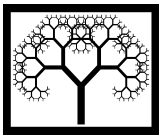
Paper 32



Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

An Independent Mesh Model Process to Write Meshing Algorithms

C. de Bellabre, F. Ledoux and J.-Ch. Weill
CEA, DAM, DIF
Arpajon, France

Abstract

Numerical simulation requires discretisation of geometrical domains into meshes for numerical schemes such as finite element methods. Depending on the meshing algorithm to be implemented, the mesh data structure may provide drastically different features: type of cells, available connections and so on. That is the reason why many mesh frameworks have been developed in the last decade in order to handle several mesh models, *i.e.* a combination of available cells and connections, with a common interface. Using such a framework, we provide in this paper a level of abstraction to write an algorithm for any mesh model. We experiment this feature onto an algorithm that generates a Delaunay triangulation.

Keywords: meshing framework, C++ generic programming, computation time, memory footprint, mesh model, Delaunay triangulation.

1 Introduction

One of the main building blocks of computational simulations is the mesh data structure. Depending on the numerical approximation methods or the meshing algorithm to implement, the mesh data structure must provide drastically different features. For instance, some algorithms require a full tetrahedral mesh where nodes and tetrahedra are known and thus kept in memory, while others require a full hexahedral mesh with the knowledge of hexahedra, faces, nodes and specific topological connections. In both cases, the optimal memory footprint should be achieved with specific implementation choices. In order to provide a single and uniform way to handle any kind of meshes, general meshing infrastructures have been developed in the last few years[2, 6, 10, 11]. In this work, we focus on one of them, the C++ *Generic Mesh Data Structure* (GMDS) framework[7], which allows the developer to select the kind of cells and connections

that are indeed kept in memory. *Generic* is used here for two meanings: first, the provided data structure can handle any *mesh model*, i.e. a combination of available cells and connections; second, it is based on a generic programming paradigm. At compile-time, it provides a tailored mesh data structure both in terms of computational complexity and memory footprint, which fits as best as possible the algorithm requirements.

A mesh framework like GMDS provides code sharing and a uniform access to any type of mesh model whatever the developers' requirements are. However, two main issues remain to be solved in an industrial context: *how to select the mesh model that best fits the requirements of an algorithm?*, and *how to combine algorithms requiring different mesh models without space memory duplication?* In order to provide a first answer, we show in this paper how we have added a new level of abstraction into GMDS: considering a mesh model M, we specialise at compile-time some pieces of C++ code to retrieve the connections that are not kept in memory according to M. By extension, an algorithm can be defined in a generic manner and launched with different mesh models in order to select the mesh model that best fits an expected balance between memory footprint and speed performances.

The remainder of this paper is organised as follows: in Section 2, some topological definitions are given to represent any mesh representation and combinatorial terms are defined to build topological connectivity; Section 3 gives some implementation details and, eventually, in Section 4, we give some results about a multi-model algorithm generating a 2d Delaunay triangulation.

2 Background Notions and Related Works

This work deals with the *cell-based* representation of meshes, which is used by most of numerical simulation that are based on finite element methods or finite volume methods. Generally , they use simplicial meshes or quad/hexahedral meshes with considering four types of cells (nodes, edges, faces and regions) and specific connections.

In this section, we introduce some background notions that are used in the paper. We begin by reviewing basic topological concepts related to meshes, then, we describe traditional combinatorial representations of meshes. Note that this work focus on topological aspects and, thus, we do not deal with geometrical representations, mesh classifications [10] or higher-order meshes.

2.1 Topological Definitions of Cells and Meshes

A *k-dimensional cell*, or *k-cell*, for brevity, is a subset of the d-dimensional Euclidean space E^d homeomorphic to a closed k-dimensional ball, where $k \leq d$ and $(d, k) \in \mathbb{N}^+ \times \mathbb{N}^+$. For instance, on Figure 1 (a), c_1^2, c_2^2 and c_3^2 are 2-cells. On Figure 1 (b), c_1^0 and c_1^1 are respectively a 0-cell and a 1-cell. For finite-element method, 3-dimensional

meshes are generally depicted as a set of 3-cells, called *regions*, 2-cells, called *faces*, 1-cells, called *edges* and 0-cells, called *nodes*.

Let $M = \{M^0, M^1, \ldots, M^d\}$ be a $(d+1)$-uple of set of cells such that for all $0 \leq i \leq d$, M^i contains all the i-cells of M and for all $c^i \in M^i$, the set ∂c^i is a collection of $(i-1)$-cells of M. Then M is a *d-dimensional mesh iff*

$$\begin{cases} \forall i \in [0..d],\ \forall (c^i, d^i) \in M^i \times M^i, \mathring{c^i} \cap \mathring{d^i} = \emptyset \\ \forall i \in [0..d-1],\ \forall c^i \in M^i,\ \exists c^d \in M^d \ / \ c^i \in \partial c^d \end{cases}$$

where $[i..j] = \mathbb{N} \cap [i, j]$ for all $0 \leq i < j$, and $\mathring{x}$ means the interior of x.

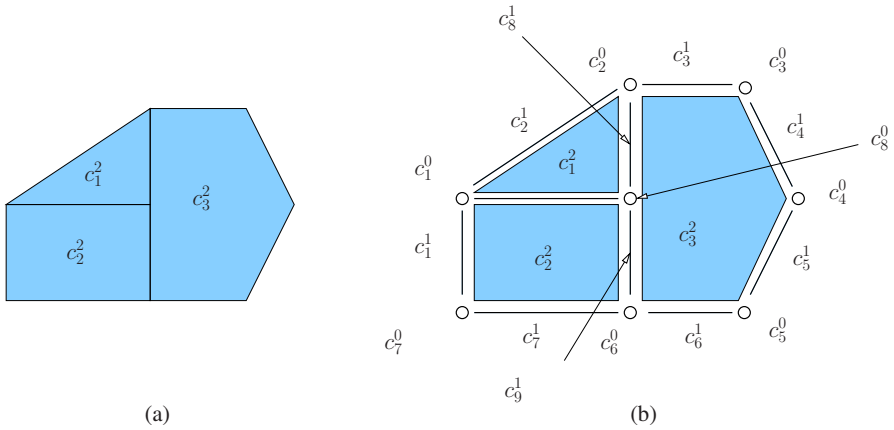


Figure 1: Example of a 2d mesh having 3 2-cells. In (a), the intersection of any pair of 2-cells is reduced to 1-cells, i.e. edges; In (b), some 1-cells and 0-cells are represented

Considering a d-mesh $M = \{M^0, M^1, \ldots, M^d\}$, some topological relations between cells are useful for writing algorithms and computing topological properties:

- An i-cell c^i and a j-cell c^j of M, with $0 \leq i < j \leq d$ are *incident iff* $c^i \in \partial c^j$.
- Two i-cells c_1^i and c_2^i are *j-adjacent*, with $0 \leq j < i \leq d$ *iff*

$$\exists c^j \in M^j \ / \ c^j \in c_1^i \cap c_2^i.$$

- Two i-cells, with $0 < i \leq d$ are said *adjacent* if they are $(i-1)$-adjacent.

