Quadtree techniques for the construction of structured auxiliary meshes

Julien Renggli
Professor : Dr. Ralf Hiptmair

Diploma Thesis, Winter term 2005-2006
Seminar of Applied Mathematics

March 1, 2006

The auxiliary space preconditioning presented in [1] and adapted in [3] to $H(curl;\Omega)$-elliptic variational problems involves the construction of an auxiliary mesh with similar properties than the unstructured mesh to solve. This auxiliary mesh can be constructed using a quadtree method, which will subdivide the space in smaller regions until the required conditions are met. The aim of this thesis is to find an algorithm able to handle that problem.

Contents

List of Figures

List of symbols and abbreviations

2D	Two dimensional space
3D	Three dimensional space
STL	The Standard Template Library
$\mathcal{T}_h$	The unstructured mesh, given as input
$\mathcal{T}_a$	The unstructured mesh, output of the program
IN	The status of a node *inside* the mesh $\mathcal{T}_h$
OUT	The status of a node *outside* the mesh $\mathcal{T}_h$
NDEF	The status of a node *on the boundary* of the mesh $\mathcal{T}_h$
V	The vertices of $\mathcal{T}_h$
E	The edges of $\mathcal{T}_h$
S	The polygons of $\mathcal{T}_h$
T	The polyhedrons of $\mathcal{T}_h$

1. Introduction

1.1. Motivation

An approach for solving partial differential equations requires a multigrid method based on auxiliary meshes as *auxiliary space preconditioning*. This has already been proposed for the Dirichlet variational problem

$$-\triangle u + u = f \quad \text{in } \Omega, \qquad u = 0 \text{ on } \Omega$$

in [8], and more recently extended in the case of an $H(curl;\Omega)$-elliptic variational problem

$$\mathbf{curl}\ \mathbf{curl} u + u = f \quad \text{in } \Omega, \qquad u_t = 0 \text{ on } \partial\Omega$$

in [3].

Both preconditioners rely on well constructed auxiliary meshes, based on an originally unstructured mesh. Some conditions must be answered to consider the former correctly constructed. The auxiliary mesh must be composed of axis-parallel rectangles, eventually split into two identical triangles connecting the lower left corner to the upper right, extended to cuboids when working in three dimensions. It should cover as much as possible of the space involved in the problem, without overlapping the original mesh though. Finally, the meshwidths should be of the same size over the whole space, therefore requiring local refinements to be present in the auxiliary mesh everywhere it appears in the unstructured mesh.

1.2. Approach

The construction of such an auxiliary mesh $\mathcal{T}_a$ can be made by different techniques. A possible method would be to determine first the mean meshwidth of the original mesh. Then a regular grid would describe the possible new one. Each of its constitutive cell would be tested and identified as beeing inside, on the boundary or outside of the problem. Remaining cells would finally be merged or split to keep local meshwidth requirements. In order to conserve local refinements that step might require to be called recursively, in a way, like building a tree. It seems natural to use the tree form from the beginning, avoiding calculation of a mean meshwidth value which does not mean anything.

Another possibility could be to construct the smallest rectangle (cuboid) around each cell of the unstructured mesh, with meshwidth value stored for each of the so constructed shape. Then the new shapes could be summed, the intersections leading to the auxiliary mesh. However, the original mesh is mainly composed of triangles, not necessarily

parallel to the axes. The surrounding rectangle then covers more than twice the area of a triangle (in three dimensions, the problem is similar with tetrahedrons and cuboids instead). This and the difficulty of finding a correct way for summing the shapes, makes that solution awkward.

The algorithm presented in this thesis uses the idea of a tree, whose leaves will constitute the basis for an auxiliary mesh generation. Using a quadtree, or octree when dealing with a three dimensional problem, to subdivide the space in four equal subspaces independent of each other, allows a logarithmic processing. The original mesh must be provided by any mesh generator able to write it in a format as described in appendix A.1. The generator used to test the program described in this thesis is Netgen [6], but other should work as well.

The tree-based approach allows to quickly eliminate large regions of space which are not part of the problem, while large regions inside it, which are known not to contain any boundary, are refined very fast without further treatment. Only regions on the boundary of the unstructured mesh require a robust algorithm able to handle any possible case. Most of the time is spent there, to determine where the boundary splits inside from outside.

1.3. Overview

Data structures used by the program will be described in chapter 2. Chapter 3 will present the algorithm itself, and its complexity will be theoretically analysed in chapter 4. A few examples will follow in chapter 5, showing real values for the time needed, a comparison of both meshes and a visual comparison of them.

2. Data structures

2.1. Programming aspects

The program has been developed in C++, using the Standard Template Library [7] as containers. Some terms in the rest of this thesis are directly derived from those defined in the STL documentation.

- *Vectors* are like arrays of objects, each one stored just after the preceding. Accessing any element in the vector, adding or removing an element at the end of the vector are operations that can be done in constant time, whilst inserting or deleting an element at any other position in the vector requires to move all subsequently elements, leading to linear time.
- *Lists*, on the contrary, contain elements which might be located anywhere in the memory. Insertion and deletion at any place is done in constant time, whilst access to any element requires linear time; just the opposite of the vector.
- *Sets* and *Maps* are almost the same containers, and store elements in a tree. Accessing one is therefore done in logarithmic time.
- *Iterators* are broadly used in the program; they extend the concept of C++ *pointers* for containers of the STL. The term *pointer* comes from the idea that they are like an arrow that *points* towards an object in the memory, like an index *points* towards an element in an array. To avoid confusion, it will not be made mention of iterators in this thesis.

Note that the numbering in the vectors lead to some confusions because of the C++ nature of the program. In the files, as in this thesis, indices go from 1 to n. In C++, they instead go from 0 to $n-1$. In itself the change is not too big, but this fact requires constant attention when dealing with the source code.

2.2. Mesh storage

The original mesh $\mathcal{T}_h$ is stored in a file of format **.vol* (see appendix A.1 for a more detailed view of that format). The program will read it, create an auxiliary mesh $\mathcal{T}_a$, and store it in an **.am* file, whose format is very similar (see appendix A.2) than the former one. Both meshes exist in the program as instantiation of classes; so much the files are very similar, so much the program stores the meshes in two totally different ways which will be reviewed here.

The unstructured mesh $\mathcal{T}_h$ is, as in the file version, divided in different lists. They are the same for two or three dimensions, but don't represent exactly the same objects.

In three dimensions, a list of all volumes $K \in \mathcal{T}_h$ describing the mesh is stored. These volumes might be of any form, but consist normally only of tetrahedrons when generated by Netgen. A second list representing the surfaces $S \in \partial K$ completes the first one. Not all surfaces S are relevant and therefore present, but only those who are also on the boundary $\partial\mathcal{T}_h$ of the problem. As it will be seen in the next chapter, that property is very important since regions on the boundary $\partial\mathcal{T}_h$ are the most time-consuming part of the whole algorithm, and the union of all S covers entirely $\partial\mathcal{T}_h$. A third list consists of edges E which are the intersection between two surfaces. Finally, a vector of vertices V_i is stored, each vertex representing a point in the space.

In two dimensions, the first list remains void. The second contains all surfaces $S \in \mathcal{T}_h$ describing the unstructured mesh, as if they were the boundary of an infinitesimally thin volume on the xy-plane. The z value is set to 0. The third list consists of the edges E of the surfaces, but only those on the boundary $\partial\mathcal{T}_h$ of the problem. As well the surfaces S were used by the algorithm for three dimensional problems, as well in 2D those edges E help the program do its job correctly. Finally, the vector of vertices V_i is the same as in three dimensions.

All different shapes in the first three lists are actually very similar, and consist of a number n of apices and of a vector of indices to these apices (see appendix B.2). Those indices point to the latter vector, where the real vertices V_i are stored. This way, they can be shared by multiple surfaces at a time without storing multiple copies of the same point.

The auxiliary mesh $\mathcal{T}_a$ is, due to the way it is created, present in the memory in the form of a tree. Such a tree is build on *nodes*, which contain four or eight *children nodes* each, that here split the space in equal subspaces and contain their own children, and so on. The tree begins with a single *top node*, and goes as deeply as needed, until it reaches a *leaf*. In the case of the auxiliary mesh, the *top node* covers just enough space to surround the whole problem, and its leaves contain all relevant information about the new mesh to build. A node N covers a portion of space with the shape of a rectangle (or cuboid), which is defined by only two points : its lower left (back) and upper right (front) corner, which set respectively the minimum and maximum values a point p might have along each axis so that $p \in N$. A node N has also a status, which can be one of the following :

$$\text{IN} \quad : \quad \forall p \in N, p \in \mathcal{T}_h \tag{2.1}$$

$$\text{OUT} \quad : \quad \forall p \in N, p \notin \mathcal{T}_h \tag{2.2}$$

$$\text{NDEF} \quad : \quad \exists p_i, p_j \in N | p_i \in \mathcal{T}_h, p_j \notin \mathcal{T}_h. \tag{2.3}$$

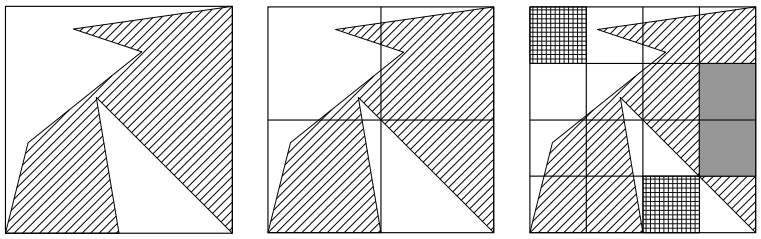
Finally, a node is called a *leaf*, when it satisfies the condition that it has no child. In this program, that condition is verified very quickly : there should not be more than one vertex V_i inside each node. This method has given up here very good results, as the examples in chapter 5 will show.

