that are all related by a common set of *requirements.* The design of generic software components consists of concept development – identifying sets of requirements that are general enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family. STL provides a set of easily composable software components of six major kinds: generic algorithms, containers, iterators, function objects or functors, adaptors, and allocators. In each of these component categories, STL provides a relatively small set of fundamental components; it is through uniformity of interfaces and orthogonality of component structure that STL provides functionality far beyond the actual number of components included.

AOMD provides its own data types for mesh entities and uses STL containers for storing mesh entities and adjacencies. STL iterators are used for accessing information. In what follows, only the principal features of the different classes of AOMD are presented.

4.1. Mesh entities

The mesh entity base class is described in Figure 6 and its inheritance diagram is presented in Figure 7. In 3-D, each mesh entity M_i^d has potentially 3 sets of adjacencies. Our mesh entities

```
class mEntity {
public:
  typedef std::vector<mEntity*> mAdjacencyContainer;
  typedef mAdjacencyContainer::const_iterator iterator;
  mEntity ( const mAdjacencyContainer & , gEntity *);
  virtual ~mEntity();
  inline iterator   begin(int dim);
  inline iterator   end  (int dim);
  inline iterator   find (mEntity*);
  inline void push_back  (mEntity*);
  inline void erase      (mEntity*);
  // Purely virtual members
  virtual int dimension () = 0;
  virtual mEntity* getTemplate(int ith ,
             int i_dim, int j_dim ) = 0;
  virtual int size (int dim) = 0;
private :
  int iD;
  gEntity *classification;
  mAdjacencyContainer *adjacency[4];
};
```

Figure 6. Mesh entity class description.

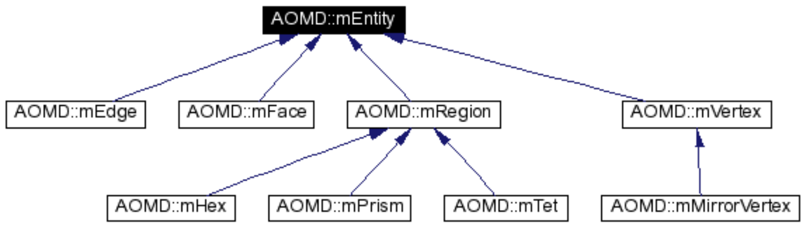


Figure 7. Inheritance diagram for the class `mEntity`.

have four pointers to containers of mesh entities for storing adjacencies. We have decided to use the `std::vector` for storing adjacencies, even if query operations in vectors are linear. There are several good reasons to use vectors in this case:

- The number of elements in an adjacency list is usually small. We have found that it was more efficient to use a `std::vector` than a `std::set` for containers of less than $\mathcal{O}(100)$ elements;
- The `std::vector` is a data structure which has small memory needs;
- The `std::vector` provides random access to its elements. This is needed for building AOMD templates.

Constructor `mEntity::mEntity` takes as input a list of downward entities and the geometrical entity `gEntity` upon which the mesh entity is classified. Mesh entity `iD` is, for the moment, uniquely relevant for vertices. We will extend it in §4.2 to higher order entities. Iterators `begin(int dim)` and `end(int dim)` are provided to iterate on adjacencies of dimension `dim`. Pure virtual members of class `mEntity` (Figure 6) are added to the `class mEntity`. Function `dimension()` returns the dimension of the mesh entity concerned (1 for edges for example). Member `size(int dim)` returns the number of entities of dimension `dim` for the mesh entity. Calling function `size(1)` in case of a hexahedron will simply return 12 which is the number of edges in an hexahedron. Finally, member function `getTemplate(int ith,int i_dim,int j_dim)` returns the `ith` mesh entity of dimension `i_dim` using adjacencies of dimension `j_dim`. For example, `mPrism::getTemplate(4,2,0)` will return a quadrilateral face with vertices 2, 5, 6 and 3 of the prism according to the "face-vertex" template of Equation (5). Note that, if the prism does not have vertices in its adjacency list, it has either faces or edges (or both). Then, AOMD will use invert templates for retrieving needed vertices so that the template operation will always be successful.

4.2. Mesh entity container

Sets of mesh entities $\mathsf{M}\{\mathsf{M}^d\}$ are not static: mesh adaptation/optimization operators add and remove entities from the database. In the case of a mesh, the number of entities is not limited to a small number like for adjacencies. Therefore, we need to use an efficient data structure for storing mesh entities.

