Chapter 2

Basic structures and algorithms

Introduction

The aim of this chapter is to introduce some basic data structures and to show how they can be used in a mesh generation context. To this end, some basic as well as more sophisticated data structures are recalled together with some algorithms of greater or lesser complexity. The discussion is then developed by means of various application examples related to some algorithms or situations that are extensively used in a meshing context.

While people having a computer science background may be familiar with these basic notions (at least from a theoretical point of view), we would nevertheless like to address this topic here in order to allow people less directly concerned with algorithmic problems to gain some knowledge of this (vast) topic (notably numericians or people having a finite element background, for instance). Moreover, in the mesh generation context, specific applications and uses of the classical data structures lead to specific situations and hence merit some comments.

The literature about data structures and algorithms is quite abundant. Among the usual references, the classics would include [Aho *et al.* 1983], [Wirth-1986], [Sedgewick-1988], [Cormen *et al.* 1990], [Samet-1990], [Gonnet *et al.* 1991] and, even more recently, [Knuth-1998a], not to mention many others that can also be consulted.

The complexity, both in terms of the number of operations and of the memory resource allocated, is analyzed from a theoretical point of view. However, specific theoretical results obtained in *ad-hoc* academic situations must be slightly nuanced when dealing with more concrete situations.

Indeed, numerous assumptions like "the points must be in general position" in a triangulation problem or all operations involved in a given numerical process have the "same cost" or again are "exact" are clearly unlikely to be realistic in "real life" (the world of numerical computations). Nevertheless, despite these remarks, theoretical results allow for a good understanding of some difficulties and help us to find appropriate solutions.

Therefore, after having described the theoretical point of view in the first sections of this chapter, we give some indications and remarks about frequently encountered difficulties in realistic applications. After that, we turn to some application examples to illustrate how to benefit from the theoretical material in simple problems related to common meshing purposes.

The first section introduces the general problem using an academic example. The second section presents the most commonly used elementary data structures (array, list, stack, etc.). The third section deals with complexity problems for a given algorithm. Section four analyzes the sorting and searching techniques and introduces three main paradigms used in many methods. Data structures in one and two dimensions are discussed in sections five and six, while topological data structures are mentioned in section seven. Sections eight and nine deal respectively with the notions of robustness and optimality of an implementation. The last section proposes several practical application examples commonly found in mesh generation.

2.1 Why use data structures ?

As an introduction, we look at a "naive" algorithm that can be used to construct a triangulation. Let consider a set of points $\mathcal{S}$, all contained (to simplify even more) in a single initial triangle. The algorithm consists in finding, for each and any point P, the triangle K enclosing it and then to subdivide K into three new triangles by connecting P to the three vertices of K :

- For $P \in \mathcal{S}$
  - Find triangle K containing the point P,
  - Subdivide this triangle into three.
- End

While very simple, this algorithm raises several questions. Among these, we simply mention the need to define the concept of a triangulation and how to represent it, for instance by using adjacency relationships between the triangles. Another question is related to the quick identification of the triangle containing the point P to be inserted. Should we examine all triangles of the current triangulation or take into account the fact that any triangle is obtained by subdividing its parent ?

This simple example gives some indications on how to proceed and what to know to implement such an algorithm (simple as it may be). This is not indeed restricted to defining the operations required to code this algorithm, but also to finding the data structure(s) adapted to the problem, in such a way as to define a

Program. According to [Wirth-1986], we have the following scheme :

$$\textit{Algorithm} \quad + \quad \textit{Data Structures} \quad = \quad \textit{Program}.$$

There is obviously a link between an algorithm and the data structures it uses. Usually, the more complex a data structure, the simpler the algorithm will be, although the simplicity of the algorithm is generally altered during the data structure update. For triangulation (meshing) algorithms in particular, a rich data structure allows useful data to be stored and retrieved thus simplifying the task of the algorithm, but on the other hand, any modification of the mesh induces a set of modifications and updates of the data structure. On the other hand, a simple data structure is efficient to update but forces the algorithm to perform explicitly a set of operations to recreate some data it needs. As an example, a data structure that keeps only the adjacency relationships between the triangles of a mesh provides instantaneously the neighbors of a given triangle but requires a computation to identify all the triangles sharing a common vertex. However, if this information is stored, it can be retrieved immediately, provided it has been updated as the mesh evolves.

Notice that a (mesh) data structure contains integer values (numbers, indices, etc.) or real values (triplets of coordinates of the vertices in three dimensions, for instance). In Section 2.2, we will describe data structures allowing this kind of information to be stored.

A data structure being fixed, we discuss the behavior of the algorithm. Therefore, we recall, in Section 2.3, several fundamental notions about the complexity, allowing to analyze the efficiency of standard algorithms and basic data structures. In Section 2.4, we describe a series of algorithms, based on one of the computer science paradigms, namely, *Divide and Conquer*. We give some examples of methods for searching and sorting, some of which we describe, such as the insertion sorting technique, the quicksort and bucket sorting as well as binary searching (dichotomy) or interpolation methods. Section 2.5 discusses the manipulation of entities of dimension one (integers). To this end, we look at :

- general data structures allowing to store, retrieve or analyze sets of objects,
- structures allowing a selective access to some entities already stored. The access can be performed according to several criteria of selection. We find here, for instance, the approaches where the smallest item (in some sense), the first, the last recorded, the neighbor of a given item, etc. is sought. Here we will find the data structures like stack (LIFO), queue (FIFO), priority queues, array with sorting and binary searching trees.
- data structures like dictionaries that can provide answers to questions like "does this item exist ?" and allow items to be inserted or suppressed. We will find here BST and hash coding techniques.

In Section 2.6, we discuss how to use data structures in two and three dimensions for fast storing and retrieving of items such as points, segments (edges) or polygons. Section 2.7 is devoted to the computer implementation of topological data.

After this overview of basic data structures and algorithms, we discuss robustness problems inherent to any implementation of a mathematical expression in a computer. The degree of the problems and the notion of predicate are then analyzed as well as the cost in terms of the number of operations and of memory requirements (Sections 2.8 and 2.9).

To conclude, we mention some applications where the previously described material can be used, in the specific context of the development of mesh generation and modification algorithms (Section 2.10).

2.2 Elementary structures

In this section, we describe tables (arrays), pointers, lists, stacks and queues. These structures are briefly introduced below using some simple examples.

2.2.1 Table or array

The table or the array is most certainly the simplest and the most efficient data structure for numerous applications. An array can be simply defined as a fixed set (connected or contiguous) of memory where items of the same nature (more precisely, items having the same storage requirement) are sequentially stored and are accessible by one or several indices. The important point is that an array allows *direct access* to each and any of its elements. Indeed, if the array begins at the address a and if each item requires b words of memory to be stored, then the item indiced by i starts at the address $a + (i-1)\,b$. This simple property means that the array is a convenient data structure, easy to use and hence, is used as a basic component in more sophisticated structures (trees, hash tables, grids, etc.) described hereafter.

5	3	?	17	?	1
1	2	3	4	5	6

Figure 2.1: *An array of length 6.*

The intrinsic drawback of the array structure (besides the need to detect a possible overflow) is related to the static memory allocation it requires, before being used. In other words, if more space is needed at some point, a new array must be allocated and the old one should be copied into this new one.

The Figure 2.1 shows an example of an array of length 6 containing integer values. Items 3 and 5 are not yet affected and thus, the corresponding values are undefined (symbolized by the ? sign).

An array allows to store vectors (a one-index array, in the usual sense), matrices (array of two indices), etc. and, as mentioned, the arrays are used as basic components in more elaborate data structures.

2.2.2 List

The list is another data structure is which the items are stored in a linear way (sequentially). Unlike an array in which the items follow each other in a portion of the memory, the nodes of a list can be accessed using an address or, more precisely, a pointer. The notion of a pointer is naturally available in most programming languages and, if not (for instance in Fortran 77), can be emulated as will be shown in Section 2.5.

To clarify, we now describe the case of a double linked list. In this case, each node contains three fields : the *key* which represents the expected information and two *pointers* that allow access to the neighboring nodes. To handle a list correctly, we need two additional pieces of information : the *head* and the *tail* of the list (Figure 2.2). The head and the tail give access to the first and the last node of the list.

To check whether a key is contained in a list, it is sufficient to cover the list starting from the head, following the pointers until the tail is reached (then the key does not exist in the list, except if it is in the last node).

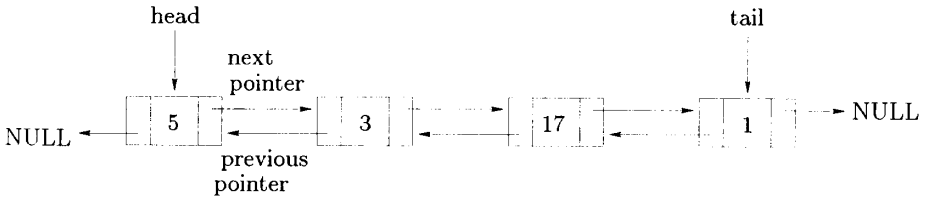

Figure 2.2: *A double linked list (containing the same keys as the array of Figure 2.1).*

Adding or deleting an item in a list is equivalent to adding a node and updating the relevant pointers or breaking existing pointers, while managing head and tail pointers, if necessary. The following schemes illustrate the operations of searching for and inserting an item in a list, that is (with obvious notations) :

Algorithm 2.1 *Searching for an item x in a list.*

```
Procedure ListContains(List,x)
node ← head(List)
WHILE node ≠ NULL AND node.key ≠ x
   node ← node.next
END WHILE
RETURN node (if node ≠ NULL, then x is present in the list)
```

Algorithm 2.2 *Insertion of an item x at the beginning of a list (newnode being the new element).*

Procedure `ListInsert(List,x)`
node ← *newnode*
node.key ← *x*
node.prev ← *NULL* and *node.next* ← *head(List)*
head(List).prev ← *node*
head(List) ← *node*

Algorithm 2.3 *Insertion of an item x in a list after a given position current.*

Procedure `ListAppend(List,current,x)`
node ← *newnode*
node.key ← *x*
node.next ← *current.next* and *node.prev* ← *current*
current.next.prev ← *node* and, finally, *current.next* ← *node*

Several types of lists exist (other than the doubled linked list). Indeed, one can find the simply linked lists (one of the two pointers is missing), the circular lists (items are accessible via a circular order), the sorted lists, etc. Irrespective of the case, a list is a simple and flexible way of managing dynamic sets of entities (the number of entities varies throughout the process), although in some cases (Section 2.3) the searching operations can be expensive. Hence, notice that lists are well adapted to cases where the considered values do not follow any specific order. Notice also that the size of the list must be known in advance (if it is implemented as an array), otherwise dynamic allocations are to be expected as it evolves.