There is now enough information to create the auxiliary mesh $\mathcal{T}_a$: for all nodes N^i with status IN (or NDEF in some cases, see next chapter for further explanations) and meeting requirement to be a leaf, the node is a valid element K_a^i and it satisfies $\sum K_a^i = \mathcal{T}_a$.

3. Algorithm

The idea behind the algorithm used in the generation of an auxiliary mesh $\mathcal{T}_a$ is quite simple. The dimension of the problem, in this case two or three, does not change the concept, but leads to some major changes in the implementation.

Starting from a *top node* that encloses just enough space to surround the original mesh $\mathcal{T}_h$ totally (figure 3.1 a)), The space is split in smaller regions totally independent of each other (figure 3.1 b)). The only relevant information that is needed is the boundary $\partial\mathcal{T}_h$, that is stored on the form of edges E or surfaces S. As long as a node is crossed by some part of the boundary, its status is by definition 2.3 NDEF. When splitting it in equal subregions though, it will appear that some of the children are not near the boundary any more : they become either IN or OUT (definitions 2.1 and 2.2). Since the only known information is on the form of edges or surfaces, the status of each child must be found at the same time that the splitting is done : the *parent* must decide the status of each of its children. As long as the status remains unchanged, nothing is to be done; an NDEF child will still have some boundary region in it for further use. An IN node will set strictly all of its children to IN, and an OUT node will not be refined any further since it is of no interest (note that by definition it can't contain any vertex V_i, which makes it automatically a *leaf*).



a) Initialisation b) 1st step c) 2nd step

Figure 3.1.: On the *left* the *top node* encloses the unstructured mesh, hatched. In the *middle*, the four children of the top node are created in the first refinement step. One more step, and two nodes are already totally outside the problem; they will thereafter be ignored. Two other nodes are fully inside, in grey.

A node N, as framed in the previous chapter, can now be completed with these new informations. It must have :

1. A status : IN, OUT or NDEF.
2. A size and position : minimum and maximum values defining a rectangle or a cuboid.

3. A vector of all vertices $V_i \in N$.

4. A vector of all boundaries $\{E|E \cap N \neq \emptyset\}$ or $\{S|S \cap N \neq \emptyset\}$.

The algorithm is sketched in a *flow chart* (figure 3.2). The list begins with only the *top node* inside it, and it continues until no more node is on the list. The idea of the algorithm is the same for both two and three dimensions, except for the number of children (4 or 8) and the type of boundary (E or S). In the implementation, though, there is one part that differs totally between both versions : the way a child is determined to be IN or OUT (down left in the *flow chart*), due to the nature of the boundary that also increases in dimension. But even there the idea is similar : there are only two sides (*left* and *right* for edges in a plane, *up* and *down* for surfaces in a volume), one where points are inside $\mathcal{T}_h$, the other where they are outside. The goal is then to have a quick way to decide on which side a node lie. For this the concept of *orientation* is introduced.

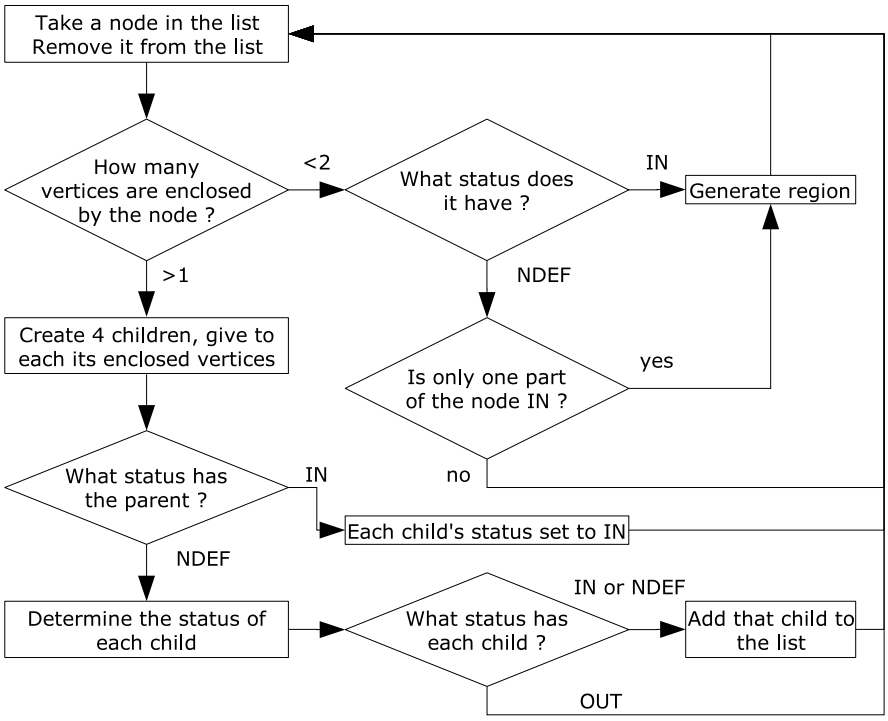


Figure 3.2.: The basic algorithm for the quadtree. The one for the octree is the same but with eight children instead of four.

In 2D Any edge E is whatsoever defined by strictly two vertices, namely $v_{E,1}$ and $v_{E,2}$, in the xy-plane π. For the rest of the thesis, it will be assumed that they are always orienter from $v_{E,1}$ to $v_{E,2}$, thus resulting in a vector $\overrightarrow{v_{E,1}v_{E,2}}$. For any point p in the plane, the vector $\overrightarrow{v_{E,1}p}$ is also required. The dimension of the problem is then increased

by one, so that the vector product $\wedge$ exists; the third coordinate is set to zero, and it satisfies

$$\overrightarrow{v_{E,1}v_{E,2}}' = \begin{pmatrix} x_1 \\ y_1 \\ 0 \end{pmatrix}, \overrightarrow{v_{E,1}p}' = \begin{pmatrix} x_2 \\ y_2 \\ 0 \end{pmatrix} \Rightarrow \overrightarrow{v_{E,1}v_{E,2}}' \wedge \overrightarrow{v_{E,1}p}' = \begin{pmatrix} 0 \\ 0 \\ z_3 \end{pmatrix}.$$

The resulting vector has both its x and y coordinates set to zero, only the third one contains relevant information. A new operator $\otimes$ is therefore introduced, which returns the z value of the vector product :

$$\overrightarrow{v_{E,1}v_{E,2}} \otimes \overrightarrow{v_{E,1}p} = x_1y_2 - x_2y_1 = z_3. \tag{3.1}$$

The orientation is now chosen so as for any edge E there are two ensembles L_E and R_E (for *left* and *right* respectively) such that

$$\begin{aligned} L_E &= \{p \in \mathbb{R}^2 | \overrightarrow{v_{E,1}v_{E,2}} \otimes \overrightarrow{v_{E,1}p} > 0\} \text{is the ensemble of points inside } \mathcal{T}_h \\ R_E &= \{p \in \mathbb{R}^2 | \overrightarrow{v_{E,1}v_{E,2}} \otimes \overrightarrow{v_{E,1}p} < 0\} \text{is the ensemble of points outside } \mathcal{T}_h. \end{aligned} \tag{3.2}$$

Of course, this assumption remains valid only in a small area around each edge. Figure 3.3 b) shows an example of correctly oriented edges, and the reason for the names *left* and *right*.

In 3D A surface S is defined by minimum three vertices, but might be as well a polygon of almost infinite apices. Since meshes generated by Netgen are made of triangles, the assumption is made that any mesh consists only of triangles. Anyway, a convex polygon from which three consecutive apices are extracted results in a triangle totally inside of that polygon. Creating such a virtual triangle from such a polygon is what is implicitly done and the assumption remains correct in that case. For non convex polygons, though, it is no more valid and might result in partially or totally wrong meshes $\mathcal{T}_a$.

The vertices of the triangle are named $v_{S,1}, v_{S,2}$ and $v_{S,3}$, and it forms a plane π_S with normal vector $\overrightarrow{n_S}$ where

$$\overrightarrow{n_S} = \overrightarrow{v_{S,1}v_{S,2}} \wedge \overrightarrow{v_{S,1}v_{S,3}} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

and

$$\pi_S : ax + by + cz + d = 0.$$

Now for any point $p = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$, the distance between p and π_S satisfies

$$\delta_{\pi_S}(p) = \frac{ax_1 + by_1 + cz_1 + d}{\sqrt{a^2 + b^2 + c^2}} \tag{3.3}$$

which can be reformulated as

$$\delta_{\pi_S}(p) = \frac{\overrightarrow{v_{S,1}p} \cdot \overrightarrow{n_S}}{\sqrt{a^2+b^2+c^2}}. \tag{3.4}$$

Both equations are given here, since they both appear from time to time in the algorithm depending on whether the plane π_S or the normal vector $\overrightarrow{n_S}$ is known.