On Figure 1 (b), the 1-cells c_2^1 and c_3^1 are adjacent since they are both incident to the 0-cell c_2^0. In order to define generic adjacency and incidence retrieval operators in Section 2.2, we also need the following notations:

- Let $(i, j) \in [0..d]^2$, let x be an i-cell, the set of j-cells adjacent or incident to x is denoted by $\text{Adj}_j(x)$. For instance, considering the 2-cell c_1^2 on Figure 2, we have $\text{Adj}_0(c_1^2) = \{c_1^0, c_2^0, c_6^0\}$. Reciprocally, considering the 0-cell c_2^0 we have $\text{Adj}_2(c_2^0) = \{c_1^2, c_2^2\}$.

- Let X be a multiset (or mset for short) of cells, i.e. an unordered collection of cells, which may contain *copies* or *multiples* of a cell. Then the multiset of k-cells adjacent or incident to at least one cell of X is denoted by

$$\text{Adj}_k(X) = \sum_{x \in X} \text{Adj}_k(x).$$

Considering that the number of copies of a cell c in an mset S is denoted by $m_S(c)$ and called its *multiplicity*, we have

$$\forall y \in S = \sum_{i \in I} A_i,\ m_S(y) = \sum_{i \in I} m_{A_i}(y).$$

For instance, considering the 2-cells c_0^2 and c_2^2 on Figure 2, we have

$$\text{Adj}_0(\{c_0^2, c_2^2\}) = \text{Adj}_0(c_0^2) + \text{Adj}_0(c_2^2) = \{c_1^0, c_1^0, c_2^0, c_3^0, c_3^0, c_4^0\}.$$

In this mset, the multiplicity of c_1^0 and c_3^0 is two and the multiplicity of c_2^0 and c_4^0 is one.

- Let $(i, j) \in [0..d]^2$ and X be an mset of cells, we introduce the notation

$$\text{Adj}_{i_1 i_2 \ldots i_k}(X) = \text{Adj}_{i_k} \circ \ldots \circ \text{Adj}_{i_2} \circ \text{Adj}_{i_1}(X)$$

where $(g \circ f)(x) = g(f(x))$ for $f : X \to Y$, $g : Y \to Z$ and $x \in X$. For instance, the mset $\text{Adj}_{02}(x)$, where x is a 2-cell, corresponds to all the 2-cells sharing a 0-cell with x, including n occurrences of x if x has n incident 0-cells. On Figure 2, the mset $\text{Adj}_{02}(c_{10}^0)$ is equal to

$$\begin{aligned}
\text{Adj}_{02}(c_{10}^0) &= \text{Adj}_0(\{c_5^2, c_8^2, c_9^2, c_7^2, c_6^2\}) \\
&= \text{Adj}_0(c_5^2) + \text{Adj}_0(c_8^2) + \text{Adj}_0(c_9^2) + \text{Adj}_0(c_7^2) +, \text{Adj}_0(c_6^2) \\
&= \{c_7^0, c_7^0, c_8^0, c_8^0, c_9^0, c_9^0, c_{10}^0, c_{10}^0, c_{10}^0, c_{10}^0, c_{10}^0, c_{11}^0, c_{11}^0, c_{12}^0, c_{12}^0, c_{13}^0, c_{13}^0\}.
\end{aligned}$$

- Considering an mset X, we introduce the set

$$\text{Keep}_n(X) = \{x | x \in X \text{ and } m_X(x) = n\}$$

which is the set of elements that have at least n copies in X. For instance, considering the previously given mset $\text{Adj}_{02}(c_{10}^0)$, we have

$$\begin{aligned}
\text{Keep}_1(\text{Adj}_{02}(c_{10}^0)) &= \{c_7^0, c_8^0, c_9^0, c_{10}^0, c_{11}^0, c_{12}^0, c_{13}^0\}, \\
\text{Keep}_2(\text{Adj}_{02}(c_{10}^0)) &= \{c_7^0, c_8^0, c_9^0, c_{10}^0, c_{11}^0, c_{12}^0, c_{13}^0\}, \\
\text{Keep}_3(\text{Adj}_{02}(c_{10}^0)) &= \{c_{10}^0\}.
\end{aligned}$$

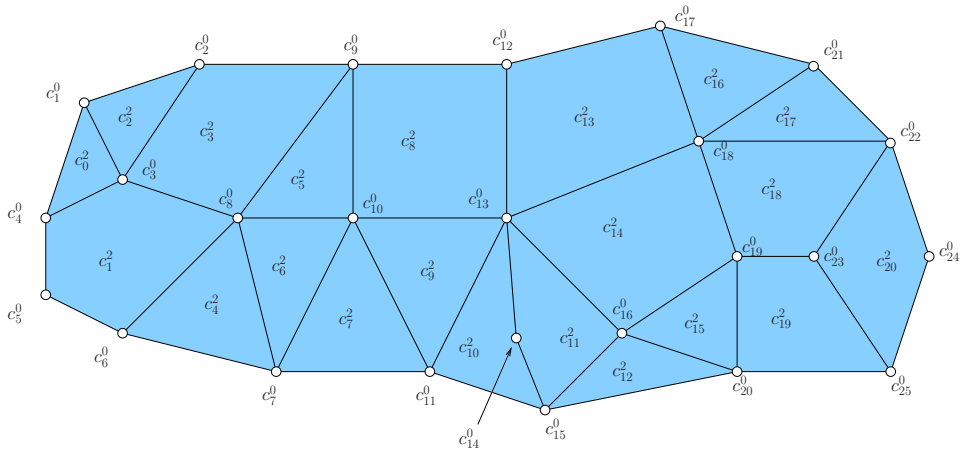


Figure 2: A simple 2d mesh containing several types of 2-cells (triangles, quadrilaterals, pentagons)

2.2 Mesh Model

Two cell-based representations differ in the type of cells they provide or the incidence and adjacency relations they explicitly store. The choice of what is explicitly described in a mesh representation is a consequence of the algorithm to implement. Then, for a particular algorithm, it might be useless to have all the types of cells. For instance, geometric smoothing or Delaunay-like algorithms only require nodes and faces in 2d. Each mesh data structure corresponds to a particular *mesh model* [6, 8] that we formalise as following: An n**-dimensional mesh model** M, with $n \geq 0$, is a couple (C, I) where

1. C, a finite subset of $\mathbb{N}$, indicates which dimensions of cells are managed by M;
2. I, a finite subset of $C \rightarrow C$, is a set of relations corresponding to the connections managed in M. Relation noted $(i \rightarrow j)$ indicates that the connection from i-cells to j-cells is stored in M.

Figure 3 and Figure 4 show some mesh models. To illustrate this definition, let us consider mesh models $R1$ and $F1$. Mesh model $R1$ is defined by $(\{0, 3\}, \{0 \rightarrow 3, 3 \rightarrow 0\})$ and mesh model $F1$ is defined by $(\{0, 1, 2, 3\}, \{0 \rightarrow 1, 1 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 1, 2 \rightarrow 3, 3 \rightarrow 2\})$. Considering this mesh model, a cell-base representation can be classified as being *full* or *reduced*, *complete* or *incomplete* [8, 9]. A mesh representation is *full iff* all the cell types are explicitly stored (see Figure 3). Otherwise it is said *reduced* (see Figure 4). A mesh representation is said to be *complete iff* any adjacency and incidence relation can be retrieved for any cell without a global traversal of the mesh. In the other cases, the representation is said *incomplete*.