Exercise 2.1 *Explain how to implement a circular list. How many pointer(s) is (are) required ?*

Exercise 2.2 *Examine a data structure based on a linked list but in which the entities point to an array. What advantages can be expected from such an organization ?*

2.2.3 Stack

As for a plate stack, where only the plate on top of the stack can be accessed, the stack is a data structure allowing access only to the last item inserted. For this reason, it is referred to as a LIFO list (Last In First Out). The usual operations associated with a stack are twofold : *Push* add an item to the top of the stack and *Pop* remove the item on top of the stack (if one exists, *i.e.*, if the stack is not empty).

A stack is very easy to implement using simultaneously an array (the stack itself) and an integer, the *stack pointer*, that indicates the index of the last item stored in the stack, Figure 2.3.

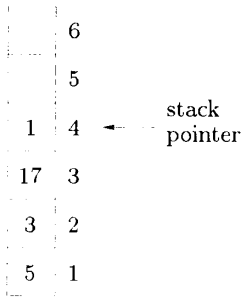

Figure 2.3: *A stack (containing the same keys as the array of Figure 2.1 or the list of Figure 2.2).*

If the stack pointer is null (void), the stack is empty. If this pointer exceeds the size of the stack, the latter overflows. In this case, a solution consists in allocating an array that is (two times) bigger and copying the old stack in this new structure. Another solution consists in using a linked list (see next exercise).

Exercise 2.3 *Can we use a simple linked list or should we use a double linked list to implement a stack ?*

2.2.4 Queue

A queue is a data structure whose behavior is very close to that of a queue of persons (as can be seen in a post office[1]). The main operation with a queue is to access the next entity, which is equivalent to removing this item from the queue. Moreover, each new item is appended at the end of the queue. According to this logic, we have a *FIFO* type structure, First In First Out. If the structure used to implement a queue (an array for instance) is of bounded size, an *overflow* is encountered whenever an item is added to an already full queue. On the other hand, attempting to remove an item from an empty queue leads to an *underflow*.

A simple linked list offers all the required facilities to implement a queue. However, if the maximum number of items to be stored is known in advance, an array is preferable as it avoids the memory allocation or deallocation problem and the possible overflow. For more details, we refer the reader to [Cormen *et al.* 1990] and to the following exercise.

Exercise 2.4 *Find a data structure to implement a queue using an array of fixed size, and address the problems of overflow and underflow (Hint : if the head (the tail) of the queue is reached and if free space is available in the array, "move" (pack down) the items of the queue).*

[1] without wishing to make any unfair assumptions about this type of institution in any particular country

2.2.5 Objects and pointers

In Section 2.2, we assumed that it is possible to associate to each entity one or two additional fields (the pointer(s)) indexing nodes of the same type. On the one hand, the notion of pointer may not exist in some languages and, on the other hand, if the number of items is known in advance, the use of pointers serves no purpose. To demonstrated this, let us look at memory allocation/deallocation problems in this context. A simple way of handling this situation is to manage a list of free memory entries, a *free list*. When more space is needed, we apply a *First-Fit* method. The list is explored and the first free available block of memory (of appropriate size) is used. If this area is bigger than necessary, it is split and the remaining block is appended to the free list. In a similar way, the addition of a free block is performed by referencing it in the free list (possibly by merging it with the neighboring memory blocks if such blocks are free).

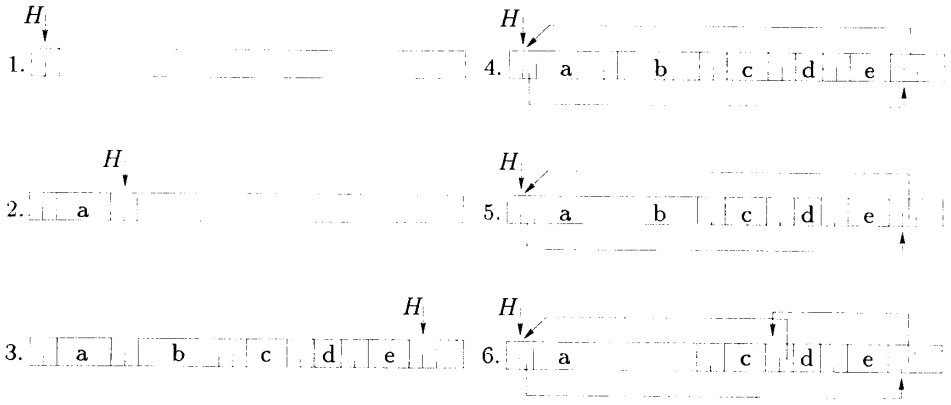

Figure 2.4: *Memory management with the First-Fit strategy.*

This point is emphasized in Figure 2.4 where H represents the pointer to the head of the free list. Notice that two integers are associated with each information unit, a pointer to another block linking the free list) and an integer giving the actual block size (used in the First-Fit operation). In the example, we have allocated a, b, c, d and e in line. If a becomes available and if b also becomes available, then the area $a \cup b$ is defined as free (available). Then, d becomes available. Hence, the free list is maintained.

Fortunately, more sophisticated memory management mechanisms exist. Nevertheless, if the size of the memory required is not known in advance or if the notion of pointer does not exist, attention must be paid to the memory management procedure. We emphasize this point in the case where a double linked list stored in an array is used. Each record contains the given information and two integers defining the next and previous records in the list. The example of Figure 2.5 illustrates this implementation for the list given in Figure 2.2. The same representation can be obtained using three arrays (one for the key, one for the

pointer to the next and one for the pointer to the previous item) instead of only one.

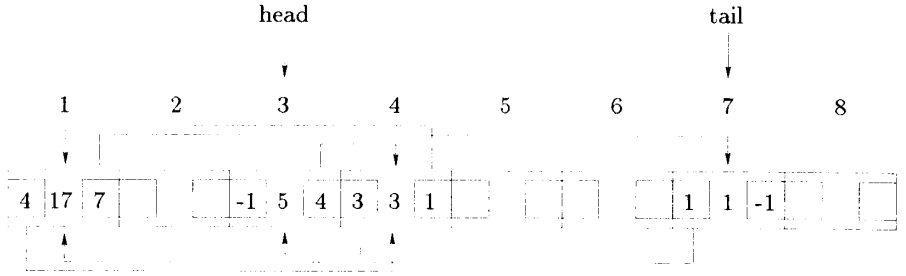

Figure 2.5: *Linked list of Figure 2.2 implemented with an array. Middle, the list, top, the pointer to the following node and bottom, the pointer to the previous node.*

2.3 Basic notions about complexity

We discuss here the problems related to the complexity of an algorithm. As will be seen, this notion concerns various aspects and affects the "quality" of the algorithm considered.

2.3.1 Behavior of a function

As seen in Section 2.1, the construction of an algorithm and/or a data structure requires evaluating the number of operations involved and, in addition, looking closely at memory problems.

Addressing these points involves analyzing the *complexity in time* as well as the *complexity in size* of the algorithm and/or the data structure. A simple and generic way to analyze these notions is to introduce a mathematical function f related to the size n of the inputs of the problem, where for instance, n is the number of points to be inserted in the triangulation. Finding the exact value of $f(n)$ can be difficult or even impossible. However, we are not usually interested in this value, but rather in a rough estimate. Several ways of quantifying this point exist. The usual notations are $f(n) \equiv \Theta(g(n))$ or $\Omega(g(n))$ or $\mathcal{O}(g(n))$ or $o(g(n))$ which indicate respectively that, for a sufficiently small n and for a "known" $g(n)$, we have :

- Θ : $c_1\, g(n) < f(n) < c_2\, g(n)$ where c_1 and c_2 are two constants. Hence, f and g have the same behavior when n grows.
- $\mathcal{O}$: $f(n) < c\, g(n)$, c being a constant, and we have an upper bound,
- Ω : $f(n) > c\, g(n)$ where c is a constant. We have then a lower bound,
- o : $f(n) < c\, g(n)$ for any positive constant c.

The quantifications in $\mathcal{O}$ and o can be seen as two ways of comparing two functions from slightly different points of view when looking at the bounds. One way of qualifying these two measures is indeed to write :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < c\,,$$

for $\mathcal{O}$, while for o, we have :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\,.$$

As an example, notice that $2\,n^2 = \mathcal{O}(n^2)$ but $2\,n^2 \neq o(n^2)$, while we have $n\,log\,n = \mathcal{O}(n^2)$ as well as $n\,log\,n = o(n^2)$. Figure 2.6 illustrates the comparisons of behavior.

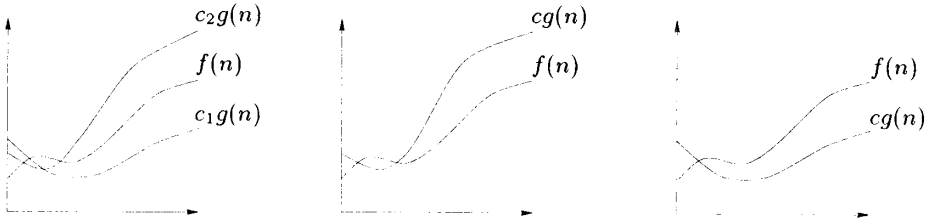

Figure 2.6: *Illustrations of the notations* Θ, $\mathcal{O}$ *et* Ω.

Notice however that the previous expressions include some constants that can be very large. For instance, an algorithm of complexity n^2 is *a priori* faster than an algorithm of complexity $100\,n$, if n is smaller than 100 in the case where the constants are equal, if not it can lead to the opposite result. Another remark is that the complexity measures the behavior for large input sizes rather than for small size problems. This will be discussed in Section 2.4 when dealing with sorting algorithms.

Exercise 2.5 *Are there some values for which an algorithm in* n^3 *is more efficient than an algorithm in* $1000\,n^2\,log\,n$ *?*

Exercise 2.6 *Let* f *and* g *be two functions assymptotically positive. Among the following assertions, indicate which are true and which are false :*

- *i)* $f(n) = \mathcal{O}(g(n))$ *with* $f(n) = \sqrt{n}$ *and* $g(n) = n^{\sin n}$,
- *ii)* $f(n) + g(n) = \Theta(min(f(n), g(n)))$,
- *iii)* $f(n) = \mathcal{O}(g(n))$ *means* $2^{f(n)} = \mathcal{O}(2^{g(n)})$,
- *iv)* $f(n) = \mathcal{O}(g(n))$ *means* $(1 + \frac{1}{logn})f(n) = \mathcal{O}(g(n))$.

2.3.2 Complexity, worst case, average case, optimal case

Looking back to the study of the complexity of an algorithm, one has to notice that several complexities can be defined. More specifically,

- the complexity in the most favorable case and that in the worst case, *i.e.*, the mimum and the maximum number of operations strictly required,
- the average complexity, *i.e.*, the complexity obtained by averaging the complexity on a series of cases.