The orientation is now chosen such that for any surface S in the plane π_S there are two ensembles U_S and D_S (for *up* and *down* respectively) such that :

$$\begin{aligned} U_S &= \{p \in \mathbb{R}^3 | \delta_{\pi_S}(p) > 0\} \text{is the ensemble of points outside } \mathcal{T}_h \\ D_S &= \{p \in \mathbb{R}^3 | \delta_{\pi_S}(p) < 0\} \text{is the ensemble of points inside } \mathcal{T}_h. \end{aligned} \tag{3.5}$$

Of course, this assumption remains valid only in a small area around each surface.

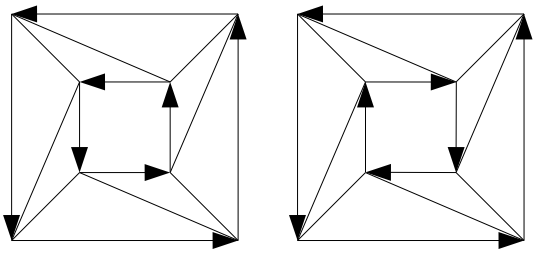
Now the algorithm is split in three main parts to analyse the way it creates the mesh. First the initialisation phase where the original mesh $\mathcal{T}_h$ is loaded and $\mathcal{T}_a$ prepared is sketched. Then the creation of the tree, which is the core of the program, is explained in two separate sections, one for each dimension. In the third part, the generation of the mesh is discussed. Finally, a few words are added to explain how the program behaves when more than one mesh $\mathcal{T}_h$ are given in the input file.

3.1. Initialisation

The input file is first read and the unstructured mesh $\mathcal{T}_h$ loaded. Some preprocessing will then take place. First, a loop over all edges E, respectively surfaces S, sets all their apices as *boundary* to indicate they belong to $\partial\mathcal{T}_h$. Then, $\partial\mathcal{T}_h$ itself must be oriented everywhere the same way, since this feature is required in order for the algorithm to work correctly. This might have been already done by the mesh generator, but it is so crucial to the program that it must be verified before going further. Actually, even though Netgen orients the edges and surfaces it generates, it is not always intuitively done, as shown in figure 3.3.

The previous section introduced the notion of orientation for both edges and surfaces. It is now time to put it in practice, assuming the boundary is randomly oriented. Once again, the two possibilities depending on whether the boundary is made of edges or surfaces are separated to avoid confusion, even though the idea is very similar.

In 2D Each edge must belong to one and only one surface, since the mesh is not degenerated. Actually, section 3.6 will nullify that assumption when more than one mesh $\mathcal{T}_h$ exists, but for the moment it is assumed only one mesh is present, in order to simplify things. A loop is therefore executed over all surfaces S, looking for edges. Every time an edge E is found, a vertex that is an apex of S but does not belong to E, is stored as $v_{\neg E}$. This vertex is important since it already exists, meaning no calculation



a) Wrong b) Correct

Figure 3.3.: A mesh with a hole in the center. On the *left*, the edges on the outside are correctly oriented by Netgen, not those around the center. On the *right*, all lines are oriented so that on their left side is the mesh, on their right is nothing.

to find it, and it is known for sure to be on the *right* of the considered edge, inside $\mathcal{T}_h$. Using definition 3.2, the edge is oriented so that it satisfies

$$\overrightarrow{v_{E,1}v_{E,2}} \otimes \overrightarrow{v_{E,1}v_{\neg E}} > 0$$

and the next edge can be searched.

In 3D Each surface belongs to exactly one volume, since the mesh is not degenerated. The same restriction than above applies, which will be discussed in section 3.6. The loop is then executed over all volumes K, and boundary surfaces S are searched. Again, one vertex that is an apex of K but not of S will be stored as $v_{\neg S}$. Since it already exists and is known to be inside the unstructured mesh, it plays the role of $v_{\neg E}$ in the previous paragraph. This time, $v_{\neg S}$ is on the *down* side of the surface. Using 3.5 as reference, S is oriented so that it satisfies

$$\begin{aligned} \overrightarrow{n_S} = \overrightarrow{v_{S,1}v_{S,2}} \quad &\wedge \quad \overrightarrow{v_{S,1}v_{S,3}}, \\ \overrightarrow{v_{S,1}v_{\neg S}} \cdot \overrightarrow{n_S} \quad &< \quad 0. \end{aligned}$$

S being any polygon with at least three apices.

Now that the boundary is correctly oriented, it can be given totally to the *top node*, since, by definition, that node surrounds $\mathcal{T}_h$, that is $\partial\mathcal{T}_h$ too. For the same reason, every vertex will also be given to the top of the tree. Doing so, the actual position and size of the *top node* can be obtained by finding the extremal values the vertices have. This could have been found more efficiently as the mesh was loaded or even better when the edges were reoriented, but the possibility that more than one mesh exist (section 3.6) forbids it. Anyway, not much time is lost there.

The tree has at this point every bit of information it needs to be built. The refinement can then take place, following the rules sketched in the *flow chart*, figure 3.2. The two and three dimensional cases will be separately analysed in the next two sections.

3.2. 2D version

From previous arguments, a node N of status NDEF contains at least one edge that crosses it. If the number of vertices $V \in N$ is lesser or equal to one, then the node is considered a *leaf* and the algorithm goes to the next node. Is this not the case, the node must be split in four equal rectangles, as figure 3.1 shows. The vertices are distributed to the child in which they each fit (a vertex exactly in the middle of the node would not be distributed at all since on the boundary of all children...). Then, each edge E of the *parent* N must be distributed to all *children* N' following the rule $E \cap N' \neq \emptyset$. This can be easily verified first by trivial tests : if the edge starts or finishes inside N', there is obviously an intersection. If both apices are outside the limits of the node (both on the same side), there can't be any intersection.

All other cases are solved using a Cohen-Sutherland clipping algorithm [5], which can be understood the following way : given an edge E to be clipped to a node N', find all intersections between E and the left, right, down and top edges of the rectangle defining N'. That is, be

$$v_{E,1} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, v_{E,2} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

the end points of E, and $x = x_{clip}$ the left edge of the node; it is assumed that $x_1 \gtrless x_{clip} \gtrless x_2$. The intersecting point is obtained by

$$v_{new} = \begin{pmatrix} x_{clip} \\ y_1 + \frac{(y_2 - y_1)(x_{clip} - x_1)}{(x_2 - x_1)} \end{pmatrix}. \tag{3.6}$$

The same is applied to the right edge, and the role of x and y coordinates is inverted for top and down edges. Is the resulting segment inside the area of the rectangle, then the edge crosses the node. Is there no such segment, the edge doesn't cross the node.

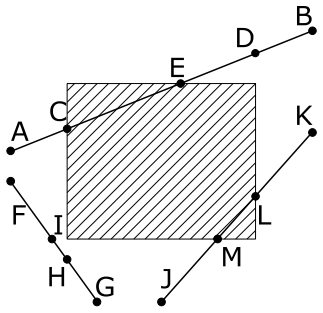


Figure 3.4.: An example for the Cohen-Sutherland clipping algorithm. The segment [AB] is clipped on C, then on D and finally on E; the segment [CE] is a true clipping. The same for the segment [JL] which returns segment [LM]. On the contrary, segment [FG] does not intersect the rectangle.

A special case happens when the intersection between the edge and the node lie in whole on the edge of the node. The clipping would suggest that there is an intersection,

but the algorithm must correct that result. In figure 3.4 for example, the edges number 42 and 43 lie on the boundary. Would they not be rejected at last, the result would be that for node C, $\exists E \in C$ and therefore would not gain status IN.

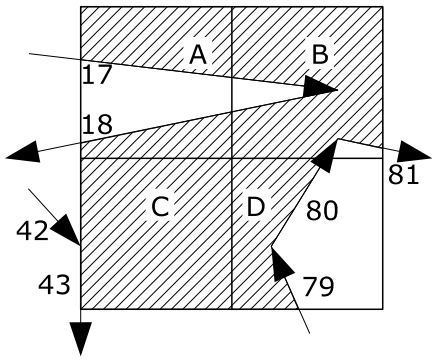


Figure 3.5.: Example of a node with its edges (numbered) and children (lettered). The result of the clipping is that child A receives edges 17 and 18, child B receives 17, 18, 80 and 81, and child D receives edges 79 and 80. Child C has no intersection, thus is IN since located in the left.

When all edges have been distributed, the algorithm will be able to determine the status of each child N'. All of them that have at least one intersection with an edge will by definition have their status set to NDEF. The other have equal chances to be IN or OUT, depending on their position relative to the edges in the *parent*. There comes the explanation why the edges were first orientated : on the left are nodes of status IN, on the right with status OUT. Obviously, when a child does contain at least one vertex inside it, it can't have status OUT, thus is IN; this happens but not always. Moreover, not all meshes are convex which complicates a little bit the task. An extreme example of what might happen is given in figure 3.5. Child C has no intersection, not even with edges 42 or 43. Remembering 2.1 and 3.2, any point $p_C \in C$ will suffice to determine the status of the whole node. Here it should satisfy

$$\overrightarrow{v_{E,1}v_{E,2}} \otimes \overrightarrow{v_{E,1}p_C} > 0.$$

The problem is that taking a random edge, or more likely the edge with the highest or lowest number, might result in wrong results. For example with edge *17* :

$$\overrightarrow{v_{17,1}v_{17,2}} \otimes \overrightarrow{v_{17,1}p_C} < 0$$

which is obviously wrong.

