Algorithm Oriented Mesh Database

Jean-Francois Remacle *†, B. Kaan Karamete and Mark S. Shephard

Rensselaer Polytechnic Institute, Troy, New York, USA.

Abstract

In this paper, we present a new point of view for efficiently managing general mesh representations. After having given 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 preliminary results and discuss technical choices that were made in the AOMD design and implementation.

1 Introduction

The aim of this paper is to present a new approch to managing the topological relationships needed from a mesh data structure to meet the needs of multiple applications: Partial Differential Equation (PDE) solvers, mesh generators, post processing viewers... In adaptive computations, mesh generation and PDE solving (and even post processing) are to be used together so that we want to use the same mesh database for all our algorithms. One solution to the problem is to store all possible adjacencies. This is not acceptable in terms of memory size and algorithmic complexity. Alternative approaches are to store a specific sets of adjacenties that ensure all others can be evaluated [2]. In this paper, a new approach which maintains most of the advantages of the other approaches without incurring the disadvantages will be presented. We call this new approach the Algorithm Oriented Mesh Database (AOMD) because we have the ability to shape the AOMD to the needs of the algorithms. The AOMD builds on the capabilities of the Standard Template Library or STL [6, 7]. The paper will discuss how the construction and implementation of the AOMD is being accomplished and present preliminary results on its effectiveness.

*This work was partly supported by AFOSR, as part of the ARPA/SFOSR Consortium for Crystal Growth Research, under grant No. F49620-95-1-0407. It was also partially suported by the DOE as part of the ASCI Flash Center at the University of Chicago, under contract B341495.

†Email: remacle@scorec.rpi.edu

2 Basics of Mesh Representation

We distinguish geometrical objects with respect to their dimension d. A geometrical object is said to be n-dimensional if there exists a map of this object to an open subset of R^n. Our three dimensional world leads then to four different kinds of geometrical objects. The interactions of these objects are best represented by the abstraction of topological entities and their adjacencies. The basic topological entities are vertices (0-dimensional objects, $d = 0$), edges (1-dimensional objects, $d = 1$), faces (2-dimensional objects, $d = 2$) and regions (3-dimensional objects, $d = 3$).

Modern solid modelers are using a boundary representation in which an entity G^d of dimension d is described by its boundaries which are of dimension $d-1$. The boundary representation of a model (topology) and the geometry (shape) of model entities are two separated notions.

A mesh is a discretization of a geometrical do-

main, that consists of mesh entities of controlled size and distribution that have very simple topology (hexaedron, tetrahedron...). The topology of a mesh is described with adjacencies between mesh entities. Meshes are used for scientific computation. Physics i.e. material properties, boundary conditions are to be prescribed on the geometrical model which is the most natural representation of the reality [4]. This is one of the reasons why we have to maintain both representations of a model (solid model and the mesh) and why we have to maintain a direct link between every mesh entity M_i^d and the geometrical entity G_j^q (with $q \geq d$) it is discretizing. We call this association a *classification* of a mesh entity to a geometrical entity and we note it $M_i^d \sqsubset G_j^q$. The geometry of mesh entities is not contained in our representation. This has the advantage that different geometrical representations can be used for the same mesh. The mesh topology contains only adjacency informations (even not vertices locations). The mesh geometry contain either vertices location and any other geometrical representation of mesh entities (e.g. the exact geometry, this is another possible use for classification).

2.1 Adjacencies

Any mesh entity bounds and/or is bounded by other ones of different dimensions. This adjacency information represents the graph of a mesh. For any mesh entity M_i^d, we distinguish two kind of adjacencies:

- Unordered upward adjacencies $M_i^d[M^k]$ when $k > d$
- Ordered downward adjacencies $M_i^d\{M^k\}$ when $k < d$

We distinguish adjacencies in this way because sets of downward and upward adjacencies are inherently different.

2.1.1 Upward Adjacencies

Upward adjacencies sets are unordered in the sense that the ordering of the set has no importance. For example, the first and second edge adjacent to a vertex can be interverted with no consequencies. This means also that asking for the i^{th} edge surrounding a vertex is not a meaningful question. The only interresting way to access upward adjacency sets is a continuous traversal, not a random access. Another particularity of upward sets is that their size is variable and unknown a priori.

2.1.2 Downward Adjacencies

Downward adjacencies are of fixed size and random access is needed. These two very different behavior of downward and upward sets will lead to a different design for their implementation.