In AOMD, we have chosen hash tables to store large sets of mesh entities. The elements of a hash table are not guaranteed to be in any meaningful order; in particular, they are not sorted (sets of mesh entities $\mathsf{M}\{\mathsf{M}^d\}$ are unordered by definition). The worst case complexity of most operations on hash table is linear in the size of the container, but the average case complexity is constant time; this means that for applications where values are simply stored and retrieved, and where ordering is unimportant, hash tables are usually much faster than sorted associative containers like binary trees [25]. The `class mEntityContainer` is described in Figure 8.

Remark. The STL `hash_set` is an SGI extension; it is not part of the `C++` standard. Nevertheless, we use the STLport (www.stlport.org) that provides a portable STL with all

```
class mEntityContainer
{
public :
  typedef std::hash_set<mEntity*, mHashFct, mEqualKey>
          container_;
  typedef container_::const_iterator iterator;
  iterator begin(int dim);
  iterator end(int dim);
  iterator find(mEntity*);
  void add(mEntity*);
  void del(mEntity*);
private :
  container entities[4];
};
```

Figure 8. Mesh entities container class description

SGI extensions. We have found that the STLport implementation of the STL is usually more efficient than, e.g., the one provided by `gcc`.

A hash function `mHashFct` maps its argument (a mesh entity) to a result of type `size_t`. A Hash Function must be deterministic and stateless. That is, the return value must depend only on the argument, and equal arguments must yield equal results. A mesh entity hash function must represent the mesh entity in the sense that we have defined in §3: if two mesh entities have the same set of vertices, then the result of the `mHashFct` should return the same value. For that reason, the mesh entity hash function cannot be, for example, the address in memory of the `mEntity` or an integer given by a `static` counter inside the mesh entity class.

A mesh vertex has an `iD`. The hash function for a vertex is, of course, returning this `iD`. We extend now `iD`'s for all higher order mesh entities in order to be used a hash function. This mesh entity `iD` must fullfill the following properties:

- Two mesh entities that share same vertices (i.e. that are equal) have the same `iD`: $\mathsf{M}_i^d = \mathsf{M}_j^d \rightarrow \mathrm{id}(\mathsf{M}_i^d) = \mathrm{id}(\mathsf{M}_j^d)$. However, two different mesh entities of dimension $d > 0$

may share the same iD, they may be differentiated in this case using a more complex equal operator defined in §3.2 that will check if all the vertices are equal;

- Mesh entity iD is a symmetric function of vertices iD's i.e. it is unchanged by permutations in the set of vertices;
- Calculation of mesh entity iD must not lead to integer overflow: in the case of big meshes, do not use square functions of the vertex iD's for example ;
- Mesh entities iD's should be dense i.e., for a given M_i^d, few entities M_j^d verify $\mathrm{id}(\mathsf{M}_i^d) = \mathrm{id}(\mathsf{M}_j^d)$.

Queries in a hash table are made in two steps. First, the hash function is used to get a list of mesh entities that share the same iD. This operation is constant i.e. it does not depend on the size of the hash table. Then, we iterate on the list and we use the binary predicate `mEqualKey` which takes 2 mesh entites M_i^d and M_j^d as arguments and that returns `true` if M_i^d and M_j^d have the same set of vertices. This operation is linear in the number of mesh entities that have the same iD and evaluation of the predicate `mEqualKey` is, in our case, rather expensive. Mesh entity iD's will then be considered as an efficient hash function if few different entities have identical iD's. In section §4.4, we will present different choices for the mesh entitiy iD's and compare them.

4.3. Algorithms for Mesh Representation Modifications

A function object, or functor (the two terms are synonymous) is simply any object that can be called as if it is a function. An ordinary function is a function object, and so is a function pointer; more generally, so is an object of a class that defines `operator()`.

We may want to add upward adjacencies of dimension `adj` to all entities of dimension `dim` of `mEntityContainer` `m`. For that, we use the standard `std::for_each` algorithms provided by STL and the unary functor of Figure 9.

```
struct createUpwardFunctor {
  int dim;
  createUpwardFunctor (int d) : dim (d) {}
  inline void operator () (mEntity *ent) const
  {
    mEntity::iterator it    = ent->begin(dim);
    mEntity::iterator itend = ent->end(dim);
    for ( ; it != itend ; ++it) (*it)->push_back(ent)
}};
```

Figure 9. Upward adjacencies creation functor. Mesh entity `ent` accesses all its adjacencies `*it` of dimension `dim` and adds itself to all of them.