Another rule is needed to find the correct edge to work with. The argument is the following : the unstructured mesh $\mathcal{T}_h$ is compact. This implies that its boundary $\partial\mathcal{T}_h$ is closed and continuous. Thereafter, the edge nearest a given point p_C will define the correct orientation for p_C. That point p_C is taken as the common corner of the parent

N and the chosen child N', on the example the left down corner. The chosen edge will be either 18 or 79, both being equally correct.

It is to mention that edges 42 and 43, if they are rejected by the algorithm, must somehow be conserved. Actually, they are needed in order to compute the constant C_δ, so they are stored in a special vector to allow that. Otherwise, they are treated like normal vectors.

A final remark : here, the *parent* was considered as of status NDEF, since it contained edges. It has also be made mention that those of status OUT are not refined since they are of no importance and can't contain any vertex. On the contrary, nodes that are defined as IN are the most important since they compose the auxiliary mesh $\mathcal{T}_a$; they must be refined until *leaves* are reached, but since they contain no edge and their *children* cover the same region as them, this whole section will not be necessary, and all children will automatically receive status IN.

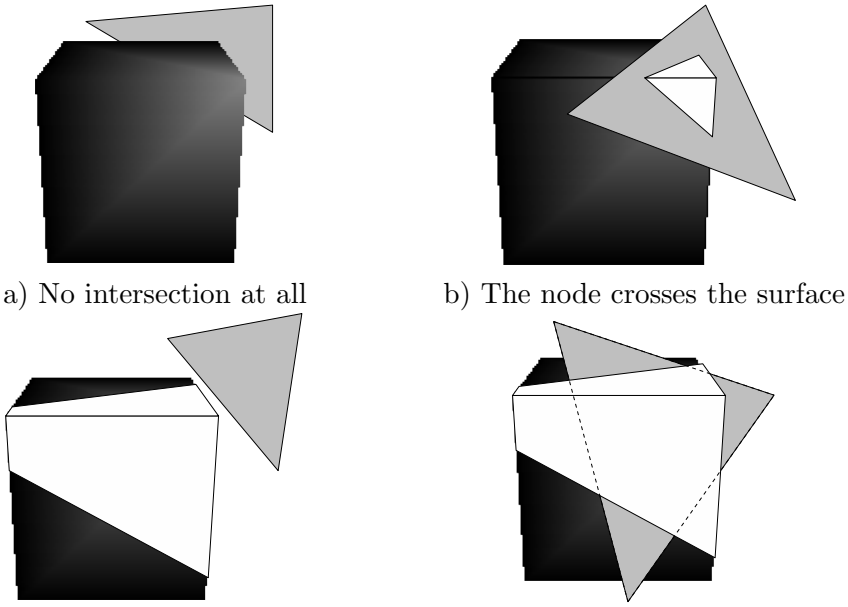
3.3. 3D version

This version is similar to the previous one, with surfaces instead of edges. The total number of children is also increased to eight, and each node has the form of an axis-parallel cuboid defined by two vertices : left down back and right up front. The goal is once again to distribute vertices and boundary to the children, in the most efficient way possible. For vertices, a simple comparison with extremal values is sufficient. For surfaces S, the solution is not as simple as in two dimensions. To determine if there is an intersection between S and a children node N', the trivial cases are first verified : either there is at least one $v_{S,i} \in N'$, and the intersection exists, or all $v_{S,i}$'s are on the same side of the node, meaning the surface is totally outside the child.

Using definition 3.5 and equations 3.3 and 3.4, it is possible to find any other possible intersection. The different possibilities are shown in figure 3.6 : on the left there is no intersection; on the right, the two types of real intersections are shown. In b) the node crosses the surface. That means, the intersection between at least one edge of the node and the surface S is non void. In d), the surface S crosses the node; the intersection between an edge of S and one side of the node is non void. Of course, a mixing of both possibilities is very likely to happen.

To algorithmically separate the possibilities shown, each one of the eight node's corners are tested against the plane π_S. If all of them are positive (or negative), then situation a) occurs. Is it not the case, then one of the three other situations is met. The last one (d) is the easiest to verify, using a Liang-Barsky clipping algorithm [4] in three dimensions. It is similar than the Cohen-Sutherland presented before, except for a few details. Any good computer graphics book like [5] or [2] should cover that topic for better understanding the different types of clipping algorithms.

When no intersection is found after that test, it is not possible to immediately conclude $S \cap N' = \emptyset$. On the contrary, situation b) might come true, so it must still be tested. Every node is axis-parallel; this implies that its edges are parallel to one of the axes.



a) No intersection at all

b) The node crosses the surface

c) Intersection in π but not with surface

d) The surface crosses the node

Figure 3.6.: Up left, the node N is totally on one side of the plane π_S. Down left, some points are up the plane, some down, but there is still no intersection between S and N. On the right, the two types of cases that might happen (or a mix between both) : either the edges of the node cross the surface (up), or the edges of the surface cross the node (down).

Projecting the whole problem to a plane perpendicular to that axis reduces that edge to a single point. The surface S itself will be projected to a two dimensional triangle, or a degenerated line. The latter will not be used any further. The problem is now reduced to verifying if a point is inside a triangle. Be p the point, projection of the edge, and $v'_{S,1}, v'_{S,2}$ and $v'_{S,3}$ the projected apices of the surface S.

Suppose a projection on the xy-plane π_{xy}. Be p' the projection of the edge $\mathbf{E}$ to a single point, and $v'_{S,1}, v'_{S,2}$ and $v'_{S,3}$ the projected apices of the surface S on that plane, forming surface S'. Then labelling O as the centre of origin of the plane, there is :

$$\overrightarrow{v'_{S,1}v'_{S,2}} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \overrightarrow{v'_{S,1}v'_{S,3}} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, \overrightarrow{v'_{S,1}p'} = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$$

and $p' \in S' \equiv \mathbf{E} \cap S \neq \emptyset$ if and only if $\exists \alpha, \beta \geq 0, \alpha + \beta \leq 1$ such that

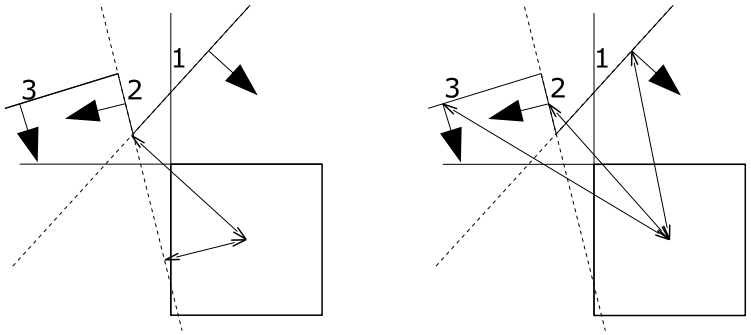
$$\overrightarrow{Op'} = \overrightarrow{Ov'_{S,1}} + \alpha \cdot \overrightarrow{v'_{S,1}v'_{S,2}} + \beta \cdot \overrightarrow{v'_{S,1}v'_{S,3}}. \tag{3.7}$$

Like in two dimensions, there are now some children who have at least an intersection with the boundary, thus of status NDEF, and some who must be decided to be either IN or OUT. As in the previous section, children that have at least one vertex V_i in them automatically gain status IN. For the other, argument 3.5 is invoked, and the nearest surface $S \in N$ must be chosen to find the correct plane π_S to work with. Unfortunately, finding that surface is not as easy as it was to find the correct edge. The intersection $S \cap N$ results in a segment and is much more complicated to extract the nearest surface. Figure 3.7 shows two ideas of choosing the closest surfaces to a node (projected in two dimensions) : on the left, equation 3.3 is used, thus picking surface number 2; on the right, the smallest distance between the centre of the node and the centre of gravity of the surface is searched, which again returns surface number 2.

The solution implemented, but that alas does not function in all cases, is slightly different than what is illustrate here. It draws axis-parallel lines from the centre of the node and looks for intersections, as if the line was an edge in figure 3.6 b). Using equation 3.7 as reference, it is searched for an intersection between the line and any surface S. If such an intersection exists, the distance between it and the middle point of the node is stored; the surface corresponding to the nearest intersection will be taken in account.

In case no intersection has been found along all axes, the nearest edge with respect to its centre of gravity will be chosen. This happens from time to time, and this solution will not ensure correct results when meshes are not quite convex. But it is still more reliable than the other possibility, the distance between the node and the plane π_S.