Another way to represent a mesh model is to use direct and indirect connectivity matrices. Here, we present their definitions only for 2d and 3d mesh models. Let M be a mesh model, the *direct matrix* D^M is the 4×4 matrix such that $D^M_{i,j} = 1$ *iff* the

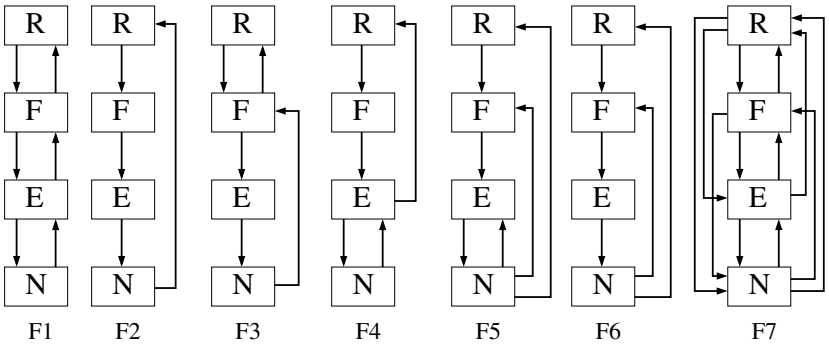


Figure 3: Some full models

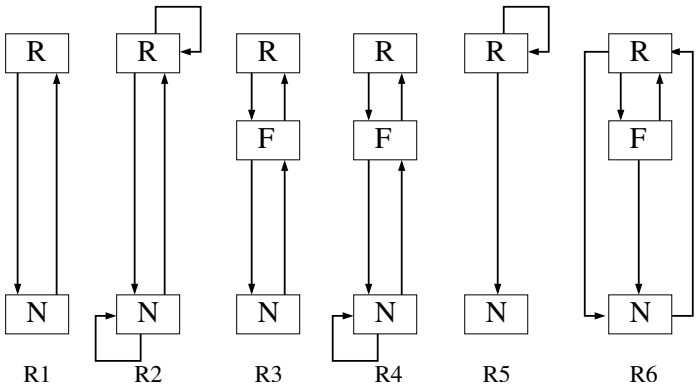


Figure 4: Some reduced models

connection $i \to j$ is in M, 0 otherwise. The *indirect matrix* I^M is the 4×4 matrix such that

$$\begin{array}{rcll} I^M_{i,j} & = & \times & \text{if } D^M_{i,j} = 1, \\ & = & \text{k} & \text{with } 0 \leq k \leq 3 \text{ if the shortest connection} \\ & & & \text{path from } i \text{ to } j \text{ is } i \to k \to \ldots \to j, \\ & = & - & \text{if there is no way to get } j \text{ from } i. \end{array}$$

For instance, let us consider the full model $F1$ (see Figure 3) and the reduced model $R2$ (see Figure 3). Their direct and indirect matrices are:

$$D^{F1} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad I^{F1} = \begin{pmatrix} 1 & \times & 1 & 1 \\ \times & 0 & \times & 2 \\ 1 & \times & 1 & \times \\ 2 & 2 & \times & 2 \end{pmatrix}$$

$$D^{R2} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \quad I^{R2} = \begin{pmatrix} \times & - & - & \times \\ - & - & - & - \\ - & - & - & - \\ \times & - & - & \times \end{pmatrix}$$

We can note that the indirect matrix of $F1$ shows we can always access from i to j for any values $(i, j) \in [0..3]^2$ while it is not the case for $R2$. For instance, in order to get nodes of a region r, i.e. the connection $3 \rightarrow 0$, $I^{F1}_{3,0} = 2$ indicates that we have to get the set of faces incident to r, i.e the set $\text{Adj}_2(r)$. Then for each face, to get nodes, we have to go through edges since $I^{F1}_{3,0} = 1$. The connection from edges to nodes being direct ($I^{F1}_{1,0} = \times$), the nodes of r are in the multiset $\text{Adj}_{210}(r)$.

2.3 Topological Queries for any Mesh Model

Considering a mesh model M, it is straightforward to get the j-cells incident to an i-cell C^i_k if $D^M_{i,j} = 1$. This operation is local and optimal in computational time. But some other options are possible to get this incidence relation under relaxed properties of M. For instance, getting the nodes incident to a face c^2_k in the mesh model $\{0, 1, 2, 2 \rightarrow 1, 1 \rightarrow 0\}$ can be obtained by computing $\text{Adj}_{10}(c^2_k)$ and keeping some elements of this multiset. The aim of this section is to show how to retrieve any adjacency or incidence relation for any mesh model M even it is incomplete. To do that, we introduce the notion of *extended mesh model* and we restrict our approach to a specific range of meshes (which are suitable for numerical simulation). Considering a mesh model $M = (C, I)$, the extended model $M_e = (C_e, I_e)$ of M is such that:

- $C_e = C$
- $I_e = \{(i \rightarrow j) | (i \rightarrow j) \in I \vee (j \rightarrow i) \in I\}$

In other words, if you consider M as being an oriented graph M_e is the corresponding non-oriented graph. In order to support the indirect topological queries we restrict our approach to any d-dimensional mesh M that verifies:

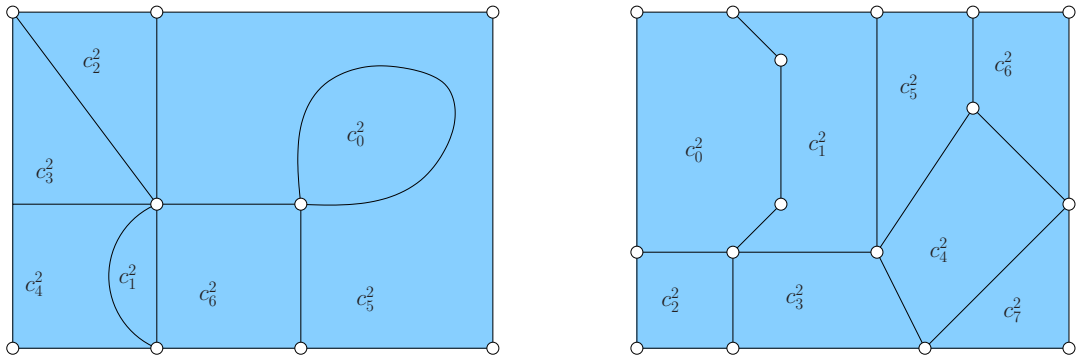
1. for all $i \in [1..d]$ and $x \in M_i$, either x is an i-simplex or can be split into i-simplices;
2. for all $i \in [1..d]$ and $(x, y) \in M_i \times M_i$, x and y share at most two $(d-1)$-cells.

The first property eliminates topological degenerated cells, like flat elements (see Figure 5 (a)), while the latter property is necessary to get some incidence and adjacency relations (see Figure 5 (b)).

2.3.1 Incidence Set

Let $M = (C, I)$ be a mesh model, let $(i, j) \in C^2$ and x an i-cell, we can retrieve the set of j-cells adjacent to x if there exists a path from i to j in M_e that does not go through a value k greater than i and j. The existence of this path allow us to create a mset containing the expected set. Considering an i-cell x, three options are then possible to get an mset $\text{Adj}_{k_1 k_2 \ldots k_p j}(x)$ containing $\text{Adj}_j(x)$:

1. $i \rightarrow k_1 \rightarrow \ldots \rightarrow k_p \rightarrow j$ with $i < k_1 < \ldots < k_p < j$ is a path in M_e;



(a) (b)

Figure 5: Example of 2d meshes we do not consider. In (a), cells c_0^2 and c_1^2 are not a 2-simplex since they have less than three incident 0-cells or 1-cells; In (b), cells c_0^2 and c_1^2 share more than two edges

2. $i \to k_1 \to \ldots \to k_p \to j$ with $i > k_1 > \ldots > k_p > j$ is a path in M_e;
3. $i \to k_1 \to \ldots \to k \to \ldots \to k_p \to j$ with $i > k_1 > \ldots > k < \ldots < k_p < j$ is a path in M_e.