The worst and optimal complexities are usually easy to determine as it is sufficient to look at the extreme configurations. The average complexity, however, requires the introduction of probabilities. These notions are now illustrated by the following example :

Algorithm 2.4 *Searching for a key* x *in the array* Tab.

```
Procedure IsContainedInArray(Tab,x,found)
i ← 1
found ← .FALSE.
WHILE (found = .FALSE.) AND (i ≤ n)
   IF Tab(i) = x THEN found ← .TRUE.
   ELSE i ← i + 1
END WHILE
IF i = n + 1 THEN found = .FALSE. (lost, x has not been found),
ElSE found = .TRUE. (found)
END IF
```

where the goal is to see whether a given key x belongs to the array Tab at an index i, the length of Tab being n. The problem is to determine how many comparisons are performed in this algorithm in the optimal case, the worst case and on average.

If the first entry of the array is x,the algorithm stops after only one test and its optimal complexity is 1. If the array does not contain x, n comparisons are involved. To find the average, in the case where x exists in the array, we find a number of comparisons equal to :

$$E(x \in Tab) = \sum_{i=1}^{n} i\,Pr(x, i)$$

where $Pr(x, i)$ represents the probability of finding x at the index i of the array. As $Pr(x, i) = \frac{1}{n}$, we find that $E(.) = \frac{n+1}{2}$, which can be written as $E(.) = \Theta(n)$. In other words, if the array is not sorted, a linear number of tests (in n) is expected on average.

These three measures of complexity show different information. More precisely, the worst complexity is a good measure whenever the time to run an algorithm has been fixed. We find here the concrete situations where we expect to have a given complexity, for instance linear, depending on the sizes of the input of the problem. In this case, in fact, it is the worst case that brings the desired information.

Exercise 2.7 *The previous example, in the line* **WHILE**, *requires two tests of equality and a comparison, hence three tests. Show that the number of tests can be reduced to only one if x, the sought key, is inserted in the array at the index $n+1$ (by increasing its size by 1, this new node being called* sentinel*).*

2.3.3 Amortized complexity

Another notion of complexity, known as the amortized complexity, measures the average performance of each operation in the worst case. More precisely, some algorithms or structures are such that the more costly operations are very rarely executed. In some other cases, a costly operation means that the required operations will be inexpensive afterwards. An example of such a behavior is given in the case of a stack.

Let consider a stack and let us assume that the operator *Multi-Pop(Pile,k)* which consists in applying the operator *Pop* to k items (for k smaller or equal to the size of the stack). Let us look at a sequence of n *Push*, *Pop* and *Multi-Pop*. A *Push* and a *Pop* are in $\mathcal{O}(1)$ while a *Multi-Pop* is in the worst case in $\mathcal{O}(n)$, hence, the worst case for a sequence of n operations is in $\mathcal{O}(n^2)$. Is this the true complexity of the algorithm ?

As each item cannot be popped more than once, the number of *Pop* (*Pop* and *Multi-Pop*) is at most equal to the number of *Push*, that is about $\mathcal{O}(n)$. Then, whatever the value of n, a sequence of *Push*, *Pop* and of *Multi-Pop* takes a time in $\mathcal{O}(n)$. Indeed, the amortized complexity of the operation is in $\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$. To summarize, this quantity measures the average efficiency of each operation in the worst case during the execution of a set of operations.

2.4 Sorting and searching

Numerous methods may be used to sort an array containing items on which an order relationship is defined.

2.4.1 Sorting by comparison algorithms

Sorting by insertion. Let us consider an array of size n and let us assume, at the stage i of the processing, for $i = 1, \ldots, n-1$, that the sub-array i (between the indices 1 and i) has already been sorted. The algorithm of *sorting by insertion* consists of inserting $v = Tab(i+1)$ in the sub-array i in the right place, by moving the items greater than v towards the right. This can be written as follows :

Algorithm 2.5 *Sorting by insertion from the smallest to the largest.*

```
Procedure InsertionSort(Tab)
FOR i = 2,n
   key ← Tab(i)
   j ← i − 1
   WHILE j > 0 AND TAB(j) > key
      TAB(j + 1) ← TAB(j)
      j ← j − 1
   END WHILE
   TAB(j + 1) ← key
END FOR
```


to obtain a sorting algorithm from the smallest to the largest. The relevant quantity in the analysis of this sorting algorithm is clearly the number of items moved towards the right side. If the input is seen as a permutation of n different numbers, this quantity can be easily seen as the number of inversions in this permutation (*i.e.*, the sum for all elements of the number of larger items located on the left hand side).

If the array is originally sorted in the reverse order, the worst complexity is obtained. It corresponds to $\sum_{i=1}^{n} i$, that is $\mathcal{O}(n^2)$. Finding the average complexity is more difficult and requires constructing a random model. Here, we simply indicate that the required number of permutations is equally probable. Under this assumption, we can demonstrate that the average number of permutations is in $\mathcal{O}(n^2)$.

While not really efficient in the worst case and on average, a sorting algorithm by insertion is still very useful for small examples or in cases where the data are already almost ordered. In the last case, a simple comparison is sufficient to decide whether or not each item is in the right place. Hence, this sorting algorithm is almost linear in time.

Exercise 2.8 *Indicate how to use this sorting algorithm to sort a linked list (its keys). Is the complexity of the process affected ?*

Quicksort. The quicksort algorithm sorts in place, that is by permutating the data. This method is widely used because it has proven to be robust, efficient and easy to implement. Its efficiency is close to the optimum. Its robustness is mainly related to the fact that it is insensitive to the properties of the values to be sorted. The ease of implementation is due to the simplicity of the underlying concept.

Indeed, the main idea consists in taking one item of the array, the *pivot*, and splitting the array into two pieces around the pivot. The elements which are larger (resp. smaller) are placed on the right hand side (resp. left hand side) and the process is iterated on each of the sub-arrays. The splitting procedure is also simple and consists in scanning the array and exchanging the elements larger than or smaller than the pivot (once it has been fixed). The sorting algorithm can be written as follows :

```
Quicksort(Tab,left,right)
```

where Tab is the array to be sorted and $left$ and $right$ are the left and right indices of this array (for instance, $left = 1$ and $right$ is the number of values in Tab). This procedure is written :

Algorithm 2.6 *Quicksort from the smallest to the largest.*

Procedure `Quicksort(Tab,left,right)`
`IF` $left < right$
  $m \leftarrow Partition(Tab, left, right)$
  $Quicksort(Tab, left, m-1)$
  $Quicksort(Tab, m+1, right)$
`END IF`

The procedure *Partition* corresponds to the following algorithm :

Algorithm 2.7 *Procedure Partition for the Quicksort algorithm.*

Procedure `Partition(Tab,left,right)`
$ipivot \leftarrow \frac{left+right}{2}$
$pivot \leftarrow Tab(ipivot)$
$j \leftarrow left$
Exchange $Tab(left+1)$ and $Tab(ipivot)$
`FOR` $i = left+1$ `TO` $right$ `DO`
  `IF` $TAB(i) < pivot$,
    $j \leftarrow j+1$ and exchange $Tab(i)$ and $Tab(j)$,
  `END IF`
`END FOR`
exchange $Tab(left)$ and $Tab(j)$
`RETURN` j

In [Knuth-1998b], it is proved that the average complexity of the *quicksort* is in the order of $1.38\, n \log n$, where the logarithm is taken in the base of 2, thus leading to $\mathcal{O}(n \log n)$. This is interesting because the sole operation used in the sorting algorithm is the two-two comparison of the elements. It is then possible to prove that, on average, each sorting algorithm requires a minimum number of comparisons is on the order of $\Omega(n \log n)$, in the worst case.

Notice however, that the worst complexity is in $\mathcal{O}(n^2)$. This is obtained when the array is already sorted in such a way that, at each step of the recursion, the smallest (resp. largest) element is picked as the pivot. To avoid this unbalanced case, a solution consists of choosing the pivot in a different way. One strategy can be to choose an element randomly in the array. But, using a random number generator in this case can be seen as superfluous. The easiest way to improve the efficiency is to use the technique known as the *average of three* in which the pivot is chosen as the average value of the element on the left, middle and right of the array.

The version of this algorithm described above can be improved in two different ways, thus leading to almost 30 % speedup, depending on the implementation. Firstly, the recursion can be avoided by using a loop. This requires storing the bounds of the sub-arrays for a further processing. Then, to avoid the time devoted to pushing and poping the data because of the recursion, a simple sorting algorithm by insertion can be performed in place of a recursive call when the number of

entities to be sorted is very small. In a similar way, the small sub-arrays can be kept (not processed) during the recursion and sorted by insertion in the whole set of data. The critical size below which a recursion is not efficient depends on the implementation and is usually between 2 and 25.

Exercise 2.9 *Analyze the case of a quicksort algorithm that does not account for the sub-arrays of size smaller than k, these being processed using a sorting algorithm by insertion on the whole set. Show that the expected complexity is in $\mathcal{O}(n\,k + n \log \frac{n}{k})$.*

Exercise 2.10 *Replace, in the algorithm* `Quicksort(Tab,left,right)` *above, the two recursive calls by a loop using a stack for the two sub-arrays.*

2.4.2 Bucket sorting

We have just seen that the quicksort algorithm is based on two-two comparisons of the elements. Hence, the number of operations is only related to the order of the data and not to the specific value to be sorted.

A dramatically different approach for sorting is based on the use of the value of the entities to be sorted in such a way as to separate them. More specifically, the domain containing the entities to sort is divided into equally sized pieces of size δ and each entity is associated with the block containing it. This association is obtained using a function (integer part), denoted $\lfloor\ \rfloor$, and then, all items belonging to the same block are chained together in a linked list.

The general scheme of a (recursive) *bucket sort* is the following :

Algorithm 2.8 *Recursive bucket sort.*

Procedure `Bucketsort`$(x_1, x_2, \ldots, x_n)$
$x_{min} \leftarrow min(x_1, x_2, \ldots, x_n)$
$x_{max} \leftarrow max(x_1, x_2, \ldots, x_n)$
$\delta \leftarrow \lfloor \frac{1}{n_b}(x_{max} - x_{min}) \rfloor$
`FOR` $i = 1, n$
  $idx \leftarrow \lfloor \frac{1}{\delta}(x_i - x_{min}) \rfloor$
  add x_i in $List(idx)$
`END FOR`
`FOR` $i = 1, n_b$
  `IF the size of` $List(i)$ `is at least 1,` $Bucketsort(List(i))$
`END FOR`

Remark 2.1 *The number n_b of blocks can be taken to be equal to the number n of data, so as to achieve an balanced partition. However, when n becomes very large, it is desirable to chose $n_b < n$ (if only for memory allocation problems).*

If each block contains $\mathcal{O}(1)$ points after the block construction, then the partition is said to be balanced. In this favorable case, the sorting algorithm is of linear complexity. This is not certain if some blocks contain a lot of points and, in this case, a solution consists of sorting these points recursively. Moreover, if at

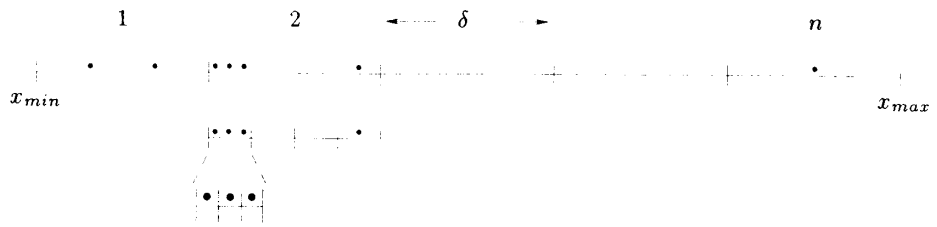

Figure 2.7: *Example of a block defined during the construction of the partition related to the bucket sorting.*

each stage, all points but one belong to the same block, for instance if $x_i = i!$, for $i = 1, ..., n$, the recursion requires $n-1$ levels and the time is $\sum_{i=1}^{n} i$ that is $\mathcal{O}(n^2)$.