Either way, the closest surface S that has been chosen will return the status of the node using equation 3.4, without caring about the normalisation factor : all that is of interest is the sign of δ. Is it positive, the child has status OUT; is it negative, then IN. In most examples tested with the program, the resulting auxiliary meshes $\mathcal{T}_a$ were correctly created. Only one very irregular mesh had nodes generated where they should not have been, see chapter 5.



a) Point nearest to plane b) Point nearest to center of gravity

Figure 3.7.: Two projections of the 3D problem, with the child to test on bottom right. On the left, the middle point of the node is tested against the plane of all triangles; the nearest is surface #2, which will result in wrong results. On the right, the middle point of the node is tested against the center of gravity of all triangles; the nearest is again surface #2.

3.4. Mesh generation

As the algorithm reaches a *leaf* of the tree, it will not refine the mesh any further. But the node is already being treated, so another function is called. That function has the purpose of generating the auxiliary mesh $\mathcal{T}_a$. Nodes with status OUT are not of concern, since they are totally outside mesh $\mathcal{T}_h$. Those with status IN are on the contrary very interesting. The program then creates the corners of the node. As for the original mesh, they might be shared two or more other nodes. Therefore, it searches in a map whether they already exist. If yes, the index to that corner is returned. If no, the corner is put inside the map and the new index is returned.

With the indices returned, surfaces are created the same way it has been done up here : a triangle is defined by an index to each of its three corners, a cuboid by an index to each of its eight corners. If an *xml* file is wanted, all relevant informations are written in the file. Nothing else is written for the moment, it is just stored in a vector.

Note that in two dimensions, it is possible for nodes with status NDEF to be partially generated, since a node is rectangular and $\mathcal{T}_a$ consists of triangles : it might be that the lower (or upper, both are possible) triangle should be considered with status IN, while the edges cross the rest of the node. This is tested every time a node of status NDEF is also a *leaf*.

When the tree is finished, the vector is post-processed and the final file can be written, with first all surfaces or volumes, and then all apices listed. The result can be directly used in the preconditioners or compared with the original mesh visually.

3.5. Computing constants

Additionally to the main algorithm, a tool to determine some constants can be called with the `--C` argument in the command line (see appendix C for more informations about command line arguments). Computed constants are :

- ρ : a parameter representing the shape of any triangle (tetrahedron) K in the original mesh. It compares the length h_K of the longest edge to the radius r_K of the inner circle (sphere) of any K in mesh $\mathcal{T}_h$:

$$\rho(\mathcal{T}_h) := \max_{K\in\mathcal{T}_h} \frac{h_K}{r_K}, \qquad \begin{array}{l} h_K := \max\{|\mathbf{x}-\mathbf{y}| : \mathbf{x},\mathbf{y} \in K\} \\ r_K := \max\{r>0 : \exists \mathbf{x} \in K; |\mathbf{x}-\mathbf{y}| < r \Rightarrow \mathbf{y} \in K\}. \end{array}$$

  The smallest value it has, the less degenerate the triangles are.

- C_a is the smallest positive constant for which

$$C_a^{-1} h \leq h_a \leq C_a h$$

  is valid $\forall K_a \in \mathcal{T}_a$; $h_a := \max\{|\mathbf{x}-\mathbf{y}| : \mathbf{x},\mathbf{y} \in K_a\}$, $\quad h(\mathbf{x}) = h_K \forall \mathbf{x} \in K$.

- C_∂ requires first some definitions :
  - $\partial\mathcal{T}_h$ is the boundary of the unstructured mesh.
  - For $\mathbf{p} \in \partial\mathcal{T}_h$,

$$h_\mathbf{p} := \frac{1}{\sharp\mathcal{T}_\mathbf{p}} \sum_{K\in\mathcal{T}_\mathbf{p}} h_K, \quad \mathcal{T}_\mathbf{p} := \{K \in \mathcal{T}_h : \mathbf{p} \in \overline{K}\}$$

    is the local meshwidth at $\mathbf{p}$
  - B is the boundary strip (see figure 3.8) :

$$\overline{B} := \bigcup \{\overline{K} \in \mathcal{T}_h : K \nsubseteq \mathcal{T}_a\}$$

  then C_∂ is defined as the smallest constant > 0 such that

$$B \subset \bigcup_{\mathbf{p}\in\partial\mathcal{T}_h} B_\mathbf{p}, \quad B_\mathbf{p} := \{\mathbf{x} \in B : |\mathbf{x}-\mathbf{p}| < C_\partial h_\mathbf{p}\}.$$

3.6. Multiple meshes

The algorithm described before works correctly for simple meshes where only one surface (volume) is described, with maybe some "holes" like in figure 3.3. A new problem arises when the unstructured mesh represents for example two different materials, one inside the other. There, the boundary between as defined by Netgen can't be seen as a real boundary any more. In 2D, as in figure 3.9, edges can in such a case be shared between

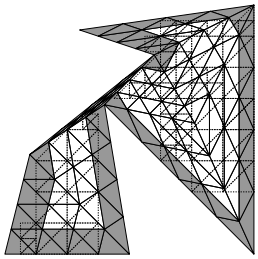


Figure 3.8.: The strip B (shaded) for some mesh $\mathcal{T}_h$.

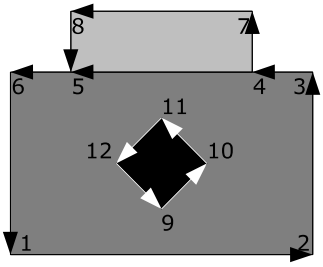


Figure 3.9.: Three different meshes : light grey, grey and black. Edges 4-5 are boundary to both grey and light grey but not boundaries of the whole. The same for 9-10, 10-11, 11-12 and 12-9.

two different meshes, thus not be located on the real boundary $\partial\mathcal{T}_h$. When considering each mesh separately, those edges are on the local boundary; this is not true any more for the whole mesh.

The meshes are therefore handled separately, and the initialisation has to be slightly modified. The different meshes have different surface numbers, and this can be exploited to tell each node to whom mesh it belongs. Edges (and surfaces) on the real boundary will still be oriented the same way. All other are oriented with respect to the mesh with the lowest number, but will know as well to with other mesh they belong (obviously they can't belong to more than two meshes at a time). Each auxiliary mesh is generated from the lowest number to the highest, and any time an edge is called for the second time, its orientation is inverted. For example, in figure 3.9, let's say the grey mesh is called first (has the lowest mesh number). Edge 4-5 is correctly oriented for the moment. When light grey mesh is treated, edge 4-5 will have its orientation inverted and the corresponding auxiliary mesh can be created.

The 3D problem is handled the same way, each surface belonging to at maximum two volumes.

4. Complexity analysis

The algorithm are analysed here for their computational cost, first theoretically, then experimentally. It is assumed only one mesh for simplicity. The variables used here are :

- E : The number of edges.
- S : The number of surfaces.
- T : The number of tetrahedrons.
- V : The number of vertices (apices).

which can vary on number depending of the problem handled.

The analysis is separated in different parts as seen in the main file of the program.

4.1. Loading

The original mesh must be loaded in the program. The function `Mesh::load` is responsible for that. It reads a file containing all relevant informations :

- In 2D : surfaces, edges, vertices $\Rightarrow O(E + S + V)$.
- In 3D : everything $\Rightarrow O(E + S + T + V)$.

Then the apices of the problem have to be found in both cases with the same simple algorithm $\Rightarrow O(E)$.

Finally, the correct orientation must be found for the edges or the surfaces :

- In 2D : $O(S \cdot E)$.
- In 3D : $O(T \cdot S)$.

All together, the loading process is simplified to :

- In 2D : $\mathbf{O(S \cdot E + V)}$.
- In 3D : $\mathbf{O(T \cdot S + V + E)}$.

4.2. Initialisation

The edges (respectively surfaces) have to be passed one at a time to the tree. The same for the vertices. All together, the initialisation process then takes :

- In 2D : $\mathbf{O(E + V)}$.
- In 3D : $\mathbf{O(S + V)}$.

4.3. Refinement

The auxiliary mesh is constructed from top to bottom. There are two cases to distinguish : first, the unstructured mesh $\mathcal{T}_h$ is uniformly refined, so that the vertices can be assumed to be well distributed in the space. This results in a well balanced tree of deepness $log_4 V$, respectively $log_8 V$. On the contrary, when $\mathcal{T}_h$ is locally refined, almost all vertices are distributed in a very narrow region, while the rest of the space is quite void. It isn't possible to balance the tree, which will have a linear deepness V. The *top node* is set to be at deepness $d = 1$.

The boundary of the problem is where the algorithm has to work the most; the rest is either ignored or treated very quickly. It seems also safe to assume that 3 children out of 4 (7 out of 8) are still in an NDEF region, while the rest has as many chances to be IN than to be OUT. This, because it is the maximum number of children an edge (a surface) can cross (worst case scenario).

4.3.1. Uniform refinement

The tree is here well balanced, having a logarithmic deepness. The two and three dimensional problems are treated separately as usual.