In order to illustrate these three options, let us consider Figure 2 and some mesh models. Considering the 0-cell c_1^0 and the mesh model $(\{0,1,2\}, \{0 \to 1, 1 \to 2, 2 \to 0\})$, the set of the 2-cells incident to c_1^0 is included in $\text{Adj}_{12}(c_1^0) = \{c_0^2, c_0^2, c_2^2, c_2^2\}$ (option 1). Considering the 2-cell c_0^2 and the same mesh model, the set of the 0-cells incident to c_0^2 is included in $\text{Adj}_{10}(c_0^2) = \{c_1^0, c_1^0, c_3^0, c_3^0, c_4^0, c_4^0\}$ (option 2). Eventually, considering the 1-cell $[c_1^0, c_3^0]$ and the same mesh model$(\{0,1,2\}, \{0 \to 1, 2 \to 0\})$, the set of the 2-cells incident to $[c_1^0, c_3^0]$ is included in $\text{Adj}_{02}([c_1^0, c_3^0]) = \{c_0^2, c_0^2, c_1^2, c_2^2, c_2^2, c_3^2\}$ (option 3). If the existence of one of the three previous path in M_e ensures the ability to build $i \to j$ for any mesh model, it does not ensure an optimal computational time solution. To achieve this optimal, we follow the next priority order:

1. If $D_{i,j}^M = 1$ then getting the j-cells incident to an i-cell c^1 is local to c^i, else
2. If $I_{i,j}^M \neq -$ then getting the j-cells incident to an i-cell c^1 is local to a neighbourhood of c^i, and thus independent of the mesh size, else
3. If $D_{i,j}^{M_e} = 1$ then getting the j-cells incident to an i-cell c^1 corresponds to a global traversal of M^j to find the j-cells being incident to c^i, else
4. If $I_{i,j}^{M_e} \neq -$ then it exists a series of local and global traversals to getting the j-cells incident to an i-cell c^1.

The two first options require a local traversal of the mesh structure, while the two last options require at least one global traversal.

In practice, in order to get the set $\text{Adj}_j(x)$ of j-cells incident to an i-cell x in a mesh model M, only some copies of a corresponding incidence mset have to be kept. The choice of copies to keep depends on their multiplicity, the type of connection $(i \to j)$ and the existing paths in M_e. Since 3 is the maximum diameter of M_e (considered as being a graph), a shortest path between any i and j in M_e has a size of 1, 2 or 3. Let us suppose that such a path exists. If its length is one, we have $D^M_{i,j} = 1$ or $D^{M_e}_{i,j} = 1$. In both cases, $\text{Adj}_j(x)$ is the expected set. If its length is two, it means there exists k such that $(i \to k) \in M_e$ and $(k \to j) \in M_e$. We have then the three possible cases given previously with only the value k in the path. If $i < k < j$ or $i > k > j$ then we have to keep a copy of each element being in $\text{Adj}_{kj}(x)$ (see Table 1). In fact, when the dimension values of the path define a monotonic function, we build a growing geometrical neighbourhood, and thus, we never reach j-cells that are not incident to x. Thus we can keep one copy of each element being in the incidence mset. Let us suppose that $i > k$ and $j > k$, then the mset $\text{Adj}_{kj}(x)$ contains j-cells that are not incident to x. In order to build the corresponding incidence set, we have to keep j-cells having at least $p + 1$ copies in $\text{Adj}_{kj}(x)$ with $p = \min(i, j)$ (see Table 1). Let us prove it. In the following, x denotes an i-cell.

- First, let us suppose that $i > j$. As we only consider 2d and 3d meshes, $i = 2$ or $i = 3$.
  - Let $i = 2$, then $(j, k, p) = (1, 0, 1)$. The mset $\text{Adj}_{01}(x)$ contains all the 1-cells incident to at least one 0-cell incident to x. Since an edge is topologically defined with two 0-cells, only the 1-cells having $2 = p+1$ copies in $\text{Adj}_{01}(x)$ must be kept.
  - Let $i = 3$, then $(j, k, p) = (1, 0, 1)$ or $(2, 1, 2)$ or $(2, 0, 2)$. If $(j, k, p) = (1, 0, 1)$, the mset $\text{Adj}_{01}(x)$ contains all the 1-cells incident to at least one 0-cell that is incident to x. Since an edge is defined with two 0-cells, only the 1-cells having $2 = p + 1$ copies in $\text{Adj}_{01}(x)$ must be kept. If $(j, k, p) = (2, 0, 2)$, the mset $\text{Adj}_{02}(x)$ contains all the 2-cells incident to at least one 0-cell incident to x. Since a face is defined with at least three 0-cells, only the 2-cells having at least $3 = p + 1$ copies in $\text{Adj}_{02}(x)$ must be kept. If $(j, k, p) = (2, 1, 2)$, we use the fact that a face has at least three edges to get the same conclusion.
- Let us now suppose that $i < j$. We get the same four cases than beforehand and the same results indeed. Let us just consider the case $(i, j, k, p) = (2, 3, 0, 2)$. The mset $\text{Adj}_{03}(x)$ contains all the 3-cells incident to at least one 0-cell incident to x. Since a face is defined with at least three 0-cells, a region, or 3-cell, incident to x has at least $3 = p + 1$ copies in $\text{Adj}_{03}(x)$. Equivalent proofs can be lead for the three other cases.

Table 1 summarises all the possibilities to retrieve incidence relations between cells in 2d and 3d meshes. Eventually, we consider paths of length 3. Each of this path goes through all the type of cells in 3d and does not goes through a dimension greater than i

and j. They are summarised in Table 2. The number of j-cells to keep is defined in the same way as for 2-length paths. In the case of paths that do not follow a monotonic way, we have to reduce msets to sets by keeping one copy per k-cell. The scheme we use to compute an incidence set is equivalent to the 2-length path having the same minimal intermediate node. For instance, the path $3 \to 0 \to 1 \to 2$ is equivalent to the path $3 \to 0 \to 2$.

	$\searrow$		$\swarrow\!\nearrow$	
$i > j$	$2 \to 1 \to 0$	$(\text{Keep}_1 \circ \text{Adj}_{10})(x)$	$2 \to 0 \to 1$	$(\text{Keep}_2 \circ \text{Adj}_{01})(x)$
	$3 \to 2 \to 0$	$(\text{Keep}_1 \circ \text{Adj}_{20})(x)$	$3 \to 0 \to 1$	$(\text{Keep}_2 \circ \text{Adj}_{01})(x)$
	$3 \to 2 \to 1$	$(\text{Keep}_1 \circ \text{Adj}_{21})(x)$	$3 \to 0 \to 2$	$(\text{Keep}_3 \circ \text{Adj}_{02})(x)$
	$3 \to 1 \to 0$	$(\text{Keep}_1 \circ \text{Adj}_{10})(x)$	$3 \to 1 \to 2$	$(\text{Keep}_3 \circ \text{Adj}_{12})(x)$
$i < j$	$0 \to 1 \to 2$	$(\text{Keep}_1 \circ \text{Adj}_{12})(x)$	$1 \to 0 \to 2$	$(\text{Keep}_2 \circ \text{Adj}_{02})(x)$
	$0 \to 1 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_{13})(x)$	$2 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_{03})(x)$
	$0 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_{23})(x)$	$2 \to 1 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_{13})(x)$
	$1 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_{23})(x)$	$1 \to 0 \to 3$	$(\text{Keep}_2 \circ \text{Adj}_{03})(x)$

Table 1: Indirect incidences for 2-length paths. The first column gathers monotonic paths and the second column gathers acceptable non-monotonic paths.