An algorithm that creates edge-to-region adjacencies for the whole set of entities in a `mEntityContainer` is described in Figure 10. Note that this algorithm can be applied to a

```
mEntityContainer mesh;
...
std::for_each (mesh.begin(3),mesh.end(3),
               createUpwardFunctor(1));
```

Figure 10. Upward adjacencies creation algorithm. The algorithm acts on range of entities provided by the mesh entities container.

reduced range of mesh entities which could be useful to restore connectivities when some local mesh modifications are made or when the users wants to build some adjacencies on parts of the whole mesh.

We may also need to build downward adjacencies of dimension `dim` to a range of mesh entities. For that, we use functor depicted in Figure 11.

An algorithm that creates region-to-face adjacencies for the whole set of entities in a `mEntityContainer` is described in Figure 12.

In this algorithm, we traverse all entities of dimension 3, we create, using templates, a set

```
struct createDownwardFunctor {
  int i_dim, j_dim;
  mEntityContainer *cont;
  createDownwardFunctor (int i, int j, mEntityContainer *c)
      : i_dim (i), j_dim(j), cont(c) {}
  inline void operator () (mEntity *ent) {
    for ( int i=0; i<ent->size(i_dim); i++){
      mEntity *temp = ent->getTemplate (i,i_dim,j_dim);
      mEntityContainer::iterator found = cont->find(temp);
      if(it != cont->end(i_dim){
        delete temp;
        ent->add(*found);
      }
      else {
        ent->add(temp);
        cont->add(temp);
}}}};
```

Figure 11. Downward adjacencies creation functor. Mesh entity `ent` builds all templates `temp` of dimension `dim` and checks if the entity is present in the container `cont`. If the entity `*found` exists, then we add `*found` into adjacency of `*ent` and we delete `temp`. If not, we add `temp` into adjacency of `ent` and in `cont`.

```
mEntityContainer mesh;
...
std::for_each (mesh.begin(3),mesh.end(3),
          createDownwardFunctor(2,0,&mesh));
```

Figure 12. Downward adjacencies creation algorithm. The algorithm acts on range of entities provided by the mesh entities container.

of downward adjacencies of dimension 2 using vertices and we look into container `cont` for an equal entity `*temp`. Both algorithms require only one traversal of the container and, if the `find` operator has a constant complexity, both algorithms are linear in data sizes. Note that these are the only two algorithms we need for mesh representation modifications and the total number of lines of codes is less than 15.

4.4. Choice of the Mesh Entity `iD`

In the algorithm of Figure 12, functor `createDownwardFunctor` does many queries. One measure of the efficiency of the query operations in a hash table is $\psi = \frac{N_e}{N_k}$ where N_e is the number of elements in the hash table and where N_k is the number of different values for the hash function: ψ is proportional to the average number of operations needed for a query operation. Ideally, $\psi = \mathcal{O}(1)$ so that query operations show constant behavior.

The efficiency of queries in AOMD depends on the choice of the mesh entity `iD`. There are a number of possible choices for calculating mesh entity `iD` $\text{id}(\mathsf{M}_i^d)$. We present three different choices $\text{id}_j(\mathsf{M}_i^d)$, $j = 1, \ldots, 3$. We call $\psi(\text{id}_i, \mathsf{M}^d)$ the efficiency factor for a hash table containing entities of dimension d and using id_i for computing mesh entity `iD`'s. In what follows, $n = \dim \mathsf{M}_i^d(\mathsf{M}^0)$.

- **Choice 1:**

$$\text{id}_1(\mathsf{M}_i^d) = \sum_{j=1}^{n} \frac{\text{id}(\mathsf{M}_i^d[\mathsf{M}^0]_j)}{n};$$

- **Choice 2:**

$$\text{id}_2(\mathsf{M}_i^d) = \max_{j=1,\ldots,n} \text{id}(\mathsf{M}_i^d[\mathsf{M}^0]_j);$$

- **Choice 3:**

$$\text{id}_3(\mathsf{M}_i^d) = \sum_{j=1}^{n} \frac{\mathcal{R}(\text{id}(\mathsf{M}_i^d[\mathsf{M}^0]_j))}{n}.$$

In id_3, $\mathcal{R}$ is a random number generator that takes the vertex `iD` as its seed ($\mathcal{R}(i)$ is a *function* of i i.e. $i = j \to \mathcal{R}(i) = \mathcal{R}(j)$).