The efficiency of bucket sorting is thus related to the number of points to be sorted in an interval of size δ, which is related to the distribution of the x_i's. More precisely, in [Devroye-1986], it is shown that points for which the distribution function is of compact support and of square integrable[2] are sorted in linear time by recursive bucket sort[3]. Intuitively, this just means that although some regions seem to be highly populated in the original sample, a good separation of the points is achieved after a small number of recursions with interval lengths δ.

Remark 2.2 *We have described here the bucket sort for one dimensional data (integer or real numbers for instance). This technique can also be applied to data in $\mathbb{R}^2$ or $\mathbb{R}^3$, for example points that need to be sorted according to various criteria.*

Remark 2.3 *Notice also, as for the previous examples, that numerical problems are not taken into account in the discussion. Indeed, what happens if a point is located on the right-hand side of a point (being considered as larger, within the roundoff errors) when it should be on the left-hand side (mathematically speaking) ?*

Leaving these remarks to one side, does this mean Bucket sort should be preferred to Quicksort when the dataset to be sorted is known to have certain properties ? The answer is not that clear : Quicksort sorts an array in place whereas Bucket sort requires more memory (several points may fall within the same bucket). So, depending upon the implementation, the $\mathcal{O}(n)$ time algorithm may outrun another one in $\mathcal{O}(n \log n)$ for some sample sizes.

2.4.3 Searching algorithms and dichotomies

In Section 2.3.2, we have analyzed the performances of a sequential search in an array and we have shown that a linear time complexity can be achieved for a

[2] If f is this density function, f is said to be square integrable if the integral of its square converges, *i.e.*, $\int f^2 < \infty$.

[3] Sufficient conditions are usually well-known for a probability density to be sorted in linear time. But coming up with necessary conditions remains an open issue.

fruitful search as well as for a failure. We now turn to a more powerful strategy, the *dichotomy* that leads to a result (positive or negative) in a $\mathcal{O}(\log n)$ time.

The intrinsic weakness of a sequential search is that, at each step, the comparison performed between the sought value x and the part of the array analyzed does not provide any global information about the array. However, for a sorted array, this drawback can be avoided. Indeed, by comparing x with the element in the middle of the array, one can decide which part of the array should contain x. A simple comparison thus allows the number of potential candidates to be divided by two. As the size of the problem decreases by a factor 2 at each step, the size of the searching domain is reduced to 1 after $\log n$ comparisons. Such a searching algorithm based on dichotomy follows the general scheme :

Algorithm 2.9 *Dichotomy search of an item x in an array Tab.*

```
Procedure IsContainedInSortedArray(Tab,x,found)
l ← 1, r ← n,
WHILE l ≠ r
   idx ← ⌊(l+r)/2⌋
   IF x ≤ Tab(idx), THEN r ← idx
   ELSE l ← idx + 1
   END IF
END WHILE
IF x = Tab(l), found = .TRUE.,
ElSE found = .FALSE.
END IF
```

Despite its performances, we must notice that a dichotomy search is not always the best method. For instance, if we want to find a name beginning by B in a phone book, it is usually recommended to start from the beginning of the book rather than from the end ! This simple remark suggests an improvement in the binary search. When the value x is searched for in a sub-array indexed from l to r, we start by estimating the place where x is most likely to be. The natural way of doing this is to interpolate x in the sub-array $Tab(l, ..., r)$, which is equivalent to replacing the index idx in the above scheme by :

$$idx = l + \left\lfloor (r-l)\frac{x - Tab(l)}{Tab(r) - Tab(l)} \right\rfloor .$$

For this reason, the searching algorithm, known as interpolation-search algorithm or interpolating search, is very similar (in its concept) to a bucket sort. Its efficiency is strongly related to the properties of the dataset. It can be proven that for a large variety of datasets (in terms of density), a fruitful search or a failure can be achieved in $\mathcal{O}(\log \log n)$.

Another search strategy of the same kind consists in using a bucket tree to store the elements. Similarly, the complexity is related to the partition density of the elements, see [Devroye-1986].

Exercise 2.11 *The* WHILE *statement in the above algorithm is skipped if* $Tab(idx) = x$. *Does this affect the complexity of a successful search (x is found) or a failure (x is not in Tab) ?*

2.4.4 Three main paradigms

Before going further, we introduce three paradigms that will be used later and which can be considered fundamental in algorithmics.

The first paradigm is known as **Divide-and-conquer**. Briefly speaking, it consists in solving a problem $\mathcal{P}$ by

i) dividing $\mathcal{P}$ into several sub-problems, for instance into two sub-problems $\mathcal{P}_1$ and $\mathcal{P}_2$ (of smaller sizes),

ii) solving the sub-problems, here $\mathcal{P}_1$ and $\mathcal{P}_2$,

iii) merging the partial solutions together.

The recursive division stops in practice when the sub-problem becomes sufficiently small and its solution is easy to obtain. The number of sub-problems created and the way of merging the solutions are related to the nature of the problem in hand. For instance, the Quicksort divides a problem in two and the merging operation simply consists in putting the solutions end to end.

The second paradigm concerns the **Computational model**. This consists in defining the type of operations that may be used when devising an algorithm. For instance, in quicksort or in bucket sort, we noted the difference between the former approach, which uses only pairwise comparisons, and the second approach which requires the floor function $\lfloor\ \rfloor$. We noticed that this resulted in complexities that are sensitive to various properties of the dataset (random permutations versus probability densities in our examples).

The last one is the **Randomization** paradigm which we encountered with the pivot selection in the quicksort algorithm. Broadly speaking, making choices at random is an elegant and efficient way to avoid worst-case complexities with high-probability. Intuitively speaking, if an event of bad complexity occurs with some probability, having it occur over a long sequence is very unlikely. Thus, this technique will be widely used in the following.

These three techniques are of course independent. In particular, quicksort-like strategies as compared with bucketsort-like ones can be viewed as two independent implementations of the Divide-and-Conquer paradigm. The former splits the task into a constant number of sub-problems while the latter attempts to make decisions faster using a higher branching factor.

Exercise 2.12 *Let* $List_1$ *and* $List_2$ *be two ordered linked lists of sizes n and m respectively. Show that they can be merged in time* $n + m - \nu(n, m)$ *by changing pointers only, with* $\nu(n, m)$ *the number of items of* $List_1$ *bigger than the largest element of* $List_2$ *(or vice versa).*

Let List be a list containing n items. Show that List can be recursively sorted by first splitting it into two equally sized sub-lists, sorting these sub-lists and merging them. What is the time complexity of this method ? (This sorting algorithm is known as the Merge sort*).*

Exercise 2.13 *Let* $L_0 = (x_1, x_2, ..., x_n)$ *be a set of n real numbers. Suppose also we have a coin and define* L_{i+1} *from* L_i *as follows : for each x in* L_{i+1}, *toss the coin and add x to* L_{i+1} *if the output is heads. Now, call e the first integer such that* L_e *is the empty-set. Show that* $E(e) = \mathcal{O}(log\, n)$ *and* $Pr(e \geq \alpha\, logn) \leq \frac{1}{n^{\alpha-1}}$.

2.5 One dimensional data structures

One-dimensional data structures for handling (one-dimensional) objects are considered amongst the most fundamental since they are the building blocks on which more involved algorithms and data structures are based. In Section 2.2, we saw how to store unordered objects. In this section, we shall see how to define *dictionaries* and *priority queues*. As for the data structures already described, these can easily carry out operations such as "is this element contained in", "insert" or "delete" an element and, moreover, are more geared to handling requests such as "find the min", "find the max".

2.5.1 Binary tree

In Section 2.4.3, we used a Divide-and-Conquer paradigm to search a sorted array. We noticed the running time improvement over the naive algorithm of Section 2.3.2. In this section, we do the same in a dynamic context based on a *Binary Search Tree* (referred to hereafter as *BST*), that improves the complexity of any searching operation on linked lists. We begin first with a definition.

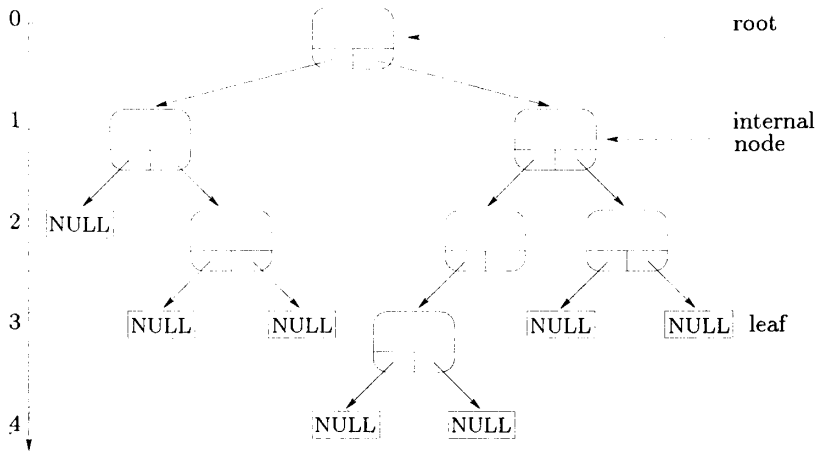

Figure 2.8: *Pointer representation of a binary tree.*

Definition 2.1 *A binary tree is a data structure whose node contains, together with the information field (the key), two pointers, the left and the right children. If in addition the information field obey some ordering relationship, such a tree is called a* Binary Search Tree *(BST).*

The topmost node is called the *root* of the tree. A distinction is made between a node that has children, called *internal*, and a node without children, called *external* or a *leaf*. The *depth* (height) of a node is the number of edges (branches) crossed from the root to that node (Figure 2.8).

An example of binary search tree growth is illustrated in Figure 2.9 where the insertion of the values 17, 5, 1 and 3 is depicted. First, 17 is inserted and stored at the root since the tree is empty. Then 5 is inserted and put in the left subtree as it is smaller than 17. Similarly, 1 goes to the left of 5 and 3 to the right of 1. The resulting tree has one internal node at depths 0, 1, 2 and a leaf 3, it has an edge at depths 1, 2 and 3.