In 2D The rules for a node N_i at deepness $d = i$ are the following :

- It contains $V_i = \frac{V}{4^{i-1}}$ vertices; dispatching them to each child takes $O(\frac{V}{4^{i-1}})$.
- From above assumption, it contains about $E_i = \frac{E}{3^{i-1}}$ edges. Clipping (equation 3.6) is done in constant time; dispatching the edges then takes $O(\frac{E}{3^{i-1}})$.
- Determining the status of any child that is crossed by no edges first requires the edges of the parent to be sorted as shown in figure 3.5. The rest is done in constant time, therefore a cost of $O(E_i)$.
- Children with status OUT will be ignored. Those with status IN (which happens about half of the time) must only be refined until each vertex stands alone in its own node. This is done in $O(V_i)$.

and the sum of all costs is $O(\frac{3}{2} \cdot V_i + 2E_i)$ for one node. The tree goes down to $d = log_4(V)$ which leads to a total of

$$\begin{aligned} O\left(\frac{3}{2}\left(\sum_{i=1}^{d}\left(\frac{3}{4}\right)^{i-1}\right)V + 2dE\right) &\leq O\left(\frac{9}{2}\cdot V + 2dE\right) \\ &= \mathbf{O(V + Elog_4 V)} \end{aligned}$$

In 3D A node N_i at deepness $d = i$ has following rules :

- It contains $V_i = \frac{V}{8^{i-1}}$ vertices, thus dispatching them to each child takes $O(\frac{V}{8^{i-1}})$.

- From above assumption, it contains about $S_i = \frac{S}{7^{i-1}}$ boundary surfaces. Clipping is done in constant time; dispatching the surfaces to each child then takes $O(\frac{S}{7^{i-1}})$.
- Finding the status of any child that has no crossing surfaces needs for each child to first find the closest edge. This is done in $O(S_i)$, and the rest needs constant time. This part has then again a cost of $O(\frac{S}{7^{i-1}})$.
- Children with status OUT will be ignored. Those with status IN must only dispatch their vertices in as many children, leading to $O(\frac{V}{.8^{i-1}})$.

and the sum of all costs is $O(\frac{3}{2} \cdot V_i + 2S_i)$ for one node. The tree goes down to $d = log_8(V)$ which leads to a total of

$$\begin{aligned} O\left(\frac{3}{2}\left(\sum_{i=1}^{d}\left(\frac{7}{8}\right)^{i-1}\right)V + 2dS\right) &\leq O\left(\frac{21}{2}\cdot V + 2dS\right) \\ &= \mathbf{O(V + S\log_8 V)} \end{aligned}$$

4.3.2. Local refinement

The tree is assumed to be in the worst case, that is where each time only one child containing all remaining vertices but one. The deepness of the tree is then $d = V$. The rules are slightly modified, and depend also whether the refinement takes place on the boundary or inside the mesh. Only the former, worst case, is analysed here.

In 2D A node N_i at deepness $d = i$ is subject to :

- It contains $V_i = V - i$ vertices; dispatching them to each child takes $O(V - i)$.
- It contains about $E_i = \frac{E}{3^{i-1}}$ edges. Clipping (equation 3.6) is done in constant time; dispatching the edges then takes $O(\frac{E}{3^{i-1}})$.
- Determining the status of any child that is crossed by no edges first requires the edges of the parent to be sorted as shown in figure 3.5. The rest is done in constant time, therefore a cost of $O(E_i)$.
- Children with status OUT will be ignored. Those with status IN (which happens about half of the time) must only be refined until each vertex stands alone in its own node. Since the worst case is analysed here, it is assumed that any child with that status is the one who already contains only one vertex. The cost is then $O(1)$, which is ignored.

and the sum of all costs is $O(V_i + 2E_i)$ for one node. The tree goes down to $d = V$ which leads to a total of

$$O\left(\sum_{i=1}^{d}(V - i) + 2dE\right) = \mathbf{O(V^2 + V \cdot E)}.$$

In 3D A node N_i at deepness $d = i$ has following rules :

- It contains $V_i = V - i$ vertices, thus dispatching them to each child takes $O(V - i)$.
- From above assumption, it contains about $S_i = \frac{S}{7^{i-1}}$ boundary surfaces. Clipping is done in constant time; dispatching the surfaces to each child then takes $O(\frac{S}{7^{i-1}})$.
- Finding the status of any child that has no crossing surfaces needs for each child to first find the closest edge. This is done in $O(S_i)$, and the rest needs constant time. This part has then again a cost of $O(\frac{S}{7^{i-1}})$.
- Children with status OUT will be ignored. Those with status IN must only dispatch their vertices. As in the 2D case, this is done in constant time since worst case is assumed.

and the sum of all costs is $O(V_i + 2S_i)$ for one node. The tree goes down to $d = V$ which leads to a total of

$$O\left(\sum_{i=1}^{d}(V - i) + 2dS\right) = \mathbf{O(V^2 + V \cdot S)}.$$

Note that a supplementary dimension does not implies a dramatic increase in costs. The only difference lies in the type of boundary, edges for 2D, surfaces for 3D.

4.4. Generation

Once the tree reaches a leaf, the auxiliary mesh is completed when needed. Leaves with status OUT are ignored. Those with status NDEF are ignored in three dimensions, but might still be considered in two dimensions : one of the two triangles might be included in the final auxiliary mesh $\mathcal{T}_a$. This is done in $O(\frac{E}{3^{d-1}})$. The number of resulting triangles is insignificant in comparison of all those generated every time an IN leaf is found.

The generation at this step is done by just inserting the apices of a shape in a vector. In two dimensions, there are two triangles (ignoring NDEF leaves) forming a rectangle, thus 6 vertices; in three dimensions, there is one cuboid, or 8 vertices. The corners required to build such shapes must be stored in a map : they can't exist twice if they are shared by two (or more) nodes. Insertion in a map is done in logarithmic time, depending on how many vertices were inserted before.

Thus, the time needed to prepare the mesh is :

- In 2D : There are about 3^{d-1} NDEF leaves and $\frac{V}{2}$ IN leaves. The cost is then $O(E + \frac{V}{2} log_2(V'))$.
- In 3D : There are about $\frac{V}{2}$ IN leaves. The cost is then $O(\frac{V}{2} \cdot log_2(8V)) = O(V \cdot log_2 V)$.

But the auxiliary mesh $\mathcal{T}_a$ is present only in memory at that time; it must still be written in the output file. Vertices are stored in a map, and the result must be a vector

: they will be sorted relative to their position in space, first x, then y and finally z if it exists. This sorting has already been made while inserting new vertices; what is still unknown is the index to each generated vertex V_i'. This is found in $O(V')$.

When the indices are known, the shapes K' defining the auxiliary mesh $\mathcal{T}_a$ can be written in $O(K')$. The total of K' is equal to (twice) the number of IN leaves. That is, this step is done in $O(V)$.

The vertices themselves are then written, in $O(V')$ again. Since V' is proportional to K', it results that $V' \propto V$.

To summarise, the costs to generate the auxiliary mesh $\mathcal{T}_a$ are :

- In 2D : $\mathbf{O(E + V \cdot \log_2(V))}$
- In 3D : $\mathbf{O(V \cdot \log_2(V))}$

4.5. Computing constants

To compute the constants described in the previous chapter, the program requires some time, trying to minimise it as much as possible. They are computed simultaneously where possible.

In 2D

- ρ & C_a : $S \cdot log_4 V$
- h_p : $V \cdot S$
- C_∂ : $E + log_4 V + E \cdot S$

which sums up to $\mathbf{O(S \cdot \log_4 V + (V + E) \cdot S)}$

In 3D

- ρ & C_a : $T \cdot log_8 V$
- h_p : $V \cdot T$
- C_∂ : $S + log_8 V + S \cdot V$

which sums up to $\mathbf{O(T \cdot \log_8 V + (T + S) \cdot V)}$

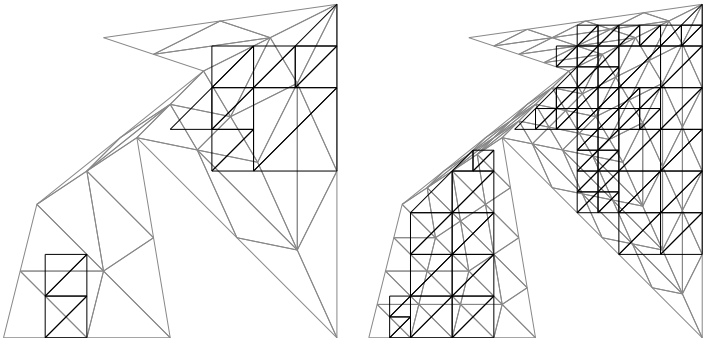
5. Examples

Some examples can be discussed now that the algorithm has been analysed. Images will show original mesh in light grey, and auxiliary mesh in black. The constants will be all put together at the end of this chapter.