Table VI show computations of ψ on several tetrahedral and hexahedral meshes. All meshes were initially loaded with their minimum representations. Then, the algorithm of figure 12 was

M	T_1	T_2	T_3	T_4	H_1	H_2	H_3
$\dim\{M^0\}$	15440	86813	184713	548141	15625	42875	166375
$\dim\{M^1\}$	95324	556103	1203286	3630889	45000	124943	490050
$\dim\{M^2\}$	155773	922968	2009605	6104220	43200	121380	481140
$\dim\{M^3\}$	75888	453677	991031	3021471	13824	39304	157464
$\psi(id_1, M^1)$	3.62452	3.69172	3.72524	3.79182	2.58621	2.69173	2.79613
$\psi(id_2, M^1)$	8.19282	8.38176	8.48276	8.51623	2.88148	2.91483	2.94560
$\psi(id_3, M^1)$	1.00002	1.00006	1.00009	1.00013	1.00000	1.00002	1.00004
$\psi(id_1, M^2)$	5.67555	5.78667	5.89832	5.81028	2.38279	2.59326	2.71983
$\psi(id_2, M^2)$	16.2891	16.3899	16.3933	16.3400	2.81598	2.85795	2.90314
$\psi(id_3, M^2)$	1.00002	1.00006	1.00011	1.00015	1.00000	1.00000	1.00003

Table VI. Computations of ψ for tetrahedral meshes T_i, $i = 1, ..., 4$ and hexahedral meshes H_j, $j = 1, ..., 3$

used to build edges and faces from regions and vertices.

Comparing $\psi(\mathrm{id}_3, M^d)$, $\psi(\mathrm{id}_2, M^d)$ and $\psi(\mathrm{id}_1, M^d)$ shows that computing `iD`'s using id_3 is by far the best choice for all meshes. Only one operation is needed for finding an entity in the database. It has taken 77.5 seconds to create all $2,009,605$ faces of mesh T_3 using id_2 while only 22.3 seconds were needed using id_3. These factors are in good relation with $\psi(\mathrm{id}_i, M^2)$ of table VI. Another reason why we should use id_3 is that we will never find special cases where $\psi(\mathrm{id}_2, M^d) \gg 1$ for which hash tables will be no longer efficient. Such an example is depicted in Figure 13.

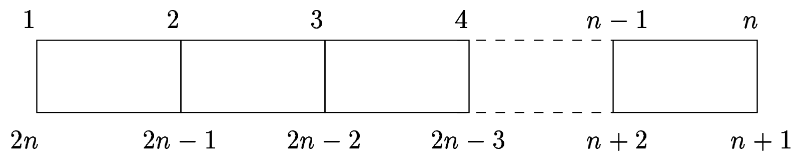


Figure 13. Vertex ordering where all vertical edges have the sum of vertices `iD's` equal to $2n+1$.

5. APPLICATIONS

This section provides three examples of the application of AOMD to three advanced mesh representation situations.

5.1. Higher Order Finite Elements

Trellis [26] is a computational framework developed at the Scientific Computation Research Center at Rensselaer. We present the example of an elliptic problem computed with the higher-order finite element capabilities of Trellis in which the polynomial order of shape functions are associated to mesh entities of different dimensions [24]. Table VII indicates the number of shape functions for a given order p for different mesh entity types. First order finite elements

M_i^d	vertex modes	edge modes	face modes	region modes
triangle	1	$p-1$	$\frac{(p-1)(p-2)}{2}$	0
quadrilateral	1	$p-1$	$(p-1)^2$	0
tetrahedron	1	$p-1$	$\frac{(p-1)(p-2)}{2}$	$\frac{(p-1)(p-2)(p-3)}{6}$
hexahedron	1	$p-1$	$(p-1)^2$	$(p-1)^3$

Table VII. Number of shape functions (or modes) associated to each vertex, edge, face and region of a mesh entity M_i^d as a function of the polynomial order p of the finite element approximation.