2.2 Mesh Representations

We introduce the following simple formalism. We define I a 4×4 matrix that we will call the *incidence matrix* of the mesh. Diagonal element I_{jj} of I is equal to 1 if mesh entities of dimension j are present in the representation and is equal to 0 if not. Element I_{ij} of I is equal to 1 if the adjacencies from entities of dimension i to entities of dimension j are present.

A mesh representation is said to be complete if any adjacencies can be retieved 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 information requires 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 have to work with a single I for all the mesh related algorithms. In [2], we were using a complete representation that we called bi-directional in the sense that every entity M_i^d had both $M_i^d\{M^{i-1}\}$ and $M_i^d[M^{i+1}]$ adjacencies set. In this case,

$$I = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad (1)$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Nodal Finite Elements

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Hierarchical Finite Elements

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Discontinuous Galerkin

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Edge Swapping

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Laplacian Smoothing (1)

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Laplacian Smoothing (2)

Table 1: Specific representation for specific algorithms

Another complete representation is the circular one [2] where

$$I = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{2}$$

Irrespective of the completeness of the mesh representations, specific algorithms can have a preferred representation that fits exactly its adjacency information needs. For example, classical fixed order, fixed mesh, lagrangian finite elements only requires element-node connectivity. 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 [3]. Discontinuous Galerkin (and finite volumes) only uses regions and faces for the calculation. Mesh algorithms like smoothing (vertex repositioning) or edge swapping needs also specific sets of adjacencies (see Table 1 for a summary). Using a complete representation will fullfill the needs of all above algorithms. However, the best representations with respect to operation count and memory are varying from algorithm to algorithm as shown on table 1. The aim of AOMD is to be able to fit the mesh representation to particular algorithms. Each algorithms "knows" a priori the set of adjacencies it will access. So, from any sufficient initial representation, the AOMD builds up the optimum set of adjacency. The AOMD builds on the capabilities of the STL.

3 Basics of the STL

STL is a `C++` library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a `template`. Like many class libraries, the STL includes container classes: classes whose purpose is to contain other objects. The STL includes the classes `vector`, `list`, `set`, `hash_set`... Each of these classes is a `template`, and can be instantiated to contain any type of object. The STL also includes a large collection of algorithms that manipulate the data stored in containers. STL provides *sort*, *invert*, *search* algorithms that are not members of container classes but who are generic. STL iterators are the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container In our presentation, we use three of the STL containers:

- `vector<>` – can be used in much the same way as you would use an ordinary C array, except that `vector` eliminates the chore of managing dynamic memory allocation by hand. The `vector` provides constant-time methods for moving forward and backward in arbitrary-sized steps. In other terms, we have *random access iterators* for vectors.

- `set<>` – is a container that supports comparable objects i.e. objects on which we can define a *less than operator*. The `set` provides `find`, `add` and `delete` operators which complexity is

always logarithmic. The `set` provides constant-time methods for moving one step forward and backward (*bidirectional iterator*). Elements of the `set` are sorted.

- `hash_set<>` – basically a hash table. Elements in a hash table have to provide a *hash value* which is usually an integer valued function and an *equal operator* that is a boolean function asserting if two elements are equal or not. A hash table is able to provide constant `delete`, `add` and `find` operators *if the hash function provided by its elements is sufficiently dense* (i.e. if few elements in the hash table share the same hash value). Accessing an item in a hash table is a two step operation: (i) first, the hash function is used to access in one operation a linked list of elements that have the same hash value (ii) then, a linear search is made inside the linked list using the equal operator For applications where values are simply stored and retrieved, and where ordering is unimportant, hash tables are usually much faster than sets.

4 Algorithm Oriented Mesh Database

The aim of the AOMD is support the specific set of adjacencies, complete or not, needed by each application. For that aim, we need to be able to start from a minimum mesh representation (we will define what we mean by minimum representation) to generate any other representation. 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 some unneeded ones.

4.1 Mesh Entity Description

Any mesh entity has to be described by a set of mesh entities of lower dimension, $M_i^d\{M^k\}$ 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. For being 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 note $id(M_i^0)$ the function that takes a vertex as parameter and that returns its `iD`.

4.2 Mesh Entities Comparison

Hypothesis 4.1 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. For vertices, we have already dealt with this issue: two vertices M_i^0 and M_j^0 are equal if $id(M_i^0) = id(M_j^0)$. 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. Current equal operator has an additional restriction relative to the more flexible possibility of [2]. Figure 1 shows the case of a 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 enforces to have at least three mesh edges for any closed curve, 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.