5.1. Example 1

5.1.1. Uniformly refined

This mesh has been created in the only goal to test as many situations as possible. The original mesh is refined uniformly so that a level number higher by one consists of four times as many triangles. Level 0 is of no interest, since the auxiliary mesh is void. Level 1 contains already some nodes and shown on the left in figure 5.1. On the right is level 2. In figure 5.1 a), there are large regions where the unstructured mesh $\mathcal{T}_h$ is



a) 1 level of refinement b) 2 levels of refinement

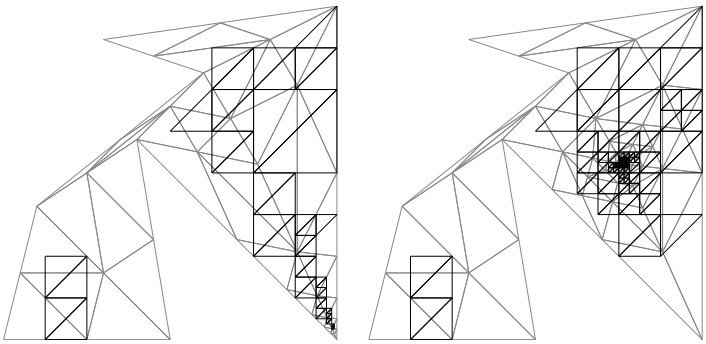
Figure 5.1.: On the left, some quite large regions are "missing" (no black rectangles are shown); that is corrected on the right

not covered at all by $\mathcal{T}_a$. This happens when a single vertex is present in that region, thus the corresponding node is a *leaf* with status NDEF. It also happens sometimes that a roundoff error occurs; for example 10 becomes 9.9999, and the edge that should lie vertically at $y = 10$ is not vertical any more. Thus, the boundary comes inside the node and an artificial intersection is found by the algorithm.

5.1.2. Locally refined

Now only one point is refined. Once it is done in the boundary, once in the middle of the mesh. Example 1 with uniform refinement level 1 is taken as basis. Figure 5.2 shows

the results of the local refinement.



a) On the boundary b) In the middle

Figure 5.2.: The mesh of example 1, refined locally

5.2. Example 2

Another two dimensional example, given as tutorial by Netgen, is shown in figure 5.3. Once again, strange patterns can be seen along the left edge because of the form of the

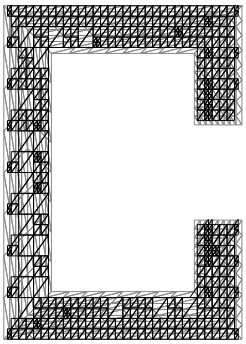


Figure 5.3.: The second example

triangles in the original mesh and roundoff errors. Here the third level of refinement is shown, since previous ones have holes in the auxiliary mesh.

5.3. 2D chip

The chip shown as an example for multiple meshes on figure 3.9 is presented now. The different parts work independently, and are assembled at the final stage of the construction.

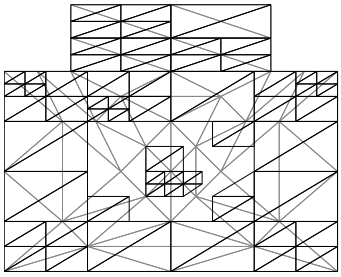


Figure 5.4.: The chip

5.4. 3D ellipsoid

A three dimensional example, an ellipsoid. Here no refinement had to be use, even the raw mesh generated by Netgen was nicely handled by the program.

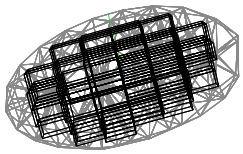


Figure 5.5.: The ellipsoid

5.5. 3D "fichera"

Another three dimensional example, given in the Netgen example with name *fichera*. It looks like an assemblage of seven small cubes that form a bigger cube from which one corner is missing. The *fichera* is composed only by cuboid forms. Since no roundoff error appeared, the auxiliary mesh $\mathcal{T}_a$ covers exactly the same space than the unstructured mesh $\mathcal{T}_h$, resulting in a perfect $C_\partial = 0$ value ! C_a on the contrary is not equal to 1, because some nodes have been split one time more than it were in the original mesh.

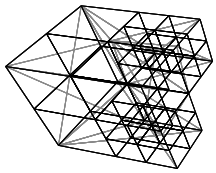


Figure 5.6.: The *fichera*

5.6. Numerical results

For the examples discussed above, plus a few other ones with an explicit name, the numerical results are shown in the following tables. The columns are : a name, the number of vertices, the number of shapes on $\partial\mathcal{T}_h$, the number of shapes forming $\mathcal{T}_h$, ρ, C_a, C_∂, time to initialise the problem, to build the tree, an to write the auxiliary mesh in a file. All times are given in seconds.

The results show very interesting values. The constant C_a is an indicator that shows how similar both meshes $\mathcal{T}_h$ and $\mathcal{T}_a$ are. Its value is always chosen to be greater or equal to one, the closer the better. Even with an increasing refinement, its value remains stable, increasing just a little. The same applies for C_∂ when the original mesh is not too distorted. In the first example though, its value doubles at each further refinement. This probably comes from the great distortion of the original mesh.

The times are also of some interest. Loading of the mesh is the most time-consuming part of the program. First, it must read the mesh in an external file. More than that, it must correct the orientation of the edges, which requires many long loops. The creation of the tree itself doesn't takes much time, as the final writing of the mesh in a file. This tends to confirm the complexity analysis.

Not shown are times to compute the constants. They are the real bottleneck of the program : in the example of the ellipsoid, uniformly refined three times, it requires about 125 seconds. In comparison with the 17.5 seconds required to load the mesh, it once again seems to confirm the analysis of the problem.

Type of mesh	V	E	S	ρ	C_a	C_∂	t_1	t_2	t_3
Example 1, lvl 0	9	8	8	41.099	2.82843	0.67049	0	0	0
Example 1, lvl 1	25	16	32	41.099	2.82843	2.16388	0.01	0	0
Example 1, lvl 2	81	32	128	41.099	2.82843	4.32777	0.01	0.01	0
Example 1, lvl 3	289	64	512	41.099	3.62215	8.65554	0.05	0.01	0.02
Example 1, lvl 4	1089	128	2048	41.099	7.24431	17.3111	0.19	0.03	0.09
Example 1, lvl 5	4225	256	8192	41.099	7.24431	34.6221	0.76	0.14	0.11
Example 1, lvl 6	16641	512	32768	41.099	7.24431	69.2443	1.33	0.18	0.45
Example 1.2, a)	37	24	48	41.099	5.65685	5.53588	0	0	0
Example 1.2, b)	71	18	122	41.099	4.1818	3.45004	0	0	0
Example 2, lvl 0	12	12	10	15.203	6.50987	0.41602	0	0	0
Example 2, lvl 1	33	24	40	15.203	6.50987	0.97072	0.01	0	0
Example 2, lvl 2	105	48	160	15.203	6.50987	0.69337	0.02	0	0.01
Example 2, lvl 3	369	96	640	15.203	6.50987	0.81343	0.06	0.02	0.03
Example 2, lvl 4	1377	192	2560	15.203	6.50987	0.97072	0.24	0.05	0.13
Chip, lvl 1	37	26	56	13.627	4.4376	2.52982	0	0	0

Table 5.1.: Numerical results for two dimensional problems

Type of mesh	V	S	T	ρ	C_a	C_∂	t_1	t_2	t_3
Ellipsoid, lvl 0	100	170	2^8	11.836	3.86767	1.92973	0.04	0.01	0
Ellipsoid, lvl 1	540	680	2^{11}	18.283	7.39261	2.89276	0.33	0.07	0.02
Ellipsoid, lvl 2	3467	2720	2^{14}	19.174	7.66315	3.03051	2.36	0.17	0.17
Ellipsoid, lvl 3	24677	10880	2^{17}	19.730	7.61601	3.39847	17.5	1.21	1.54
Cone, lvl 0	237	444	575	16.533	5.87157	1.98446	0.14	0.04	0.02
Cone, lvl 2	1270	1776	4600	25.542	7.34567	3.48741	1.10	0.06	0.05
Ferrit, lvl 0	512	366	2755	12.565	5.90989	1.67208	0.26	0.08	0.04
Fichera	17	30	25	9.631	4.55895	0	0.01	0	0

Table 5.2.: Numerical results for three dimensional problems.

6. Conclusion and outlook

The program works well under normal conditions. The generated mesh has the properties required for the preconditioners presented in the introduction, it covers as much space as possible with properties similar than the unstructured mesh (constants C_a and C_∂).

There are yet some improvements that could be made in further versions : first, some 3D meshes $\mathcal{T}_a$ are partially wrong because the *nearest surface* search chose the incorrect boundary surface. This happens but only in particular cases where the boundary is not convex, and folded around itself, like in figure 6.1. Changing the way the nearest surface is found would solve that problem.

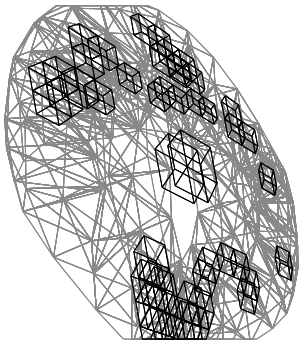


Figure 6.1.: A sculpture in St.-Gallen; in the centre there are two wrong nodes.

The input file has also some limitations. When using only triangles and tetrahedrons to create an unstructured mesh $\mathcal{T}_h$ there is no problem. Normally, Netgen generates such files. But when dealing for example with an *hp*-refined mesh, cuboids and other shapes are created. The auxiliary mesh seems to be correctly created, but the constants that are computed can't be trusted any more; they show only an approximation of what their real value is. Therefore no such example has been listed in table 5.2.

The local refinement has been implemented only as a small tool aside of the main problem. It has been created at the early stage of development of the program, and only for two dimensional meshes. The locally refined unstructured mesh $\mathcal{T}_h'$ that is created is stored in the file *myGrid.vol*. It can be used to compare both meshes in a viewer, but should not be used for anything else. In particular, multiple meshes as in example 5.3 might not be correctly refined that way.