	$\searrow$	
$i > j$	$3 \to 2 \to 1 \to 0$	$(\text{Keep}_1 \circ \text{Adj}_{210})(x)$
$i < j$	$0 \to 1 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_{123})(x)$
	$\swarrow\!\nearrow$	
$i > j$	$3 \to 2 \to 0 \to 1$	$(\text{Keep}_2 \circ \text{Adj}_1 \circ \text{Keep}_1 \circ \text{Adj}_{20})(x)$
	$3 \to 1 \to 0 \to 2$	$(\text{Keep}_3 \circ \text{Adj}_2 \circ \text{Keep}_1 \circ \text{Adj}_{10})(x)$
	$3 \to 0 \to 1 \to 2$	$(\text{Keep}_3 \circ \text{Adj}_2 \circ \text{Keep}_1 \circ \text{Adj}_{01})(x)$
$i < j$	$2 \to 1 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{10})(x)$
	$2 \to 0 \to 1 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{01})(x)$
	$1 \to 0 \to 2 \to 3$	$(\text{Keep}_2 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{02})(x)$

Table 2: Indirect incidences for 3-length paths. Non-monotonic paths require to use the *Keep* operator twice

2.3.2 Adjacency Set

Considering a mesh model $M = (C, I)$, we want now to retrieve the d-cells adjacent to a d-cell C_k^d with $d = 2$ (2d) or $d = 3$ (3d). If $D_{d,d}^M = 1$, it is straightforward, local and optimal in computational time. Otherwise, getting an adjacency relation consists in getting a value $k \in C$ such that it exists a path from d to k and $k \to d \in I$. The possible cases are resumed in Table 3. Computing the path from d to k is done by applying the computational terms given for building incidence sets previously. We get

then a set S_k containing all the k-cells incident to c^d. Then we use the Adj_d operator to get an mset S' containing all the d-cells adjacent to c^d. By construction, the d-cell c^d has $|S_k|$ copies in S' and must be removed to get the expected adjacency set. Moreover, the d-cells to keep are the ones having more than $d-k$ copies in S' but c^d.

path	computational term
$2 \to 1 \to 2$	$(\text{Keep}_1 \circ \text{Adj}_{12})(x) - \{x\}$
$2 \to 0 \to 2$	$(\text{Keep}_2 \circ \text{Adj}_{02})(x) - \{x\}$
$3 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_{23})(x) - \{x\}$
$3 \to 1 \to 3$	$(\text{Keep}_2 \circ \text{Adj}_{13})(x) - \{x\}$
$3 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_{03})(x) - \{x\}$
$2 \to 1 \to 0 \to 2$	$(\text{Keep}_2 \circ \text{Adj}_2 \circ \text{Keep}_1 \circ \text{Adj}_{10})(x) - \{x\}$
$2 \to 0 \to 1 \to 2$	$(\text{Keep}_1 \circ \text{Adj}_2 \circ \text{Keep}_2 \circ \text{Adj}_{01})(x) - \{x\}$
$3 \to 2 \to 1 \to 3$	$(\text{Keep}_2 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{21})(x) - \{x\}$
$3 \to 1 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_3 \circ \text{Keep}_3 \circ \text{Adj}_{12})(x) - \{x\}$
$3 \to 2 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{20})(x) - \{x\}$
$3 \to 0 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_3 \circ \text{Keep}_3 \circ \text{Adj}_{02})(x) - \{x\}$
$3 \to 1 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{10})(x) - \{x\}$
$3 \to 2 \to 1 \to 0 \to 3$	$(\text{Keep}_3 \circ \text{Adj}_3 \circ \text{Keep}_1 \circ \text{Adj}_{210})(x) - \{x\}$
$3 \to 1 \to 0 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_3 \circ \text{Keep}_3 \circ \text{Adj}_2 \circ \text{Keep}_1 \circ \text{Adj}_{10})(x) - \{x\}$
$3 \to 0 \to 1 \to 2 \to 3$	$(\text{Keep}_1 \circ \text{Adj}_3 \circ \text{Keep}_3 \circ \text{Adj}_2 \circ \text{Keep}_1 \circ \text{Adj}_{01})(x) - \{x\}$

Table 3: Indirect Adjacency paths

3 A C++ Multi Model Mesh Framework

In order to provide a framework allowing the developer to select the mesh model he wants for writing algorithms, we have developed a C++ library called GMDS for *Generic Mesh Data Structure*[7].This library uses the main features of the object-oriented programming (OOP) and generic programming paradigm:

- OOP provides the necessary level of abstraction to ease the development of complex software. Some key concepts of OOP are data encapsulation, inheritance and pure virtual classes, or *interfaces*: In OOP, software is organised around classes that encapsulate data structures and the treatments needed to access and manipulate this data; Inheritance allows the developer to create new classes specialising existing ones; Interfaces allow the developer to define sets of services (depicted by operations) that concrete classes will implement. The latter notion is essential to define modular and evolving software architectures.

- In response to some of the shortcomings of OOP (like computing time penalties), generic programming has gain widespread acceptance as a complementary

programming paradigm, especially in the C++ community. Generic programming is mainly used to define generic containers or generic algorithms which are independent of the objects they deal with. In numerical simulation codes, generic programming is also used to improve speed performances by performing mathematical computation at compile-time. Despite the appeal of generic programming, its exclusive use for large-scale software is not viable, since the excessive use of templates usually leads to cryptic software that is hard to understand, debug and has very long compilation times.

Mixing oriented-object programming and generic programming has also been done in many software packages like CGAL [4] and GrAL [1] for computational geometry and meshing, OpenMesh [2] for polygonal meshing and some other works for finite element software [5]. In GMDS, generic programming is used to get for any mesh model M a tailored memory footprint and specific behaviours.

In the following of this section, we shortly present some key points of the GMDS implementation and we focus on how we handle adjacency and incidence relations whatever the mesh model is. The interested reader can refer to [7] for more details about the memory footprint of the data structure.

3.1 Design Overview

Figure 6 shows a simplified class diagram, which gives a structural view of the cell-based mesh kernel. Classes can be split into four categories :

1. **Mesh interfaces** are `Node`, `Edge`, `Face`, `Region`, `Cell` and `Mesh` classes. They give a user-friendly access to mesh concepts without dealing with template parameters (except for the `Mesh` class).

2. **Internal mesh classes** are `TNode`, `TEdge`, `TFace`, `TRegion` and `TCell` classes. They use generic programming to optimise memory footprint.

3. **Allocation classes** are not represented in Figure 6 but are essential. They manage memory allocation to improve memory allocation performances. Traditional techniques like chunk allocations are performed.

4. **Policy classes** encapsulate the behaviour that depends on template parameters (mesh model and cell types).