4.3 Downward Adjacencies Ordering: Templates

Our third hypothesis for AOMD is a convention for downward adjacencies ordering. Downward adjacencies sets are ordered, this implies that *two different*

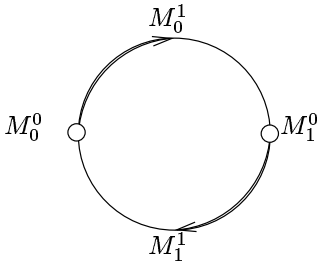

Figure 1: Pathological case of two mesh edges for teh discretization of a periodic model edge (circle)

ordering of the same downward set may lead to two different entities. Note that this is a limitation, some previous work on mesh representations were not needing such convention. This convention allows the definition of a set of tables that describes local relationships between downward entities of a same mesh entity. Two tables are needed for each topologically distinct mesh entities. 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 compostion of the two previous ones. As an example, we consider the tetrahedron of figure 2. A tetrahedron M_j^3 is a mesh entity with 4 vertices $M_j^3\{M^0\}$, 6 edges $M_j^3\{M^1\}$ and 4 faces $M_j^3\{M^2\}$. The "edge-vertex template" can be written in the matrix form as :

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

Elements T_{i1}^{ev} and T_{i2}^{ev} gives vertices indices of the i^{th} edge of the tetrahedron. These indices are local, they represent a position in the vertex adjacency list of the current tetrahedron. For example, the second edge of the tetrahedron is defined by the first and third vertices: $M_2^1[M_1^0, M_3^0]$. The "face-edge template" can be written in the matrix form as :

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

Representations may be wrong because they are not coherent with templates. If we define the "edge-vertex" template of the prism as :

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

The following representation:

$$M^3 = \{M_7^1, M_6^1, M_3^1, M_1^1, M_8^1, M_9^1, M_4^1, M_5^1, M_2^1\}$$

of the prism of figure 3 is correct and we can easily recover the ordered set of vertices using templates. For example, the first vertex of the prism is the one shared by its first (M_7^1), second (M_6^1) and fourth (M_1^1) edges. So, it is vertex M_3^0. We can continue and find the new coherent representation of the prism in terms of vertices:

$$M^3 = \{M_3^0, M_1^0, M_2^0, M_6^0, M_5^0, M_4^0\} \quad (4)$$

The representation:

$$M^3 = \{M_7^1, M_6^1, M_3^1, M_8^1, \dots\}$$

is wrong because the fourth edge of the wedge has to be the one that shares a vertex with the first and second edges and it is not the case in this representation. The template convention is crucial in mesh entities other than simplices. In simplices, the set of edges is the set of all combination of two vertices. The faces are the all the combinations of three vertices (or of three edges). Every representation of a simplex is coherent with this default template. For non-simplices, this is not the case. All combinations of two vertices are not necessary forming an edge so

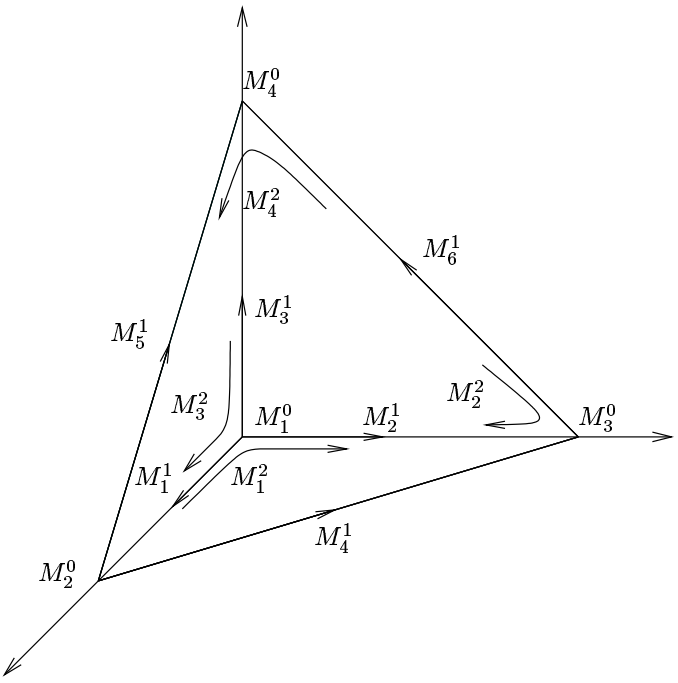

Figure 2: Topology of a tetrahedron

that two different representations using the same vertices can represent two different mesh entities. For the prism, representation (4) and the following one

$$M^3 = \left\{ M_3^0, M_4^0, M_1^0, M_5^0, M_6^0, M_2^0 \right\} \tag{5}$$

leads to the two different prisms (with different edges and faces) of figure 3.