only require vertex modes so that we can use the minimum representation. Second order finite elements require edge modes in all cases so that we then need to build the edges. For hexahedral meshes, second order finite elements require faces. For a polynomial order $p \geq 3$ we need the whole set of mesh entities. We have solved a simple Poisson equation $\Delta u = 1$ on Ω_{M}, $u = 0$ on $\partial\Omega_{\mathsf{M}}$ using mesh T_1 of Table VI (Figure 14) and finite elements of orders $p = 1, ..., 5$. In a tetrahedral mesh T with an order p basis, it is easy to see, using Tables VII, I and II, that the number of equations in the system is $p^3 N^0(\mathsf{T})$ and that the bandwith of the matrix grows linearly with p. If we use the Compressed Sparse Row (CSR) format for storing the finite element matrix, then the amount of storage needed is related to the number of non zero elements in the matrix. In case of hierarchical finite elements on tetrahedral meshes, we have just showed that this number grows like $p^4 N^0(\mathsf{T})$. In Table VIII, we compare the storage cost of the finite element matrix and the one of the mesh. We have used both AOMD

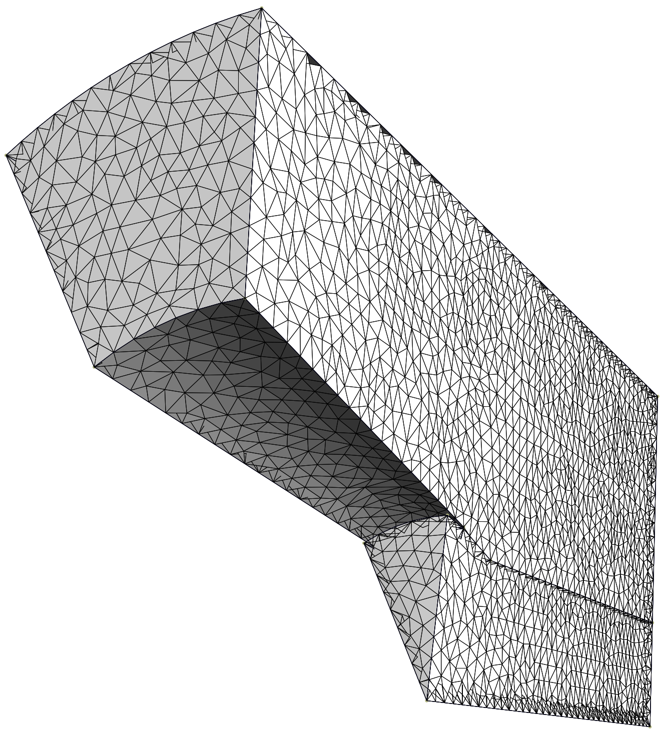


Figure 14. Mesh T_1.

p	1	2	3	4	5
Nb. Unknowns	11343	94303	324766	778620	1531753
Bandwidth	11.28	24.54	43.30	68.83	88.01
Nb. Non Zero	128007	2313905	14065562	53599856	134812122
Matrix Storage (Mb)	2.07	20	121	462	1162
AOMD Mesh Storage (Mb)	8.2	20.1	35	35	35
$\mathcal{I}_{ol}$ Mesh Storage (Mb)	44	44	44	44	44

Table VIII. Number of unknowns and number of non zeros in the finite element matrix associated to a Poisson problem on mesh T_1.

dynamical representation and the one-level structure $\mathcal{I}_{ol}$. For $p = 1$, we see that the mesh requires more memory that what we need for building the finite element matrix. For $p = 2$, AOMD mesh and finite element matrix have nearly the same memory needs: around 20 Mb.

The $\mathcal{I}_{ol}$ representation has constant needs of 44 Mb. For $p \geq 3$, the mesh size is smaller than the matrix. It is clear that the flexibility of AOMD allows one to select the level of storage required for the specific p, thus saving substantial memory costs, while still maintaining all the power of an advanced mesh database.

5.2. Curved Meshes

The usual way to deal with curved mesh entities is to define new sets of topological entities that have more nodes [27]: 6-nodes triangles, 10-nodes tetrahedron, etc. We have not chosen this approach in AOMD. A curved triangle has 3 corners and 3 edges and middle-edge nodes are viewed as an attribute of the edge that qualifies its shape. Doing this allows other qualifiers such as different polynomial approximations or a pointer to the true geometry [28]. Another reason not to use middle nodes is related to the implementation. One of the major advances of the STL is the introduction of an orthogonal decomposition of components. Topology and geometry of mesh entities are orthogonal concepts. We have

- Topological mesh entity types: triangles, tetrahedrons, etc.
- Geometries for mesh entities: straight-sided, Lagrange interpolation, Bézier interpolation, exact mapping to the geometry, etc.