Bibliography

[1] T. F. Chan and J. Zou, *A convergence theory of multilevel additive schwarz methods on unstructured meshes*, Numerical Algorithms, 13 (1996), pp. 365–398.

[2] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hugues, *Computer Graphics, Principles and Practice (2nd Edition)*, Addison-Wesley, 1995.

[3] R. Hiptmair, G. Widmer, and J. Zou, *Auxiliary space preconditioning in* $h_0(curl, \omega)$, Numer. Math., (2005). Submitted. Published as Technical Report CUHK-2005-06 (327), Department of Mathematics, The Chinese University of Hong Kong.

[4] Y.-D. Liang and B. A. Barsky, *A new concept and method for line clipping*, ACM Transactions on Graphics 3, 1 (1984), pp. 1–22.

[5] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics (2nd Edition)*, McGraw-Hill International Editions, 1979.

[6] J. Schöberl, *Netgen - an advancing front 2d/3d-mesh generator based on abstract rules*, Comput. Visual. Sci., 1 (1997), pp. 41–52.

[7] A. Stepanov and M. Lee, *The Standard Template Library.* HP Technical Report HPL-94-34, February 1995.

[8] J. Xu, *The auxiliary space method and optimal multigrid preconditioning techniques for unstructured grids*, Computing, 56 (1996), pp. 215–235.

A. File formats

A.1. *.vol

This very simple, almost self-describing, format is the default format used by Netgen (Use *File : Save Mesh*). It represents the unstructured mesh $\mathcal{T}_h$. The file is split in different parts. Each part begins with a line describing what kind of elements will follow. The second line determines how long the list is, then each element is written, one at a line. Comments can be written on a line beginning with a ♯ and are ignored. The first three lines of the file are a header giving the type of the file and the dimension of the problem. The parts that can be expected by the program discussed here are :

- *volumeelements* : In 3D, they are the shapes K constituting the unstructured mesh $\mathcal{T}_h$. In 2D, they just don't exist. A mesh number, useful when multiple meshes are involved, is given. The number of corners of the shape, normally 4, comes next. Finally, an index to each of these corners is written.
- *surfaceelementsgi* : In 3D, this is the boundary of the mesh; in 2D, these are the shapes K forming $\mathcal{T}_h$. A mesh number is given first, useful when dealing with multiple meshes. Three parameters are ignored. Then the number of corners of each shape is written (normally 3), directly followed by the indices to these vertices.
- *edgesegmentsgi2* : The edges, boundary of the 2D problem. The first two parameters are ignored, leaving the start and end points of the segment as indices.
- *points* : The full list of all vertices relevant for the mesh $\mathcal{T}_h$. The x, y and z coordinates are all what is given there.

A.2. *.am

This format is very similar to the previous *.vol* format, and represents the auxiliary mesh $\mathcal{T}_a$ generated by the program. The header summarises the type of mesh stored and its dimension.. The different parts are then subdivided in an analogue way as above, with the following sections :

- *surface* : The triangles forming the 2D mesh $\mathcal{T}_a$ (not present in 3D). The number of apices is given, set to 3. Then the indices to the corresponding vertices are written.
- *cuboid* : The cuboids forming the 3D mesh $\mathcal{T}_a$ (not present in 2D). The number of apices is given, set to 8. Then the indices to the corresponding vertices are written.

- *points* : The list of all vertices relevant for the mesh $\mathcal{T}_a$. The x, y and z coordinates are written there.

A.3. *.ref

The files with that extension contain information where to refine the mesh locally. The format is not very different than the previous ones, and each part is given with a label, a number of elements, and the elements themselves.

- *points* : The indices to vertices to locally refine.

B. Programme classes

B.1. Class Vertex

This class represents vertices, or apices. Its members are :

- Member datas
  - `bool boundary` : This boolean is set to *true* if the vertex is on the boundary, else it is left to *false*.
  - `vector<double> position` : The coordinates of the problem : `[0]` represents x, `[1]` y and `[2]` z.
- Constructors
  - `Vertex(int d)` : Initialises the vertex with dimension d (default 3); `boundary` is set to *false*.
  - `Vertex(const Vertex v)` : Copy constructor, to initialise a vertex with the same values as another.
- Operators
  - `=` : Affectation operator.
  - `!=` : Comparison operator.
- Member functions
  - `at(int i)` : An equivalent to `getX()` and `setX()` using indices.
  - `double getX()` : Returns the value along the x axis (analogue for y and z).
  - `double hash()` : Returns the hash value of the vertex to sort it in a map.
  - `bool isBoundary()` : Returns true if the vertex is on the boundary.
  - `setBoundary(bool b)` : Sets `boundary` to the value of `b` (*true* by default).
  - `setX(double x)` : Sets the value of the x axis (analogue for y and z)

B.2. Class Polygon

This class represents any polygon (in 2D : line, triangle) or polyhedron (in 3D : tetrahedrons) whom vertices are stored in a separate `vector`. The class doesn't know anything about the vertices except for their indices.

- Member datas
  - `bool boundary` : This boolean is set to *true* if the polygon is on the boundary, else it is left to *false*.
  - `int size` : The number of points defining the polygon / polyhedron.
  - `vector<unsigned int> vertices` : The index to the vertices (1..n).
- Constructors
  - `Polygon(int s)` : Initialises the polygon with size s; `boundary` is set to *false*.
  - `Polygon(const Polygon v` : Copy constructor, to initialise a polygon with the same values as another.
- Operators
  - `=` : Affectation operator.
- Member functions
  - `at(int i)` : Access to the index of the i-th vertex.
  - `bool isBoundary()` : Returns *true* if the polygon is on the boundary.
  - `setBoundary(bool b)` : Sets `boundary` to the value of `b` (*true* by default).
  - `getSize()` : Returns the number of apices of the polygon.

B.3. Class Mesh

This class is used to store the unstructured mesh $\mathcal{T}_h$ and perform some operations on it. For example, it is inside that class that the mesh can be locally refined or its constants ρ, C_a and C_∂ are computed. It is also from that class that the Quadtree (Octree) is initialised.

- Member datas
  - `int dimension` : The size of the problem (2D or 3D).
  - `int verbose` : A parameter to set how much information should be displayed.
    - 0 : The only information displayed is to tell the auxiliary mesh has been created.
    - 1 : Some useful informations about $\mathcal{T}_h$ are printed.
    - 2 : The same, plus some timing informations.
    - 3 : Debugging informations.
  - `list<Polygon> edges` : A list of edges on the boundary.
  - `list<Polygon> surfaces` : A list of surfaces (only boundaries in 3D).
  - `vector<Vertex> vertices` : The list of all vertices used by the mesh.

  - `vector<unsigned int> local_points` : A list of points to refine the node locally.
  - `extremes` : Minimal and maximal values of the vertices along each axis.

- Constructors
  - `Mesh()` : Default constructor.

- Member functions
  - `compute_C()` : Computes the constants ρ, C_a, C_∂.
  - `display2D()` : A function to display the polygon using OpenGL (exists also for 3D).
  - `init(Quadnode *)` : Initialise the quadtree (exists also for Octree).
  - `load(string file)` : Reads the specified file and load the mesh $\mathcal{T}_h$ it contains.
  - `find_boundary_vertices()` : Mark all vertices that belong to a boundary edge as boundary themselves.
  - `finish_edges()` : Find the correct orientation for the boundary.
  - `int getDimension()` : Returns the dimension of the mesh.
  - `refine(string file)` : Refine the mesh locally following instructions written in the specified file.
  - `setVerbose(int v)` : Set the verbose level to v.
  - `write(string file)` : Write the unstructured mesh in the specified file.

C. Command line arguments

The program, called *amr*, is called simply by running the command `./amr -in input_file.vol`. The output file storing the auxiliary mesh is then `myGrid.am`, in the current directory. Some options are available, listed here in tables separating the ones that require an additional argument and the ones that don't. The whole synopsis is :

```
./amr -in filename.vol [-out filename.am] [-ref filename.ref] [--C]
      [--xml|-xml filename.xml] [--C] [-verbose [0|1|2|3]] [--help]
```

Table C.1.: Command line arguments with 1 parameter.

`--C`	Prints informations about the constants C_a and C_∂.
`--xml`	Write the xml informations into the default file, *myGrid.xml*.
`--help`	Prints the list of all parameters.

Table C.2.: Command line arguments with 2 parameters.

`-in` *input_file*	The input file, of the format *.vol*.
`-out` *output_file*	The output file, of the format *.am* (default is myGrid.am).
`-xml` *xml_file*	An optional output file, of the format *.xml* describing all nodes of the quadtree.
`-ref` *refinement_file*	A file, of the format *.ref*, which describes a local refinement to apply. The locally refined unstructured mesh so created is stored in the file *myGrid.vol*, but should not be used except to compare the meshes in the viewer.
`-verbose` *level*	Prints some additional messages, like timestamps (level 2), warning messages (level 3).

There is a viewer aside of the program, compiled with the command `make viewer`. Its synopsis is

```
./viewer -in filename.vol -out filename.am
```