In Figure 6, generic parameter `TMask` defines the mesh model we want to use. We call it the *model mask* and it defines the mesh model. Every internal mesh class has this generic parameter. It is defined as a combination of the following symbols: `Dim2` and `Dim3` define the mesh dimension; `N` `E`, `F`, `R` indicates which types of cells are available; `X2Y` with (`X`, `Y`)$\in$\{`N`, `E`, `F`, `R`\}^{2} indicates that the topological relation from cells of type `X` to cells of types `Y` are stored. For instance, to define and work with a mesh in the mesh model R_1 (see Figure 4), the user has to use an instance of the

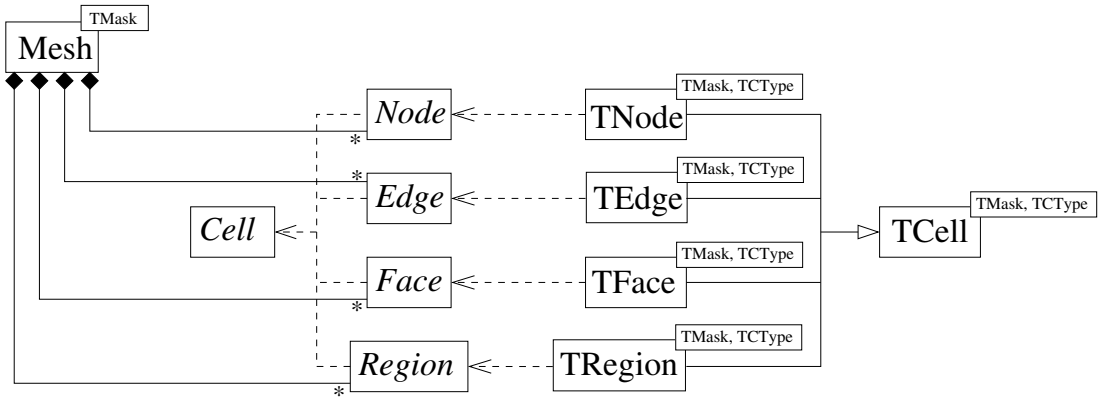


Figure 6: Simplified class diagram of the GMDS cellular mesh module.

class `Mesh<Dim3|N|R|R2N|N2R>`. It means this is a 3d mesh (`Dim3`) composed by nodes (`N`) and regions (`R`) where connections $3 \rightarrow 0$ (`R2N`) and $0 \rightarrow 3$ (`N2R`) are stored.

All internal classes, but `Mesh` class, have also an extra parameter defining the type of cell they represent. For instance, for a face, values can be `GMDS_QUAD`, `GMDS_TRIANGLE` or `GMDS_POLYGON`. These parameters are used to specialise memory consumption and cell behaviour. Each internal cell class implements an interface and inherits from the `TCell` class. From a user point of view, internal classes, allocation classes and policy classes are hidden. Developers only use cell interfaces and the `Mesh` class. This design provides flexibility and a user-friendly interface.

3.2 Implementation of Basic Queries and Modifications

We use template specialisation to optimise both accesses and basic modifications for a specific mesh model. For instance, if an operation is not available, the decision to throw a C++ exception for this operation is performed at compile-time. As a consequence, no extra conditional test is performed during the program execution. This way, even if we certainly do not reach the speed performances of dedicated mesh data structure that can be optimised, there is no penalty due to the mesh model.

Since the mesh model is known at compile-time, we use traits and policy techniques to generate the adequate source code corresponding to the computational terms given in Section 2. For instance, let us consider the mesh model $M = (\{0,1,2\}, \{2 \rightarrow 1, 0 \rightarrow 1\})$. We want to get the nodes incident to a face x. Considering M_e, we have a path $2 \rightarrow 1 \rightarrow 0$, and thus the nodes incident to x are obtained by $(\text{Keep}_1 \circ \text{Adj}_{10})(x)$. Considering the C++ variable `x`, which is a pointer to a `Face` object, we write with GMDS,

```
std::vector<Node*> nodes = x->getNodes();
```

In the body of the `getNodes()` operation, a call to the operation `vec()` of the policy class `GetAdjPolicy` is done. This template class has three template param-

eter corresponding to the mesh model, the dimension of the departure cell and the dimension of the destination cells. For the incidence relation $(2 \rightarrow 0)$, we use the specialisation `GetAdjPolicy<M,2,0>` (see Listing 1). In the operation `vec()`, we check some properties of the model in order to get the nodes in an optimal computational time according to the mesh model. For that we follow the priority order given in Section 2. Thus, if the mesh model stores the connection $(2 \rightarrow 0)$ (`DirectAdj<M,2,0>::yes`), we use an template operation given directly the nodes of our face. The second test is for $(2 \rightarrow 1) \wedge (1 \rightarrow 0)$ that uses two local traversal. With the mesh model M, the condition of Line 9 in Listing 1 is true, and thus three operations are called (line 10 and 11):

- First, `Get<1>::Adj(f)` provides all the edges incident to the 2-cell `f`. Let `E` to be this set of edges.
- Second, `GetInv<M,0>::Adj(m,E)` computes the mset of nodes incident to at least one 1-cell of `E`. It is a global operation that requires to go through all the nodes of the mesh. Let `N` to be this mset of nodes.
- Third, `Keep<Node*,1>::AtLeast(N)` is an operation that returns the set of nodes having at least one copy in `N`.

The template class `GetAdjPolicy<M,i,j>` is specialised in a similar way for any couple $(\texttt{i}, \texttt{j}) \in [0..3]^2$ and it implements the computational terms given in Tables 1, 2 and 3. Note that the conditional decisions that are in the body of the `vec()` operation are solved at compile-time since they only depend of template parameters. Thus, in the case of M, the `vec()` operation is reduced to a call to line 10/11.

4 A Multi Model 2d Delaunay Algorithm

In order to illustrate the benefits of having multi-model topological query operators, we have studied the Bowyer-Watson algorithm [3, 12], which generates a Delaunay triangulation $\mathcal{T}$ of an arbitrary set of points $\mathcal{P}$ by sequentially adding new points and modifying the existing triangulation by means of purely local operations (if the mesh model allows it). This algorithm proceeds in an iterative manner, taking a point of $\mathcal{P}$ and inserting it into $\mathcal{T}$. Traditionally, the algorithm starts with an initial triangulation $\mathcal{T}_0$ of the bounding box of $\mathcal{P}$ and terminates when $\mathcal{P}$ is empty.

4.1 Algorithm implementation

Let us now briefly detail each step i of this algorithm. Let p_i be the point of $\mathcal{P}$ to insert during this step and $\mathcal{T}_{i-1}$ be the triangulation before beginning this step:

1. *Getting a triangle of $\mathcal{T}_{i-1}$ containing p_i* (see Figure 7 (a) and (b)) - We start with a triangle $T \in \mathcal{T}_{i-1}$ randomly chosen and we compute the barycentric

```
template<int M> struct GetAdjPolicy<M,2,0>{
  static std::vector<Node*> vec(Mesh<M>& m, Face* f)
  {
    if(DirectAdj<M,2,0>::yes)
        return Get<0>::Adj(f);
    else if(DirectAdj<M,2,1>::yes && DirectAdj<M,1,0>::yes)
        return Keep<Node*,1>::AtLeast(Get2<1,0>::Adj(f));
    else if(DirectAdj<M,0,2>::yes)
        return Keep<Node*,1>::AtLeast(GetInv<M,0>::Adj(m,f));
    else if(DirectAdj<M,2,1>::yes && DirectAdj<M,0,1>::yes)
        return Keep<Node*,1>::
                  AtLeast(GetInv<M,0>::Adj(m,Get<1>::Adj(f)));
    else if(DirectAdj<M,1,2>::yes && DirectAdj<M,1,0>::yes)
        return Keep<Node*,1>::
                  AtLeast(Get<0>::Adj(GetInv<M,1>::Adj(m,f)));
    else if(DirectAdj<M,1,2>::yes && DirectAdj<M,0,1>::yes)
        return Keep<Node*,1>::AtLeast(GetInv2<M,1,0>::Adj(m,f));
  }
};
```

Listing 1: Policy class specialisation providing the $2 \rightarrow 0$ incidence relation for any mesh model

coordinates of p_i in T. These coordinates indicate if p_i is in T or in which direction to go to find a triangle containing p_i. In the latter case, we go to the incident face of T along this direction. Thus we need the connection $(2 \rightarrow 2)$ in order to topologically traverse the mesh and the connection $(2 \rightarrow 0)$ to compute barycentric coordinates.