4.4 Inverse templates

We define now the very interresting notion of inverse templates. Inverse templates are related to templates, they are in fact inverse mapping of templates. The first invert 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 tetrahedron of figure 2, we write:

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

It means, for example, that the first vertex of the tetrahedron is the one common to its first and second edge. Similarly, we can define a "edge-face template" that describes pairs of faces sharing an edge:

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

We can define then 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 2. This implies that vertex 1 is common to faces 1, 2 and 3. We can continue that reasoning for all vertices to obtain:

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

Templates and inverse templates are of a great use for AOMD. From any mesh entity representation, we

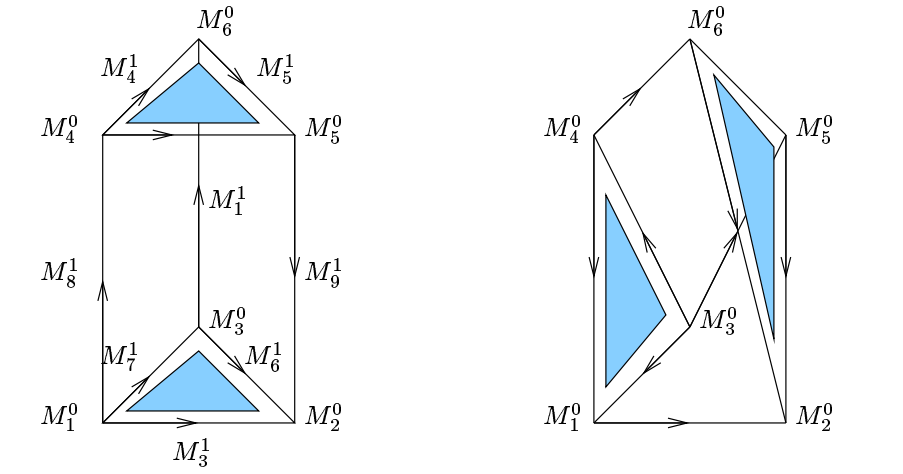

Figure 3: Two prisms with vertices that are ordered differently. The "edge-vertex" template (3) leads to the definition of different set of edges (and faces) with same set of vertices. Opposite triangular faces of the prism are drawn in both cases

are able to build up any other one using local operators (i.e. without looking through all the mesh). We will show how templates will be used for modifying mesh representations in a very straightforward way in section 5.4.

4.5 Orientations

A direct consequence of hypothesis 4.3 is that sets of vertices are always ordered because they are always downward sets. Two entities that have the same set of vertices are equal due to 4.2. Differential geometry [1] asserts that some manifolds are orientable. Our very simple mesh entities are of course of that kind. Only two orientation are possible: we use 1 for a positive orientation and 0 for a negative one. When a mesh entity M_i^d is first created, we take as a convention that its orientation is positive. If, at a certain stage, we have to deal with an entity M_j^d that have the same vertices than M_i^d, orientation of M_j^d may be positive or negative, depending if an even or odd permutation of M_j^d vertices is necessary to recover M_i^d set of vertices. In case of figure 3, the use of templates gives us a first edge $M_j^0\{M_3^0, M_1^0\}$ and there is an edge M_7^1 that has the same vertices. Edge M_7^1 is then the first edge of the prism because it has the good set of vertices but the prisms *uses* M_7^1 negatively. It is easy compute efficiently "on the fly" all the edge uses using templates. In case of the prism, we have $U^1(M^3) = \{0, 1, 1, 1, 0, 0, 1, 1, 0\}$. Uses information are important for higher order hierarchical shape functions calculations in finite element analysis. They are also important to ensure mesh validity [2, 5].