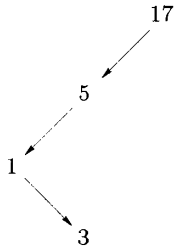

Figure 2.9: *Binary tree constructed from the sequence of keys* 17, 5, 1 *and* 3. *Notice that in this tree, according to the sequence of the values, the nodes have only one child. A different ordering could lead to nodes having two children.*

To see which parameters influence the operations on *BST*, let us consider the searching and insertion procedures whose general schemes are given below :

Algorithm 2.10 *Searching in a BST.*

```
Procedure Contains(node,x)
IF node = NULL THEN return = .FALSE., END
IF node.key = x THEN return = .TRUE., END
ELSE
   IF x ≤ node.key THEN Contains(node.left,x)
   ELSE Contains(node.right,x)
   END IF
END IF
```

Algorithm 2.11 *Inserting a key in a BST.*

```
Procedure Insert(node,nodeFather,x)
IF node = NULL
   allocate a new node, newnode
   newnode.key ← x
   newnode.left ← NULL and newnode.right ← NULL
   IF x ≤ nodeFather.key THEN nodeFather.left ← newnode
   ELSE nodeFather.right ← newnode
   END IF
END IF
IF x ≤ node.key THEN Insert(node.left,node,x)
ELSE Insert(node.right,node,x)
END IF
```


Basically, the strategy consists in tracing a path down the tree and making the right decision at each node encountered. If a key already present in the tree is asked for, the searching process stops in an internal node or in a leaf (terminal node). If the key is not in the tree, the process ends up in a node and adds a child to it. The average cost of a search is related to the sum of the depths of the node in the tree. As for the worst-cases, these quantities are bounded by the depth of the tree, h_n which is such that $\lfloor log_2 n \rfloor \leq h_n \leq n-1$. On average, see [Mahmoud-1992], the expected depth of a random tree containing n keys is $E(h_n) \approx 2.98 \log n$.

Randomly building *BST* trees is therefore interesting and is easy to handle if one can possibly afford bad performances. However, should this not be the case or should deletions be allowed (very little is known for a random tree after a sequence of deletions and insertions), a different strategy must be applied.

Bad performances clearly arise from "skinny" and "elongated" trees. This is, for example, the case in Figure 2.9. One would prefer the configuration given in Figure 2.10, right-hand side.

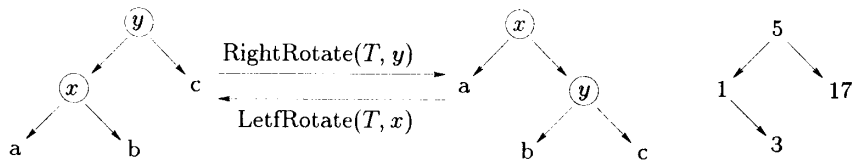

Figure 2.10: *Left hand side : balancing by rotation around a node. Right hand side : balancing the tree of Figure 2.9.*

To avoid this type of situation, the strategy consists in *balancing* the tree to have its subtrees contain roughly the same number of items. The elementary operation towards this goal is the rotation around a node depicted in Figure 2.10.

In practice, there are several methods to balance a tree. For instance, AVL[4] trees, [Wirth-1986], are such that for any node the depth of the two subtrees differs by at most one. For the red-black trees [Cormen *et al.* 1990], the balancing is achieved by some constraints satisfied by the color of the nodes. The reader is referred to the cited references for more details about tree balancing, an operation that can be a bit tricky.

The performances of the red-black trees are summarized in the following theorem :

[4] acronym for Adelson, Velskii et Landis, inventors of this type of tree.

Theorem 2.1 *The depth of a red-black tree containing n keys satisfies $\frac{\log n}{2} < h_n < 2 \log n$. The operations "insert", "delete", "exists", "find the min", "find the max" are in $\mathcal{O}(\log n)$.*

We now give some schemes for traveling through (traversing) a binary tree.

Algorithm 2.12 *Tracing a path in a binary tree.*

```
Procedure inOrderProcessing(node)
IF node ≠ NULL
   inOrderProcessing(node.left)
   process node
   inOrderProcessing(node.right)
END IF
```

Algorithm 2.13 *Tracing a path in a binary tree (pre-order processing).*

```
Procedure preOrderProcessing(node)
IF node ≠ NULL
   process node
   preOrderProcessing(node.left)
   preOrderProcessing(node.right)
END IF
```

Algorithm 2.14 *Tracing a path in a binary tree (post-order processing).*

```
Procedure postOrderProcessing(node)
IF node ≠ NULL
   postOrderProcessing(node.left)
   postOrderProcessing(node.right)
   process node
END IF
```

Exercise 2.14 *A m-ary search tree is defined as a search tree whose nodes have exactly m children (a node contains the key and m pointers). How do m-ary search trees compare against BST in terms of search time and memory requirements ?*

Exercise 2.15 *Show how a red-black tree can be used for sorting in $\Theta(n \log n)$.*

2.5.2 Hashing

Like the bucket sort, the hash functions consist in splitting the dataset processed into bins or buckets. The hashing process is viewed here as a one-dimensional structure, in particular to emphasize the fundamental difference between the hashing technique and the bucket sort.

However, it seems obvious that this structure is adequate for multi-dimensional data and, practically, is commonly used in such situations. If h denotes the hash function and if x is the element considered, then $h(x)$ is the hash value associated with x, as will be seen hereafter. In three dimensions, we will find, with obvious notations, $h(x, y, z)$ the hash value associated to the element (x, y, z).

There is however a fundamental difference between the hash function used in bucket sort, namely $h(x) = \lfloor \frac{x - x_{min}}{\delta} \rfloor$, and general hash functions. While the former is monotonic, *i.e.*, if $x \geq y$ then $h(x) \geq h(y)$, this property is not strictly required in the latter case. Moreover, general hash functions are usually implemented using *modulo*, see Figure 2.11, left-hand side, for instance.

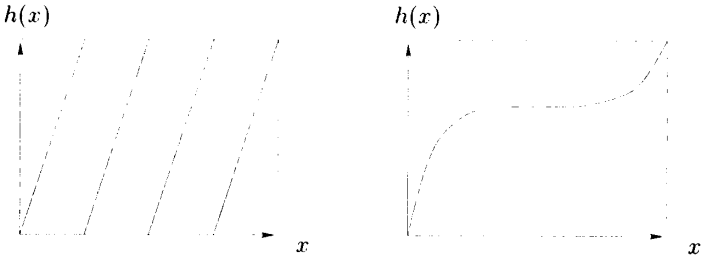

Figure 2.11: *Non monotonic hashing (left) and monotonic hashing (right).*

It should be emphasized that this enables the building and handling of dictionaries[5], but does not allow the processing of proximity queries such as "given $h(x)$, report the neighbors of x from the values of h in a neighborhood of $h(x)$". We shall return to this issue in Section 2.6.

To define precisely a few standard hash functions, we start with the following definitions.

Definition 2.2 *Let $\mathcal{U}$ be a universe,* i.e., *a set of possible keys (0, 1, ...), let $\mathcal{S}$ be a subset of $\mathcal{U}$ of size n, let $\mathcal{T}$ be an array of buckets indexed by a set of integers $\mathcal{I}$. Suppose, in addition, that each bucket is endowed with an auxiliary data structure (linked list, array, BST, ...) that may contain up to b items. Then, a hash function is an application h from $\mathcal{U}$ to $\mathcal{I}$. Two keys x and y are said to* collide *if for $x \neq y$ we have $h(x) = h(y)$. A bucket is said to* overflow *if more than b keys have been hashed into it.*

As an example, consider Figure 2.12. Here, the set of all possible keys is the integer (in this example) range $1, ..., 50$. The values to be hashed are five numbers $\mathcal{S} = (2, 5, 10, 15, 37)$. The hash table is an array of 10 linked lists (thus $b = \infty$). The hash function satisfies $h(10) = h(37) = 1$, $h(2) = 3$, $h(15) = 6$ and $h(5) = 8$. Typically, if one wants to know whether x is stored in the hash table, the algorithm consists in checking whether x is present in the data structure associated with the bucket of $\mathcal{T}$ indexed by $h(x)$. The important parameters are therefore the relative size of $\mathcal{S}$ as compared with that of $\mathcal{T}$ and the number of items referenced within each bucket.

Several types of hash function can be envisaged :

[5] A set on which insertion, deletion and searching perations are defined.

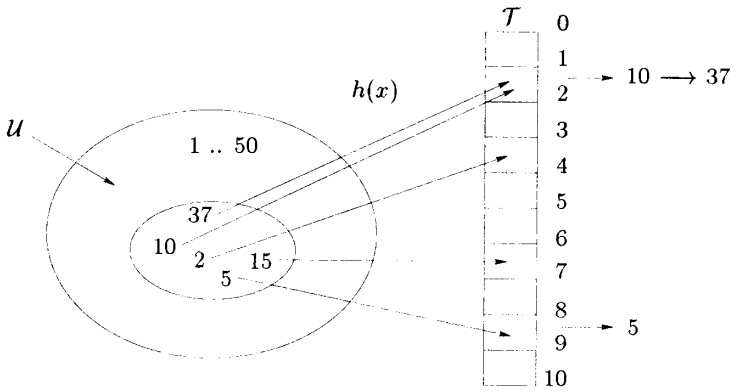

Figure 2.12: *Example of a hash table.*

- universal hashing, for which h is chosen randomly,
- perfect hashing, where h is injective,
- minimal hashing, where $card(\mathcal{S}) = card(\mathcal{T})$,
- dynamic hashing, where $card(\mathcal{S})$ is not known beforehand,
- monotonic hashing, where h keeps the ordering on the keys.

Two main problems are to be considered when implementing a hash technique. The first is related to the choice of the hash function (see above). The second, related to this choice, deals with collisions handling.

The collisions handling is indeed very important. One can of course rehash the whole table if some buckets are overflowing, but that does not tell us how to choose the right hash function.

An initial strategy consists in setting $b = 1$, in which case a single key at most is stored per bucket. This peculiar hashing is called "open addressing". The hash function maps $\mathcal{U} \times \mathcal{I}$ to $\mathcal{I}$ and the sequence of buckets attached to the key x is $seq(x) = (h(x,0), h(x,1), ..., h(x,m-1))$. The function h should be chosen such that for any key x, $seq(x)$ is one of the $m!$ permutations of $0, 1, ..., m-1$ with equal probability. Several such functions are described in [Cormen *et al.* 1990]. Their design is mainly concerned with avoiding overly long common sub-sequences between $seq(x)$ and $seq(y)$.