2. *Cavity creation* (see Figure 7 (c)) - Let T_i be the triangle containing p_i, some incident triangles of T_i could have their circumcircle that contains p_i. All those triangles define the cavity $\mathcal{C}_i$. Connections $(2 \rightarrow 2)$ and $(2 \rightarrow 0)$ are necessary here too.

3. *Cavity triangulation* (see Figure 7 (d)) - All the triangles of $\mathcal{C}_i$ must be deleted and new triangles are created by properly connecting p_i and the nodes of $\partial\mathcal{C}_i$.

In 2d, a natural mesh model to implement this algorithm is thus $(\{0,2\}, \{2 \rightarrow 0, 2 \rightarrow 2\})$. Indeed, faces are defined by their incident nodes and local traversal are naturally performed by finding the faces adjacent to a given one. We have thus implemented a simple Bowyer-Watson algorithm on this mesh model before extending it to any available 2d mesh model. From a programming point of view, the extension to any mesh model M is straightforward if:

- the algorithm is encapsulated in a template class having the mesh model as a template parameter,

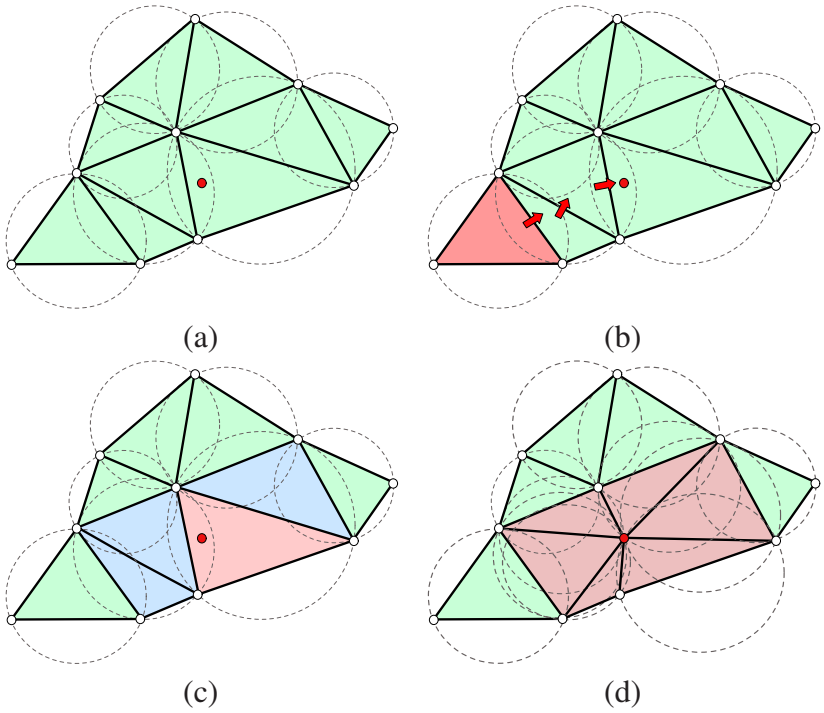


Figure 7: The main steps of the Bowyer-Watson algorithm. The red point is the point to insert in the Delaunay triangulation

- the topological modifications are handled for any mesh model,
- there is no optimisation specific to M.

In order to satisfy the second item, we had to provide generic operations to create and delete a triangle. These operations are the two mesh modifications that are performed during the Bowyer-Watson algorithm. Each time, a triangle is created or deleted the adjacency and incidence relations are updated and some cells can also be created or deleted. It is the case of edges if they are defined in the mesh model M. They are useless in our implementation of the Bowyer-Watson algorithm but they have to be handled in order to keep a well-defined mesh. As a consequence, when we create a face, we can have to create its incident edges or find them if they already exist in the mesh. This research operation requires to get the nodes incident to an edge: if M contains edge, it has to contain $(1 \rightarrow 0)$ or $(0 \rightarrow 1)$. The third optimisation can be done, but a generic algorithm must be provide. Using the mesh model $(\{0, 2\}, \{2 \rightarrow 0, 2 \rightarrow 2\})$, a traditional optimisation to improve the mesh traversal is to store incident nodes and adjacent faces in a specific order locally to a face: In a face f, the node with local number i is not incident to the face adjacent to f and having local number i. As we want a generic algorithm, we can not use such a local numbering in our implementation. This led us to write an operation that returns the that shares two nodes with another face. To be optimal in computational time, this operation requires that the mesh model provides relation $(0 \rightarrow 2)$.

Another drawback of our implementation is that we have to check the orientation of triangles before any computation where the nodes' order is important (circumcircle computation). Indeed, we can not ensure that getting the nodes of a triangle in two different mesh models gives the same ordered collection of nodes. If the connection $(2 \rightarrow 0)$ is stored then you can store and retrieve nodes in a defined order. On the contrary, getting nodes of a face using connections $(2 \rightarrow 1)$ and $(0 \rightarrow 1)$ does not preserve any order. It is important to note that defining a multi-model algorithm will never provide the best algorithm in terms of computational time. It is always better to write a dedicated mesh data structure where connections are stored in a tricky way and to improve the algorithm.

4.2 Experimental Results

We have implemented the Bowyer-Watson algorithm in 2d using the GMDS framework. In 2d, there are 7 possible connections and thus 128 theoretical mesh model. But all these potential mesh models are not supported by our implementation. The empty model (without any connection) can be removed as well as all the mesh models that do not allow to compute connections $(2 \rightarrow 0)$ and $(2 \rightarrow 2)$ that are used into our implementation. Moreover if the mesh model deals with edges, it has to contain connections $(1 \rightarrow 0)$ or $(0 \rightarrow 1)$. Finally it remains only 105 convenient mesh models. In practice, without any change in the source code, our implementation of the Bowyer-Watson algorithm runs onto these 105 mesh models. As expected, the memory footprint and the performance speed depends of the mesh model. Table 4 gathers some results for different mesh models. In this case study, the final triangulation contains $1\,225$ points. With the nine first mesh model, we get the triangulation in less than one second. It was expected since the relations that we use in the algorithm are $(2 \rightarrow 0)$, $(0 \rightarrow 2)$, $(2 \rightarrow 2)$ and $(1 \rightarrow 0)$ (if edges are available or can be built with local topological traversals only). In models 10 to 14, a connection must be computed using a global traversal, while models 15, 16 and 17 require several global traversal. Note that the model 14, which was our first choice, does not provide the best speed performances. It would be with model-specific optimisations.