4.6 Mesh Entity Identificator

Mesh vertices have an `iD` as we have seen in 4.2. We also define an `iD` for entities of higher dimension This `iD` must fullfill the following properties:

- Two mesh entities that share same vertices (i.e. that are equal) have the same `iD`. However, two different mesh entities of dimension > 0 may share the same `iD`, they may be differentiated in this case using a more complex equal operator that will check if all the vertices are equal

- Mesh entity `iD` is a symmetric function of vertices `iD`'s i.e. does not depend on the particular ordering of vertices
- Calculation of mesh entity `iD` must not lead to integer overflow: in 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. few mesh entities should share the same in order to avoid a lot of equal operator calculations`iD`

The mesh entity `iD` will be used as hash function in a hash table. In results sections, we will compare different `iD` definitions.

4.7 Minimum information

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? Due to hypothesis 4.1, vertices are to be present in all representations ($I_{00} = 1$). 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 $M_i^d \sqsubset G_j^d$ are to be present and classified if we want the ability to construct a representation with $I_{ij} = 1 \ , \ i,j = 0,1,2,3$. Note that all vertices are to be present in the representation but only ones that are classified on model vertices are to be classified. With the minimum of information we have defined, all the other ones may be classified univoquely: take all classified edges $M_i^1 \sqsubset G_j^1$and classify their unclassified vertices to G_j^1. Do the same for classified faces and regions and all vertices will be classified.

The algorithm for being able to build up all entities is straightforward. For a $3-D$ mesh, take all regions $M_i^3 \sqsubset G_j^3$, create faces $M_i^3\{M^2\}$ using region templates and classify all new faces to G_j^3. Then, take all faces $M_i^2 \sqsubset G_j^d$, create faces edges using faces templates and classify new edges to G_j^d. Finally, take all regions $M_i^3 \sqsubset G_j^3$, create region edges using region templates and classify new edges to G_j^3. We do not assert that this minimum data is neccessary, it is possible to recover classification informations using geometrical criteria but this operation is not that simple and, usually, there are multiple solutions to this problem. Our goal is to avoid this problem and only use topological information.

5 Design of the Algorithm Oriented Mesh Database

We describe principal classes used in AOMD:

- `class AdjacencyContainer` – a generic container for upward and downward adjacencies
- `class meshEntity` – a base class for all mesh entities
- `class meshEntityContainer` – a general container for meshes containing all kind of mesh entities

The oriented language used here is `C++`. Functions whose declarations are followed by `= 0` are pure virtual functions, and are required to be overridden by a derived class. For evident reasons of conciseness, only the principal features of the different classes we describe are presented.

5.1 Adjacency containers

The `class AdjacencyContainer` is the base class for all adjacency containers. The public interface of `class AdjacencyContainer` includes the following members:

```
class AdjacencyContainer {
public:
 virtual iterator begin()=0;
 virtual iterator end()=0;
 virtual meshEntity*
             find(meshEntity*)=0;
 virtual void add(meshEntity*)=0;
 virtual void del(meshEntity*)=0;
};
```

Iterators are STL iterators and the behavior of `begin()` and `end()` functions are like in STL containers. Members `add`, `del` and `find` are there to

add, delete and find mesh entities `meshEntity` in adjacency lists.

Downward entities containers are of fixed size, are ordered and need random access. These considerations lead our choice to use a STL vector `vector<meshEntity*>` which fullfill exactly our needs. Operations like deleting or searching should not be used because they are linear with respect of the size of the `vector`.

Upward entities requires a more complex data structure. We need to delete, add and search rapidly elements in the container. We have chosen the STL set `set<meshEntity*>` which provides bidirectional iterators, requires more storage because of extra pointers but provides delete, add and search operators in $\log_2 n$ if n is the data size. STL `set` requires a "less than" operator for its elements. We have shown in 4.2 that two mesh entities are always comparable. The less than operator will simply compare vertices `iD`'s of entities in lexicographic way. We define two derived classes `upwardAdjacencyContainer` and `class downwardAdjacencyContainer` that override all virtual functions of `class AdjacencyContainer` using specific STL algorithms.

5.2 Mesh Entity Class

The `class meshEntity` is a base class for all mesh entities. What is common to all mesh entities is:

```
class meshEntity {
protected :
 unsigned long iD;
 AdjacencyContainer *adjacencies[4];
public:
 meshEntity(DownwardAdjacencyContainer&,
            GeomEntity &);
 virtual ~meshEntity ();
 iterator begin(int dim);
 iterator end(int dim);
 meshEntity *find(meshEntity*);
 void add(meshEntity*);
 void del(meshEntity*);
 ...
```

All mesh entities have four `AdjacencyContainer` which may be of different nature depending on the dimension of the actual mesh entity.