The second strategy aims at avoiding collisions as much as possible. A fundamental notion in this context is that of universality :

Definition 2.3 *Given two integer ranges $\mathcal{U} = 0, ..., n-1$ and $\mathcal{I} = 0, ..., m-1$, with $n \geq m$, a family of hash functions $\mathcal{H}$ is called 2-universal if for any x_1 and x_2 of $\mathcal{U}$ such that $x_1 \neq x_2$ and h chosen at random in $\mathcal{H}$, the following holds :*

$$Pr(h(x_1) = h(x_2)) \leq \frac{1}{m}.$$

Interestingly, the condition :

$$x_1 \neq x_2, y_1 \neq y_2, \quad \text{AND} \quad Pr(h(x_1) = y_1, h(x_2) = y_2) = \frac{1}{m^2}$$

implies the previous condition and corresponds to a 2-universal hashing called *strong*. The idea is to have the images of two points behave as independent random variables.

2.5.3 Priority queues

Many situations require the records to be processed in order, but not necessarily in full order and/or not necessarily all at once. For instance, an algorithm may require the highest value to be processed, then more values to be collected, etc. The data structures supporting this kind of operation are called *priority queues* and can be viewed as generalizations of stacks or queues. More precisely, a priority queue is a data structure containing records with numerical keys (numerical values), the priorities, and supporting the following operations :

- the construction of a priority queue for a set of items,
- the insertion of a new value,
- the search for the maximum,
- the modification of the *priority* of an item,
- the deletion of an arbitrary specified item,
- the merging of two priority queues.

Interestingly enough, the red-black tree data structure turns out to be a simple yet efficient implementation for priority queues. From the discussion of Section 2.5.1 and according to Theorem 2.1, we know that the insertion, deletion and search operations have a $\mathcal{O}(log\ n)$ complexity. Moreover, the construction, the modification of the priority and the merge requests simply require a sequence of insertions and deletions.

More sophisticated implementations of priority queues provide better complexities for the construction, modification of priority and merge operations. In particular, a fashionable data structure for that purpose is the *heap* data structure, which is a complete binary tree with the property that the key in each node should be larger than (or equal to) the keys in its children (if any). The reader is referred to [Sedgewick-1988] and [Cormen *et al.* 1990] for further reading.

Exercise 2.16 *Show how to implement a priority queue using an ordered linked list. What are the complexities of the "merge" and "change priority" operations ?*

Exercise 2.17 *Suppose a red-black tree is endowed with an additional pointer to the maximum (or minimum) entry stored. Give the complexity of the "find the Max" ("find the min"), "insert" and "delete" operations.*

2.6 Two and three-dimensional data structures

After one-dimensional data structures, we now turn to multi-dimensional data structures. Here, we find grids and trees (quadtrees and octrees). Described in this section as data structures, these structures will be viewed again (Chapters 5 and 8) as they also serve to construct meshing algorithms.

In this section, we show how several ideas developed for one-dimensional data structures can be reused in two and three dimensions to handle more complex objects such as points, polygons, etc. In Section 2.6.1, we present several grid-like structures, in Section 2.6.2 we consider tree-like structures and in Section 2.6.3 we mention some problematic side-effects that may arise when using these constructions.

2.6.1 Grid-based data structures

Grid-based data structures are the two and three dimensional equivalent of the one-dimensional bucket-like data structure. Interestingly, their performances vary greatly depending on the kind of items processed. If points are stored in two- or three-dimensional grids, very precise theoretical results have been known for a while, see [Devroye-1986] for example, that give indications about the behavior of the operations involved with such grids. However, when more complex objects are involved, such as polygons, many theoretical results remain to be established and justified experimentally. The aim of this section is precisely to present guidelines for the efficient use of grid-like partitions for applications such as mesh generation. A good starting point for further reading is [Cazals,Puech-1997].

Let $\Delta = \Delta_x \times \Delta_y \times \Delta_z = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$ be some three-dimensional domain containing a set $\mathcal{E}$ of n objects, which may be polygons, etc. Suppose we also want to answer proximity requests over the items of $\mathcal{E}$. For example, we want to find the closest polygon to a given point or find the pairs of intersecting polygons.

Similar to the bucket sort for the one-dimensional case, a nice way to address this class of problems consists of subdividing Δ, the domain of interest, into $n_x n_y n_z$ axis aligned boxes (voxels) of size $\delta_x \delta_y \delta_z$ with $\delta_x = \frac{\Delta_x}{n_x}$ (and similar values for the other dimensions) and having each voxel reference the items intersecting it. Once this pre-processing step has been done, the voxel containing a point $P(x, y, z)$ is identified by the triple ind_x, ind_y and ind_z with, for instance, $ind_x = \frac{x - x_{min}}{\delta_x}$ (and similar expressions for the other indices). From this voxel, the items of $\mathcal{E}$ close to P are easily retrieved. In order for this construction to be efficient, two types of constraints must be taken into account. First, each voxel should reference a small number of items of $\mathcal{E}$. Then, the memory requirements should be affordable, *i.e.*, $n_x n_y n_z = \mathcal{O}(n)$ with a small constant. We discuss here three grid-based data structures, Figure 2.13, that achieve these goals for different input datasets.

A grid is in fact a structure that can be uniform, recursive or related to a hierarchy of uniform grids.

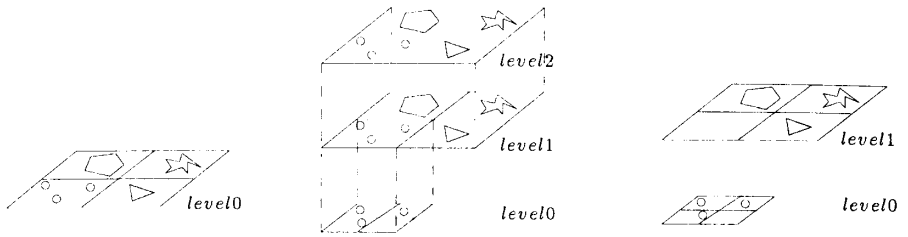

Figure 2.13: *The three types of grid-based structures. Left, a uniform grid, middle, a recursive grid and right, a hierarchy of uniform grids.*

Uniform grid. A uniform grid for the set $\mathcal{E}$ is a partition of its bounding box into $n_x n_y n_z$ subdivisions of equal lengths along the x, y and z axes. To get a memory requirement that is linear in the number of objects, the n_i are usually taken so that $n_i = \alpha_i \sqrt[3]{n}$. The α_i are positive constants and the simplest choice is $\alpha_x = \alpha_y = \alpha_z = 1$. Other choices may be preferred, for instance related to heterogeneous values based on the ratios of the dimensions of the box, the amount of memory available, etc.

Uniform grids provide a simple and efficient way of handling uniform distributions. But their performances get catastrophic for more structured datasets, so other solutions have to be found.

Recursive grid. Once a uniform grid has been built for a set $\mathcal{E}$, one may find that some voxels are too populated (they record too many items). If $maxi$, standing for the maximum number of items, is some positive integer (e.g. 50), a *recursive grid* for $\mathcal{E}$ is a hierarchical structure based on uniform grids such that whenever a voxel contains N items, $N > maxi$ (the initial voxel being the bounding box) it is recursively split into a uniform grid of approximatively N voxels.

Especially if no memory limitation is set, recursive grids provide the simplest and fastest implementation for handling many proximity problems. For unevenly distributed inputs, the performance gain over uniform grids can be up to several orders of magnitude.

Hierarchy of uniform grids. A weakness of the previous construction is that the recursion may waste a lot of empty voxels (Figure 2.13, middle). To solve this problem, the blind recursion can be replaced by a process that figures out more cleverly which are the dense areas that should be allocated resources. The strategy proposed in [Cazals,Puech-1997] successively separates the objects according to their size (filtering step), finds subsets of neighbors called clusters within these classes (clustering step), stores each cluster in a uniform grid and finally builds a hierarchy of uniform grids (Figure 2.13, right). For a description of the filtering and clustering steps, the reader is referred to the previous reference.

Nevertheless, the hierarchy of uniform grids offers a flexible and efficient data structure for input datasets with strong coherence properties. This is especially

true since its construction overheads are almost the same as those of a recursive grid.

2.6.2 Quadtrees and octrees

A quadtree is a two-dimensional spatial data structure whose counterpart is the octree in three dimensions. The term quadtree refers to a class of hierarchical data structures whose common property is to recursively decompose some spatial region. Very much like grid-based subdivisions, quadtrees can be used to store a variety of inputs (points, line-segments, polygons, etc.) in any dimension (to simplify, the term quadtree will be employed whatever the dimension). And also similarly to grids, very little is known about the theoretical properties of this kind of structure, other than for points.

The basic idea consists of splitting the region processed into two sub-regions along each axis. In dimension d, a region is therefore split into 2^d children. The way the splitting hyper-plane(a line in two dimensions, a plane in three dimensions) is chosen, together with the recursion termination condition determines the quadtree type. We present below the two basic schemes when the input is a set of n points, see [Samet-1990] for more details.

Point quadtrees. The point quadtree can be seen as a generalization of a binary search tree. In two dimensions for instance (Figure 2.14), each point is stored in a node and serves as the pivot in the subdivision of the associated region into four quadrants. The quadrants are numbered from left to right and from top to bottom (see the figure).

As will be shown, locating a point in a quadtree is an easy matter that requires comparing its coordinates to those of the nodes currently traversed in order to decide which quadrant it is contained in.

The optimal strategy for building a point quadtree depends on whether or not the input dataset is known *a priori*. If it is, choosing at each step the median point along one axis (which luckily may also be the median of the other axis) results in a tree of depth between $log_4 n$ and $log_2 n$. If no *a priori* information is known, the points have to be sequentially inserted and the weakness already mentioned for *BST* trees may arise, *i.e.*, the tree may be very elongated. Finally, similarly to *BST* trees, point quadtrees can be made dynamic, that is support deletions. Nevertheless, this operation is rather tricky, see [Samet-1990].

Overall, point quadtrees are an interesting and versatile data structure which however suffers from several drawbacks. First, the higher the dimension, the higher the number of empty null pointers created . Then, its depth is usually larger than that of grid-based structures. Its use is therefore recommended when recursive grids or a hierarchy of uniform grids are too demanding resource-wise and/or when a dynamic feature for the process is desirable.

Point-Region quadtrees. If one requires the four regions (the quadrants in two dimensions) attached to a node to have the same size, the data structure obtained is called a Point-Region quadtree (*PR-quadtree*). The recursion stops

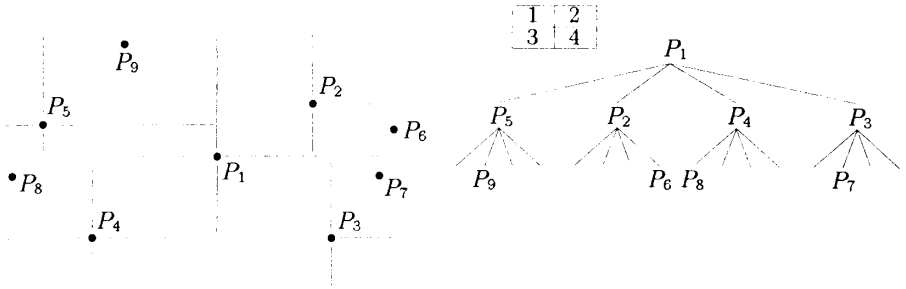

Figure 2.14: *Point quadtree based on a set of points in* $\mathbb{R}^2$.

whenever a quadrant contains at most one data point, so that every point may be stored in a leaf node. As an illustration, consider Figure 2.15 which represents the PR-quadtree for the set of points of Figure 2.14. Before going further, notice that this type of quadtree is the one that will be commonly used in this context (see Chapter 5).