In our approach, all triangles M_i^2 have 3 vertices (or corners) and it is possible to chose different geometries for M_i^2 and its boundary edges $\mathsf{M}_i^2[\mathsf{M}^1]$. For dealing with the various mesh geometries, we have defined the notion of `Mapping` that contains relevant geometry information for mesh entities. Mappings provides the shape information for mesh entities. A mapping can be evaluated i.e. giving a point ξ in the reference system, the mapping is able to evaluate world coordinates $x(\xi)$ of point ξ. Mappings are also able to do the invert operation (which may lead

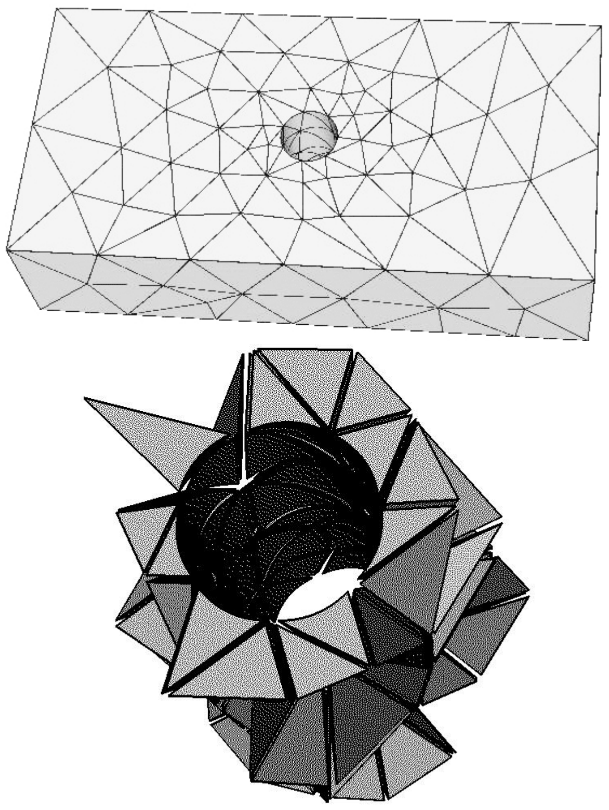


Figure 15. Curved Mesh (top) and view of the set of curved tetrahedron (bottom).

to the resolution of a system of non linear equations). Mapping also provides the Jacobian $\frac{\partial x(\xi)}{\partial \xi}$ of the change of coordinates.

Mesh entities are curved when they have one edge or face on curved model boundaries. Interior mesh entities are usually straight-sided. However, they may be curved in cases when it is required to maintain the shape of an element it bounds due to the curving of other bounding entities. Curved mappings are usually expensive to compute. Moreover, integration of functions on curved elements require more effort than for straight-sided ones. In AOMD, we are able to define different mappings for different mesh entities through a `Builder` pattern. The system checks if shape attributes are associated to the mesh entity and returns either a curved or a straight-sided mapping. We show an example of curved mesh in Figure 15.

5.3. Periodicity

Just as there is substantial flexibility in the way shape information can be associated with the mesh entities in AOMD, analysis information is also treated in a flexible manner. A set of parameters of particular interest to the analysis process are the equation numbers of the finite element system (to be referred to as DOF for degrees of freedom). The DOF have an association with the mesh entities. As discussed in reference [10], topological mesh data structures classified against non-manifold geometric models can support the mesh representation needs of such models without the explicit introduction of the use entities at the mesh level. There are a number of specific analysis modeling situations where some information is best associated with what corresponds to use entities in the model. An obvious example of this is two bodies in contact. At the level of the geometric model a contact surface is associated with a single face. However, at the analysis model level, it is necessary to account for the possibility of relative slip of the regions on the two sides of the contact face by having independent DOF associated with the two uses of the face (one each for the two regions) for each mesh entity for which there are DOF. The structures of AOMD easily supports this need.