Constructor `meshEntity::meshEntity` takes as input a list of downward entities and a geometrical entity where the mesh entity is classified. At this stage, it is possible to generate all downward adjacencies demanded for the particular algorithm. Another important operation done by the constructor is the calculation of the mesh entity `iD`. For vertices, we provide an iD generator that is able to give a unique iD to each new vertex.

Destructor `meshEntity:: meshEntity` insure that the object to be destructed is not present in any upward adjacency of any of its downward adjacencies.

Iterators `begin(int dim)` and `end(int dim)` are provided that iterates on adjacencies of dimension `dim`.

Pure virtual members :

```
 ...
 virtual int dimension () = 0;
 virtual bool use (meshEntity*) = 0;
 virtual meshEntity*
         template(int ith,int dim)=0;
 virtual int size (int dim) = 0;
};
```

are added to the `class meshEntity`. 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 an hexaedron will simply return 12 which is the number of edges in an hexaedron. Finally, function `template(int ith,int dim)` creates the `ith` mesh entity of dimension `dim`. It is using the template convention described in previous sections. Function `use (meshEntity* m)` computes the use of mesh entity `m` using templates.

5.3 Mesh Entity Container class

The `class meshEntityContainer` is a generic container of mesh entities of all dimensions:

```
class meshEntityContainer
{
  hashtable<meshEntity*> entities[4];
```

```
 public :
  iterator begin(int dim);
  iterator end(int dim);
  meshEntity *find(meshEntity*);
  void add(meshEntity*);
  void del(meshEntity*);
};
```

Since we must deal with large meshes, we need containers that provides efficient operators for searching, adding and deleting mesh entities and that have small memory requirements. For these reasons, we have chosen the `hash_set` container of the STL. Mesh entity `iD`'s will be considered as a good hash function if few different entities have identical `iD`'s. We must understand here that the hash function must represent the mesh entity, it has to be equal if two mesh entities are equal in the sense that we have defined. The hash function cannot be, for example, the address of the `meshEntity` object or an integer given by a `static` counter inside the mesh entity class. The `iD` must be based on vertices `iD`'s.

A good measure of the efficiency of an hash table is simply $\psi = \frac{N_e}{N_k}$ where N_e is the number of elements in the hash and where N_k is the number of different keys. In fact, ψ is proportional to the average number of operations needed for a search operation. The hash table also needs an equal operator which we already have defined: two entities are equal if they have the same vertices. If $\psi \sim 1$, only few equal operation will be needed.

5.4 Algorithms for Mesh Representation Modifications

Algorithms for modifying mesh representation apply to `class MeshEntityContainer`. There are only two algorithms of this kind. We may want to add upward adjacencies of dimension `adj` to all entities of dimension `dim` of `meshEntityContainer m`:

```
iterator i,j;
for(i=m.begin(dim);i!=m.end(dim);++i)
 for(j=(*i)->begin(adj);
     j!=(*i)->end(adj);
     ++j)(*j)->add(*i);
```

In this algorithm, we traverse all entities `*i` of dimension `dim`, we take all downward adjacencies `*j` of dimension `adj` and we add `*i` to the adjacency of `*j`. It is an inverse mapping operation. We may also want to add downward adjacencies of dimension `adj` to all entities of dimension `dim` of `meshEntityContainer m`:

```
iterator i; int j;
for(i=m.begin(dim);i!=m.end(dim);++i)
 for(j=0;j<(*i)->size(adj);++j){
  mEntity *q=(*i)->template(j,adj);
  if(mEntity *t = m.find(q)){
   (*i)->add(t); if(q!=t)delete q;
  }
  else{
   (*i)->add(q);m.add(q);
  }
 }
```

In this algorithm, we traverse all entities `*i` of dimension `dim`, we create, using templates, a set of downward adjacencies `*q` of dimension `adj` and we look into container `m` for an equal entity `*t`. If the entity exists, then we add `*t` into adjacency of `*i` and we delete `*q`. If not, we add `*q` into adjacency of `*i` and in `m`. Both algorithms requires 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.