Locating a point in a PR-quadtree requires finding the quadrant it lies in, which is similar (although certainly simpler here) to the previous quadtree case.

Inserting a point starts with a quadrant location. If the quadrant found is empty, insert the point and it is finished. Otherwise, it is refined into four regions (children) and if the other point does not belong to the same quadrant, insert the two points and it is finished, otherwise both points are recursively inserted into the quadrants. It should be clear that many splits may be necessary to separate points located very close to one to another. More precisely, the depth of the tree may be as much as $log_2(\frac{L}{l})$, where L and l are the distances between the closest and farthest pair, respectively. In other words, the depth of the PR-quadtree depends upon the values manipulated (see also Section 2.4.2). In order to avoid wasteful memory allocation, a conservative approach consists of splitting a node only if its depth in the tree is not more than a certain threshold. Alternatively, the leaves can be allowed to accommodate up to *maxi* points, with *maxi* a small integer. Deletion of a node is considerably simpler than for point quadtrees because there is no need to rearrange the tree as all values are stored in the leaf nodes. However, the deletion of a node having exactly one brother should be followed by a collapse step of the four leaves.

In conclusion, PR-quadtrees offer an interesting alternative to usual point quadtrees especially because it is easier to obtain a dynamical aspect. However, one has to be careful about the height of the tree which may become very great if two points happen to be very close to each other. Hence, grid-based data structures are usually (much) more efficient because of their greatly reduced depth.

2.6.3 About filters and side-effects

Numerous operations on meshes and triangulated surfaces require *a priori* running more or less tricky and expensive algorithms. For example, in order to compute

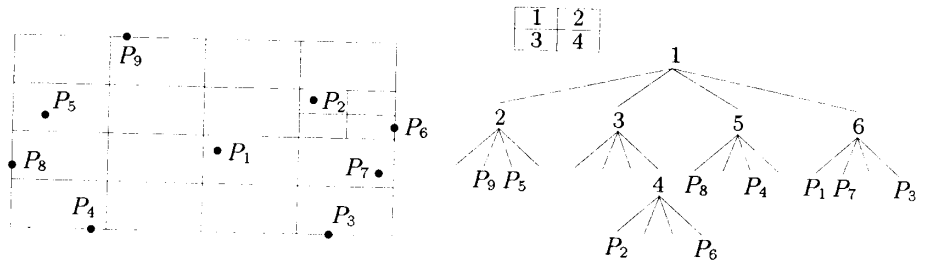

Figure 2.15: *The equivalent Point-Region quadtree data structure.*

the intersection between a line and a discretized surface (a triangulation), one has to compute pairwise intersections between the line and each triangle. Similarly, to render and plot such a (meshed) surface on a graphical device using the ray tracing method[6], one has to intersect lines (in $\mathbb{R}^3$) with the surface. Because this could be very expensive, as with any algorithm involving two geometric entities, a conservative approach consists in first making sure some necessary conditions are satisfied before running the computation. For example, for a ray intersecting a polygon, the intersection point P is necessarily contained in the bounding box of the polygon. If this is so, it must be further checked whether P lies within the polygon and this requires a linear time in the number of vertices of the polygon (see [O'Rourke-1994]), otherwise, any further check is unnecessary.

Suppose now that the polygons are accessed through a grid-like data structure. In order to lessen the number of entities referenced by each voxel and thus the number of polygons tested (for intersection for a given ray), one is tempted to reduce the voxel sizes (thus increasing the total size of the grid in memory). By doing so, one also increases the probability of the ray belonging to the rectangle bounding box. Put differently, reducing the number of intersections results in more expensive computations because the coarse filter given by the bounding box becomes less efficient. This kind of side effect should be borne in mind when performing fine tuning of an algorithm.

2.7 Topological data structures

For the sake of simplicity, we restrict ourselves here to the two-dimensional case and we only consider triangulations. Not surprisingly, the central building block to describe such meshes is the triangle, together with a couple of data structures encoding the adjacency relationships (in some sense). For surface meshes, a more general structure is necessary (a triangle could share an edge with more than one other triangle).

[6] Very briefly, the ray-tracing method consists of figuring out, for a given scene and viewpoint, the color of each pixel in the image representing this scene is rendered into has to be painted with. For a given pixel, this is done by casting a ray onto the scene from the viewpoint that intersects the pixel. The pixel color is deduced from the objects hit by the ray together with the contributions of the relected and refracted rays.

Triangulations (meshes) can be represented in many different ways. First, we present a representation based essentially on the triangles themselves. Then, we briefly discuss another representation based upon the triangle edges.

2.7.1 A triangle based representation

The triangles are indexed from 1 on. A triangle is an (oriented) triple of vertices (cf. Chapter 1). With each triangle are associated its (at most) three edge neighbors (two dimensional case). The neighborhood relationships are encoded such that

$$k = Neigh(j, i)$$

which means that the triangle of index k is adjacent to the triangle of index i and that the edge j of triangle i is the common edge ($k = 0$ means that the edge j of triangle i is a boundary edge). Suppose also that vertex j of triangle i is opposite to edge j of this triangle (see also Chapter 1).

The pair of triples *vertices-neighbors* is one possible way of representing a triangulation (and probably the most concise one) also called the *adjacency graph* of the triangulation.

A richer representation is based upon the triples of vertices and the connection matrices. Each triangle K is endowed with a 3×3 matrix defined as follows (Figure 2.16) :

- the diagonal coefficient i, c_{ii} indicates the local index of the triangle vertex opposite to triangle K by the edge i,
- another coefficient c_{ij} gives the index of the jth vertex of K in its ith neighbor.

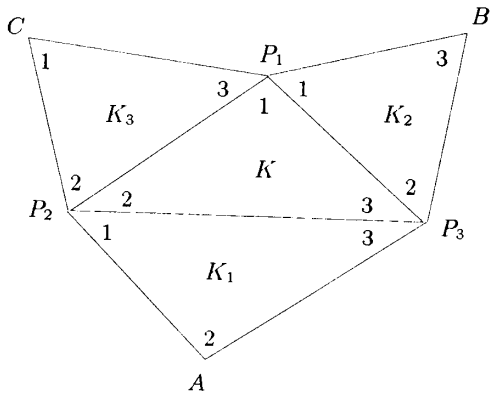

Figure 2.16: *Definition of a triangle K and of its three neighbors K_i. One can see the global indices of the vertices (P_i,A, ...) and the local indices (in any triangle) of these points (1, 2 et 3).*

According to its definition, the adjacency matrix of triangle K in Figure 2.16 is :

$$C^K = \begin{pmatrix} 2 & 1 & 3 \\ 1 & 3 & 2 \\ 3 & 2 & 1 \end{pmatrix}.$$

Hence, for the first line, point A of K_1 sees K and $c_{11} = 2$ because A is the second vertex of K_1. Similarly, $c_{12} = 1$ because the index of P_2 in K_1 is 1 and finally, $c_{13} = 3$. Notice that, unlike the depicted example, the matrix has no specific property (symmetry, for instance).

This representation is richer than the previous one and makes access to neighborhood items easier, as vertex information is added to element information.

Remark 2.4 *One can notice that the diagonal of this adjacency matrix allows the whole set of coefficients to be reconstructed.*

A question arises at this point : "Which data structure should table *Neigh* be implemented with ?" As pointed out in Section 2.7.1, if the number of vertices of the triangulation is known, so is the number of triangles (an upper bound). In this case, an array can be allocated beforehand, although the triangulation algorithm creates intermediate (transient) triangles. Another solution consists in using a dynamic hash table.

Remark 2.5 *To decide which is the best structure, we face a recurrent problem. Shall we use a "rich" structure, which is more memory consuming but provides more information at once or a "poor" structure which is less expensive but gives less information. The answer is strongly related to the memory available, to the cost of retrieving stored information and also to the cost of updating the structure.*

2.7.2 Winged-edge data structure

Another way to represent a mesh consists in viewing it from the point of its edges. This solution, as opposed to the previous one (for which a constant number of neighbors per face is assumed), allows edges common to more than two faces to be dealt with as well as with faces having an arbitrary number of vertices. We can find here two alternatives, the winged-edge (see [Baumgart-1974], [Baumgart-1975], [Weiler-1985]) and the half-edge (see [Mäntylä-1988], [Kettner-1998]) data structures which are edge-based structures. This kind of structure, also described in [Knuth-1975], essentially allows the following operations :

- walk around the edges of a given face,
- access a face from the adjacent one when given a common edge,
- visit all edges adjacent to a vertex.

2.7.3 Hierarchical representation

A more general description, useful for triangulations as well as for arbitrary meshes (manifold or non-manifold, conforming or not) is based upon the exhaustive enumeration of the relationships between the mesh entities.

Schematically, this description indicates the hierarchy between the entities according to their dimensions (points, edges, faces, elements). Hence, we have a direct link such as :

$$Points \longrightarrow Edges \longrightarrow Faces \longrightarrow Elements,$$

and the reverse link

$$Elements \longrightarrow Faces \longrightarrow Edges \longrightarrow Points.$$

This type of storage, [Beall,Shephard-1997], offers numerous advantages, although it is rather memory consuming and expensive when updating the structure. It provides a direct access to the entities of a higher (resp. smaller) dimension.

For a dynamic type of situation (such as graphical visualization, for instance), this kind of structure is especially attractive.

2.7.4 Other representations

Some peculiar applications can benefit from a very specific organization of the data. For instance, for a two-dimensional triangulation, if each vertex is endowed with the oriented list of the neighboring vertices (sharing an edge), this structure allows the related triangles to be easily retrieved, [Rivara-1986].

Other representations can also be found, for instance the *STL* format which consists in enumerating all faces (elements) using the vertex coordinates (necessarily duplicating this information).

To conclude, one can notice that interchange formats (*STEP, IGES, SET, VDI, CGM, etc.*), although not directly aimed at meshing structures, can nevertheless give some information on how to design structures for meshes.

2.8 Robustness

Non-robustness refers to two notions. First, the result may not be correct (for instance, the convex hull of a set of points is not convex). Then, the program stops during the execution with an error or in a more catastrophic way (the computer crashes) with an overflow, an underflow, a division by zero, an infinite loop, etc. If the algorithm is reputed to be error-free (from a mathematical point of view), this means that its implementation leads to an erroneous behavior. Anyone who has implemented a geometric algorithm is likely to have faced this type of problem at some point.