An additional, less obvious, such capability easily supported in AOMD is the effective accounting for periodicity in “matched meshes” by assigning the same DOF to the matched mesh vertices, edges and faces which have DOF on the periodic boundaries. From a topological model point of view, this is the compliment of the contact situation in the following way: The geometry of the matched mesh vertices, edges and faces are associated with the two face uses with the coordinates of the one set equal to that of the other plus one period. The degrees of

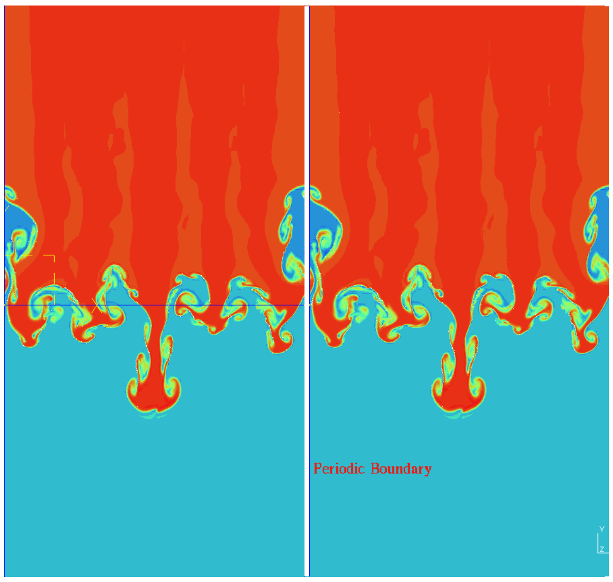


Figure 16. Simulation of a fluid instability. Density colormap of the fluid is replicated twice in order to illustrate the periodicity in the flow.

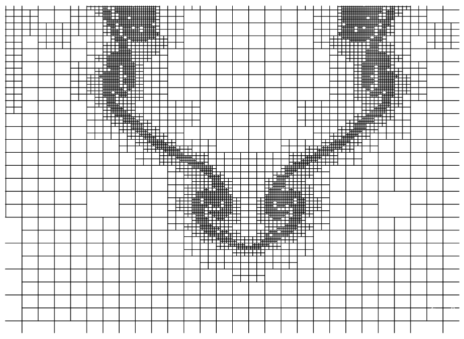


Figure 17. Non-conforming mesh refinement at the vicinity of a fluid spike.

freedom are associated with the single topological face so that there is only one set of DOF, even though the mesh entities using the two face uses have different coordinate locations.

In [29], we have developed an adaptive discontinuous Galerkin method (DGM) for solving non-linear conservation laws. We have used AOMD as the mesh tool for our DGM. A Rayleigh-Taylor instability involves a heavy fluid overlying a light fluid [30,31]. We consider two inviscid fluids initially in hydrostatic (unstable) equilibrium in a cavity. The upper half of the cavity is filled with a fluid of density two while the lower part is filled with a fluid of unit density. The initial pressure corresponds to hydrostatic equilibrium. An initial perturbation of the velocity initiates the instability. The flow is governed by the Euler equations of gas dynamics. We present a result for a 2-D Rayleigh Taylor instability with one vertical periodic boundary (Figure 16). We have used the non-conforming adaptivity capabilities of AOMD to refine the mesh, as depicted in Figure 17.

6. CONCLUSIONS

A model of mesh representation that is able to manage any adjacency sets has been presented. Specific hypothesis were made on mesh entities (equality operator, identificator) and on the mesh itself (minimum representation) for insuring coherence of the whole mesh database. The resuling database is light and efficient in terms of compilation and memory use. In addition to some specific improvements, current efforts are focused on a parallel version of AOMD. We will completely take into advantage the structure of AOMD in parallel [11]. AOMD is available for the community at `http://www.scorec.rpi.edu/AOMD`.

REFERENCES

1. R. Löhner, Some useful data structures for the generation of unstructured grids, Comm. appl. numer. methods 4 (1997) 123–135.
2. H. Dannelongue, P. Tanguy, Efficient data structure for adaptive remeshing with fem, Journal of Computational Physics 91 (1990) 94–109.
3. F. Noel, J. J. C. Leon, P. Trompette, Data structures dedicated to an integrated free-form surface meshing environment, Computers And Structures 57(2) (1991) 345–355.