6 Choice of the Mesh Entity `iD`

Let us consider a mesh entity M_i^d. We can always access its vertices so that we can always have the representation $M_i^d\{M^0\}$. There are a lot of different possible choices for calculating mesh entity `iD` $id(M_i^d)$:

- $id_1(M_i^d) = \sum_{j=1}^{n} \frac{id(M_j^0)}{n}$
- $id_2(M_i^d) = \max_j id(M_j^0)$
- $id_3(M_i^d) = \sum_{j=1}^{n} R(id(M_j^0))\%\mu$

where μ is a big integer number for example 10^8 and where R is a random number generator that takes the vertex `iD` as its seed ($R(i)$ is a *function* of i i.e. $i = j \rightarrow rand(i) = rand(j)$). Let us call $\psi(d_i, M^d)$ the efficiency of hash table as defined in 5.3 for a hash table containing entities of dimension d and using id_j for computing mesh entity `iD`'s. Table 2 shows results on tetrahedral meshes and table 3 shows results on hexaedral meshes. Comparing $\psi(d_3)$ and $\psi(d_2)$ or

Tet. meshes	T_1	T_2	T_3	T_4
Nb. Vertices	10891	30525	47546	61411
Nb. Edges	74357	213446	355119	568290
Nb. Faces	124398	360675	568290	280716
Nb. Regions	60931	177753	280716	364716
$\psi(id_1, M^1)$	3.56245	3.68714	3.74776	3.76395
$\psi(id_2, M^1)$	7.98593	8.23004	8.31643	8.33284
$\psi(id_3, M^1)$	1.00001	1.00004	1.00004	1.00007
$\psi(id_1, M^2)$	5.30731	5.59976	5.73278	5.81028
$\psi(id_2, M^2)$	15.5555	16.1817	16.3229	16.3923
$\psi(id_3, M^2)$	1.00001	1.00004	1.00007	1.00008

Table 2: Results for tetrahedral meshes

Hex. meshes	H_1	H_2	H_3
Nb. Vertices	15625	42875	166375
Nb. Edges	45000	124943	490050
Nb. Faces	43200	121380	481140
Nb. Regions	13824	39304	157464
$\psi(id_1, M^1)$	2.58621	2.69173	2.79613
$\psi(id_2, M^1)$	2.88148	2.91483	2.94560
$\psi(id_3, M^1)$	1.00000	1.00002	1.00004
$\psi(id_1, M^2)$	2.38279	2.59326	2.71983
$\psi(id_2, M^2)$	2.81598	2.85795	2.90314
$\psi(id_3, M^2)$	1.00000	1.00000	1.00003

Table 3: Results for hexaedral meshes

$\psi(d_1)$ shows that computing `iD`'s using d_3 is from far the best choice for all meshes. Only one operation is needed for finding an entity in the database. It takes 56 seconds to create all faces in mesh T_4 using id_2 while only 13 seconds are needed using id_3. Another more important reason while we should use id_3 is that we will never find special cases where $\psi(id_2, M^d) \gg 1$ so that hash tables will be no longer efficient.

7 Conclusions

We have developped here a model of mesh representation that is able to manage any adjacencies set. Some 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 very light: 2500 lines of codes for the whole mesh database. AOMD is also very efficient, as it was proved in section 6. Finally, AOMD is also relatively light in memory (70 MB for mesh T_4) for but some enhancements may still be done for making AOMD lighter.

Future work will focus mainly on a parallel version of AOMD. We will completely take into advantage the structure of AOMD in parallel.

References

[1] W. Burke, "Applied Differential Geometry," Cambridge, 1991.

[2] M.W. Beall and M.S. Shephard, "A general topology-based mesh data structure," *Int. J. Num. Meth. Engng.*, 40:727-758, 1997.

[3] M.S. Shephard, S. Dey and J.E. Flaherty, "A straightforward structure to construct shape functions for variable p-order meshes," *Comp. Meth. Appl. Mech. Engng.*, 147:209-223, 1997.

[4] M.S. Shephard, "The specification of physical attribute information for engineering analysis," *Engnineering with computers.*, 4:145-155, 1988.

[5] M.S. Shephard and M.K. George, "Reliability of automatic 3-D mesh generation," *Comp. Meth. Appl. Mech. Engng.*, 101:443-462, 1992.

[6] David R. Musser and Atul Saini, "STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library," Adison-Wesley, 1996.

[7] Silicon Graphics STL web site, "Standard Template Library Programmer's Guide," http://www.sgi.com/Technology/STL/.