In order to select a mesh model, computational time and memory footprint have to be taken into account. In Table 4, models 1 to 9 provides an equivalent computational time but have quite different memory footprints. If edges are not necessary to write an algorithm, it is then better to use models 1 or 2. Another criterion to select a mesh model is the scalability. Figure 8 gives the behaviour of three mesh models when the number of points goes from $2\,500$ to $40\,000$. We can deduce that the best balance between computational time and memory footprint is achieved by the model `DIM2|F|N|F2N|N2F|F2F`. It provides the best computational time since it has all the required connections in a direct manner.is equivalent to the model `DIM2|F|N|F2N|N2F|F2F` for memory footprint. In fact, there is a small overhead because the number of connections `F2F` is small in comparison with the other connections.

	Mesh model	Comp. time (s)	Memory (bytes)
1	DIM2|F|N|N2F|F2N	< 1	2 334 720
2	DIM2|F|N|N2F|F2N|F2F	< 1	2 342 912
3	DIM2|F|E|N|F2N|N2F|F2E	< 1	3 526 656
4	DIM2|F|E|N|F2N|N2F|F2E|F2F	< 1	3 547 136
5	DIM2|F|E|N|F2E|E2F|F2N|N2F	< 1	4 038 656
6	DIM2|F|E|N|F2N|N2F|N2E|E2F	< 1	5 672 960
7	DIM2|F|E|N|F2N|N2F|N2E|E2F|F2F	< 1	5 726 208
8	DIM2|F|E|N|F2N|N2F|E2N|N2E|F2E|F2F	< 1	5 332 992
9	DIM2|F|E|N|F2N|N2F|E2F|F2E|E2N|N2E|F2F	< 1	5 902 336
10	DIM2|F|N|N2F|F2F	49	2 314 240
11	DIM2|F|E|N|N2F|N2E|F2F	50	5 136 384
12	DIM2|F|N|N2F	55	2 289 664
13	DIM2|F|N|F2N	90	942 080
14	DIM2|F|N|F2N|F2F	91	987 136
15	DIM2|F|E|N|E2F|N2E	541	4 567 040
16	DIM2|F|E|N|F2E|E2N	2 129	1 454 080
17	DIM2|F|E|N|F2E|E2N|F2F	2 243	1 499 136

Table 4: Computational time and memory footprint for generating a 2d Delaunay triangulation of 1225 points.

5 Conclusion

In this paper we have described an approach allowing users to write multi-model algorithms in the GMDS framework. Some combinatorial terms have been exhibited in order to compute missing connections of a mesh model. We have also showed that some adjacency and incidence relations can not be computed for some mesh models. These combinatorial terms have been implemented in GMDS in order to build at compile-time the appropriate pieces of source code. This feature has been evaluated with implementing a Bowyer-Watson algorithm. The obtained algorithm runs onto 105 mesh models. It shows that an algorithm can be written for a specific mesh model and then be reused with another mesh model.

In this work, we have provided generic computational terms to build adjacency and incidence relations for any mesh models. We have now to test them with larger meshes and to improve their behaviour if necessary. For instance, GMDS is partially based on STL containers for storing the adjacency and incidence relations. Some improvements are possible on this side. Another work is to extend the topological modifications for any kind of cells. In this work, they were only written for triangular meshes.

Eventually, an long-term work will be to provide a tool allowing to analyse a meshing algorithm written with GMDS and to deduce the best model to use according to a given balance between computational time and memory footprint.

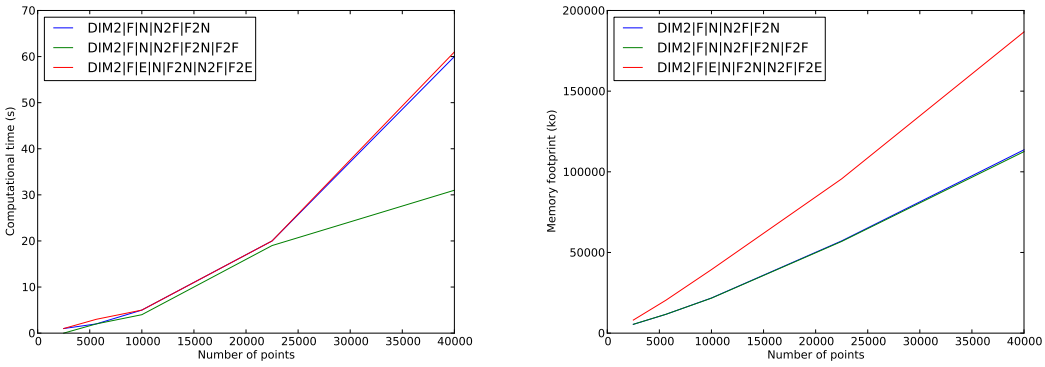


Figure 8: Computational time and memory footprint for generating Delaunay triangulation having 2 500 to 400 000 points with 3 mesh models

References

[1] G. Berti, *Generic Software Components for Scientific Computing*, PhD thesis, Fakultt fr Mathematik, Naturwissenschaften und Informatik der Branderburgischen Technischen Universitt Cottbus, Dortmund, 2000.

[2] M. Botsch, S. Steinberg, S. Bischoff, L. Kobbelt, "OpenMesh – a generic and efficient polygon mesh data structure", in *In OpenSG Symposium*, 2002.

[3] A. Bowyer, "Computing Dirichlet Tessellations", *The Computer Journal*, 24(2): 162–166, 1981.

[4] H. Brönnimann, L. Kettner, S. Schirra, R.C. Veltkamp, "Applications of the Generic Programming Paradigm in the Design of CGAL", in *Selected Papers from the International Seminar on Generic Programming*, pages 206–217. Springer-Verlag, London, UK, 2000, ISBN 3-540-41090-2.

[5] F. Cirak, J.C. Cummings, "Generic programming techniques for parallelizing and extending procedural finite elements program", *Engineering with Computers*, 24: 1–16, 2008.

[6] R.V. Garimella, "Mesh data structure selection for mesh generation and FEA applications", in *International Journal for Numerical Methods in Engineering*, Volume 55, pages 451–478, 2002.

[7] F. Ledoux, Y. Bertrand, J.C. Weill, "Definition of a Generic Mesh Data Structure in HPC Context", *Computation Technologies and Innovation Series*, (26): 49–80, September 2007.

[8] J.F. Remacle, B. Karamete, M. Shephard, "Algorithm Oriented Mesh Database", *International Journal For Numerical Methods in Engineering*, 58: 349–374, 2003.

[9] E.S. Seol, *FMDB: Flexible Mesh Database for Parallel Automated Adaptative Analysis*, PhD thesis, Faculty of Renssealer Polytechnic Institute, 2005.

[10] M.S. Shephard, E.S. Seol, "Flexible Distributed Mesh Data Structure for Parallel Adaptive Analysis", *Advanced Computational Infrastructures for Parallel and*

Distributed Adaptive Applications, pages 1–38, 2007.

[11] T. Tautges, C. Ernst, K. Merkley, R. Meyers, C. Stimpson, "Mesh Oriented datABase (MOAB)", 2005, http://cubit.sandia.gov/cubit.

[12] D. Watson, "Computing the n-dimensional Delaunay tessellation with applications to Voronoi polytopes", *The Computer Journal*, 24(2): 167–172, 1981.