4. G. F. Carey, M. Sharma, K. C. Wang, A class of data structures for 2-d and 3-d adaptive mesh refienement, International Journal for Numerical Methods in Engineering 26 (1997) 2607–2622.
5. D. J. Mavriplis, Adaptive meshing techniques for viscous flow calculations on mixed element unstructured meshes, International Journal for Numerical Methods in Fluids 34(2) (2000) 93–111.
6. Y. Kallinderis, P. Vijayan, Adaptive refinement-coarsening schemes for three-dimensional unstructured meshes, AIAA J. 31 (1993) 1440–1447.
7. R. Biswas, R. C. Strawn, Tetrahedral and hexahedral mesh adaptation for cfd problems, Applied Numerical Mathematics 21(1-2) (1998) 135–151.
8. D. Hawken, P. Townsend, M. Webster, The use of dynamic data structures in finite element applications, International Journal for Numerical Methods in Engineering 33 (1992) 1795–811.
9. Terascale simulation tools and technology center `url=http://tstt-scidac.org/` (2001).
10. M. W. Beall, M. S. Shephard, A general topology-based mesh data structure, International Journal for Numerical Methods in Engineering 40 (1997) 1573–1596.
11. J.-F. Remacle, O. Klaas, J. E. Flaherty, M. S. Shephard, A parallel algorithm oriented mesh database, to appear in Engineering With Computers .
12. J.-F. Remacle, B. K. Karamete, M. S. Shephard, Algorithm oriented mesh database, in: Ninth International Meshing Roundtable, 2000, pp. 349–359.
13. D. R. Musser, A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Addison-Wesley, 1996.
14. M. H. Austern, Generic Programming and the STL. Using and Extending the C++ Standard Template Library, Addison-Wesley, 1998.
15. J. Thompson, B. K. Soni, N. P. Weatherill (Eds.), Handbook of Grid Generation, CRC Press, 1999, Ch. 14.
16. X. Bonet, J. Peraire, An alternated digital tree (ADT) algorithm for 3d geometric searching and intersection problems, International Journal for Numerical Methods in Engineering 31 (1991) 1–17.
17. K. Weiler, The radial-edge structure: a topological representation for non-manifold geometric boundary representations, in: M. Wosny, H. McLaughlin, J. Encarnacao (Eds.), Geometric Modeling for CAD Applications, North-Holland, 1988, pp. 3–36.
18. A. A. G. Requicha, J. R. Rossignac, Solid modeling and beyond, IEEE Computer Graphics and Applications 12(5) (1992) 31–44.
19. C. M. Hoffman, J. R. Rossignac, A road map to solid modeling, IEEE Trans. Visualization and Computer Graphics 2(1) (1996) 2–10.

20. J. R. Rossignac, Structured topological complexes: A feature-based api for non-manifold topologies, in: Sym. on Solid Modeling and Applications, 1997, pp. 1–9.

21. W. Burke, Applied Differential Geometry, Cambridge, 1991.

22. M. S. Shephard, The specification of physical attribute information for engineering analysis, Engineering With Computers 4 (1988) 145–155.

23. M. S. Shephard, M. K. George, Reliability of automatic 3-d mesh generation, Computer Methods in Applied Mechanics and Engineering 101 (1992) 443–462.

24. M. S. Shephard, S. Dey, J. E. Flaherty, A straightforward structure to construct shape functions for variable p-order meshes, Computer Methods in Applied Mechanics and Engineering 147 (1997) 209–223.

25. D. E. Knuth, The Art of Computer Programming. Vol 3, Sorting and Searching, Addison-Wesley, 1973.

26. M. W. Beall, M. S. Shephard, An object-oriented framework for reliable numerical simulations, Engineering With Computers 15 (1999) 61–72.

27. T. J. R. Hughes, The Finite Element Method. Linear Static and Dynamic Finite Element Analysis, Prentice-Hall, 1987.

28. S. Dey, M. S. Shephard, J. E. Flaherty, Geometry-based issues associated with p-version finite element computations, Computer Methods in Applied Mechanics and Engineering 150 (1997) 39–55.

29. J.-F. Remacle, J. E. Flaherty, M. S. Shephard, An efficient local time stepping scheme in adaptive transient computations, Submitted to SIAM J. Sci. Comput. .

30. Y.-N. Young, H. Tufo, A. Dubey, R. Rosner, On the miscible Rayleigh-Taylor instability, Under consideration for publication in J. Fluid Mech. .

31. J. Glimm, J. Grove, X. Li, W. Oh, D. C. Tan, The dynamics of bubble growth for Rayleigh-Taylor unstable interfaces, Phys. Fluids 31 (1988) 447–465.