This section has several aims. First, we give a very brief overview of the potential reasons why numeric problems arise; we recall how real numbers are encoded on most computers, and explain why the issues are even more difficult

in a geometric context. We then provide some guidelines to reduce these risks, and finally we give an overview of the state-of-the-art techniques used to make floating-point operations robust.

2.8.1 Robustness issues

Numerical issues in scientific computing have been known since the early days of computers. The core of the problem lies in the limited resources used to encode numbers (the bits) and the drawbacks are twofold. First, the biggest and smallest numbers that can be represented are upper and lower bounded, so that some calculations cannot be carried out if the intermediate value exceeds these bounds. Second, real numbers have to be represented approximatively since one cannot squeeze infinitely many of them into a finite number of bits. These difficulties can yield to erroneous results and also generate undefined operations such as $\sqrt{x}$ with $x < 0$ or $x \times y$ with $x = 0$ and $y = \infty$. The unpredictability of floating point operations across different platforms led in the eighties to the adoption of the IEEE-754 standard [Goldberg-1991], [Kahan-1996]. In addition to the previously mentioned exceptions, this standard also defines several floating-point storage formats and templates for the $+, -, \times, \div$ and $\sqrt{\cdot}$ algorithms, that is, any implementation of an operation should produce the same result as the operation provided by the standard. Note that this makes calculations consistent across different architectures, but does not eradicate exceptions at all.

More practically, the way floating-point numbers are represented on most modern processors is in the form $mantissa \times 2^{exponent}$ where the mantissa is represented by a p bits binary number. For example, $p = 24$ (resp. $p = 53$) for simple (resp. double) precision in the IEEE-754 standard. Rephrasing the issues raised above, how should one proceed when calculations produce numbers that cannot be represented using that many bits ? If one does not care so much about exactness, the standard format specifies how such results have to be rounded to fit back into the finite representation. If one does care about exactness, one needs to switch to another representation.

One solution is the *multiple-digit* format based on a sequence of digits and a single exponent. Another solution is the *multiple-term* format where a number is expressed as a sum of ordinary floating point words. This latter approach has the advantage that the result of an addition such as $2^{40} + 2^{-40}$ is encoded in two words of memory while the multiple-digit solution requires 81 bits and incurs a corresponding speed penalty when further processed.

2.8.2 Computational geometry

From the computational point of view, geometric computations are even more difficult to handle than pure numerical calculations. A simple remark makes this point clear. Consider for instance the computation of the solution of a general matrix system by any suitable method. A solution close to the exact one, within a given precision, is usually obtained, except in some pathological cases. Let us now consider two examples of geometric calculations.

The first example concerns the problem of computing the convex hull of a set of points $\mathcal{S} = (P_1, ..., P_n)$ in the plane.

Clearly, two points P_i and P_j contribute to the convex hull if all the remaining points lie on the same side of the line passing through P_i and P_j. So that the only *predicate* one needs for the computation is the so-called orientation test that, for three points A, B and C, indicates whether C lies to the left of, on or to the right of the line passing through A and B (Figure 2.17).

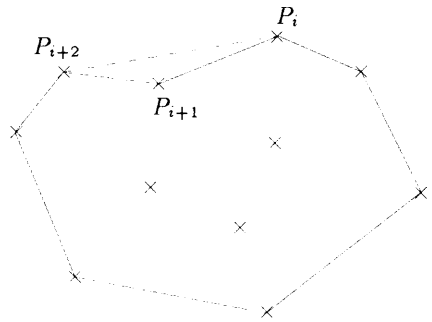

Figure 2.17: *A convex hull that is non convex !*

Using this predicate, the output of any convex hull algorithm consists of a combinatorial structure over $\mathcal{S}$, namely the list of vertices defining the convex hull, together with a consistency condition stating that the corresponding polygon is convex. But should a predicate calculation yield an erroneous result, the result computed may not be convex (as on the figure where P_{i+1} was reported to lie on the right side of P_iP_{i+2}). The result may be 100 % erroneous with respect to the goal aimed at, but one could always consider that this result is close enough to the theoretical result, if the error is small, as in the previous example, a correct answer about a few percent.

The second example corresponds to a more critical situation. To construct a Delaunay triangulation using an incremental method, one has to determine whether a given point belongs to the circumscribed disk (in two dimensions) of an element. The predicate used in this case is the *inCircle* predicate described above. A slight error, however minor, in the answer can lead to a result (here a triangulation) that may be correct (but that does not satisfy the Delaunay criterion) or to an erroneous result (overlapping elements). With respect to the goal expected, one can get an approximate answer within a few percent (the result is valid but the triangulation is not Delaunay) or a totally wrong answer (no triangulation at all).

Here, we have emphasized two different cases, one where a result is obtained and the other one for which the errors are so great that no result at all can be expected. Notice also that such errors can lead to a "fatal" error, a program failure.

2.8.3 The algebraic degree of a problem and predicates

In spite of this apparent difficulty, geometric computations have the nice property of requiring numeric calculations which mostly consist of evaluating algebraic expressions. Examples are distance computations, the orientation predicate already mentioned (*Orientation*), as well as the *inCircle* predicate which indicates whether a point D belongs to the disk whose boundary is the circle passing through three points A, B and C or is external to it.

The common feature between these calculations can be formalized as follows : define an elementary predicate as the sign $(-, 0, +)$ of some homogeneous polynomial over the input variables and its degree as the maximum degree of the irreducible factors. Also define the degree of an algorithm as the maximum degree of its predicates. Similarly, the *algebraic degree* of a problem is defined as the minimum degree of any algorithm solving it. For example, a convex hull algorithm using only the orientation predicate has degree 2. But a Delaunay triangulation using only the *inCircle* predicate has degree 4.

We now introduce the *Orientation* and *inCircle* predicates.

$$Orientation(A, B, C) = \begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_x & C_y & 1 \end{vmatrix}, \tag{2.1}$$

where A_x denotes the x coordinate of point A. Here we analyze C with respect to the line passing through A and B.

$$inCircle(A, B, C, D) = \begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix}. \tag{2.2}$$

We analyze the position of point D with respect to the circle (disk) passing through A, B and C.

Exercise 2.18 *What is the algebraic degree of bucket sort and quicksort algorithms described in Section 2.4 ?*

2.8.4 Robust and efficient floating-point geometric predicates

We are thus interested in evaluating the sign of the predicates. We will assume in the discussion that the sole operations required are the addition and the multiplication. As mentioned above, the main issue of floating-point calculations is related to rounding and we noticed that the arbitrary precision computations solved this difficulty. Unfortunately, the overhead makes these solutions impractical. In this context and especially since we are interested in the sign of the expressions rather than their value, it was observed that floating-point calculations were, in fact, very often reliable, so all that was needed was a way to take care of these confusing situations. To that end, the following paradigms have been proposed.

- interval analysis : the evaluation of an expression is replaced by that of guaranteed upper and lower bounds on its value. When the sign is needed, if the interval does not contain 0, one can conclude. Otherwise, the evaluation must be performed again with higher precision (or by any other method).
- arithmetic filters : the evaluation of an expression is accompanied by that of the maximum absolute error.

Details can be found in [Shewchuk-1997b] and also in [Devillers-Preparata 1998], for instance. Several methods are available (or have been studied) and it seems likely that these methods will be further refined and, possibly, work their way into real-world applications.

Remark 2.6 *In mesh generation, the main idea is to make sure the algorithm will provide a valid result, not necessarily strictly conforming to the theory but usable (a non-convex convex hull is theoretically awkward although it depends on the application envisaged).*

2.9 Optimality of an implementation

The design and the implementation of a computer program performing some task usually requires taking care of various aspects. What is needed is to design the algorithm, to implement and test it and, possibly, to profile it (speed and memory requirements). Optimizing a computer program consists in optimizing its running time and/or its memory requirements, or fine tuning its time-space trade-off. In this section, we briefly review some features any programmer should be familiar with when pursuing these goals (dealing with efficiency and robustness).

2.9.1 Memory handling

Broadly speaking, the memory handling of a computer by the operating system is usually divided into two categories, the *static* and the *dynamic* memories. Basically, the static memory is a chunk used in a stack-like manner to store the local variables and the parameters used in the user-defined procedures and functions. The dynamic memory is a pool the user can request slots from in a dynamic fashion.

In Section 2.2.5, we saw that allocating and de-allocating dynamic memory could be very costly. A good strategy sometimes consists in writing a special case dedicated memory handler using particular features of the requests processed. For example, if one knows beforehand how much space is going to be needed, a linked list or a stack may be better implemented by an array allocated once for all.

Another problem is fragmentation. If too many small slots are requested and freed too often, the memory map may end up like a piece of gruyere cheese. In this case, although a significant amount of memory may be available overall, any request for a big chunk may fail since no such continuous block is available.

Finally, there is another problem worth mentioning. It is desirable to group into memory the data manipulated in a program so as to avoid, as far as possible, *cache*

defaults which are extremely costly. This kind of problem is rarely mentioned. Cache memory is random access memory RAM that a computer microprocessor can access more quickly than it can access regular RAM. As the microprocessor processes data it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time consuming reading of data from larger memory. Therefore, a judicious data organization can save a lot of time in memory reading because of cache defaults.

2.9.2 Running time profiling

When trying to reduce the time required by a calculation, two questions have to be addressed :

- which are the most time consuming functions (procedures) of the program ?
- can one significantly reduce the amount of time spent therein ?

One way of knowing how much time is spent in the different steps of a computer program is to use a *profiler*. Most modern computers come with tools geared to this goal and the functionalities offered are twofold. First, the number of times a given block is called (typically a function) is reported. Second an estimate of the time spent within the block is given. This value is obtained either via a compiler directive or by sampling the program counter regularly. In the latter case, the desired value is obtained by calling (and loading) very often (usually several times per period) the internal clock, thus running the risk of distorting the measurement. Indeed, this estimate may not be very sound for functions whose unit call cost is much less than the sampling grain. Getting much better information can be done by running a system call when stepping in and out of a particular function to retrieve the system time, thus slowing down less dramatically the execution time and not altering the measurement as much.

Several strategies may be employed to optimize a piece of code. In practical terms, a set of simple rules can be followed. Before reviewing this briefly, we provide Table 2.1 that shows, with respect to the number of cycles, the total cost of the classical operations.

In this table, the computer architectures M_i are the following :

M_1 : *HP PA 7100* M_2 : *Sun HyperSparc*
M_3 : *DEC alpha 21064* M_4 : *Apollo 68040*
M_5 : *Intel Pentium MMX-based PC*

The simple analysis[7] of Table 2.1 shows that some operations must be avoided as far as possible. To this end, one has to find another way of implementing the desired functionality while asking the question about the pertinence of such an operation (say, for instance, a distance calculation d, if d^2 makes it possible to decide unambiguously, thus allowing to avoid the extra $\sqrt{\cdot}$ call. Similarly, comparing angles can be achieved by comparing their cosine values).

[7]The goal is not to compare such or such an architecture but to point out that significant differences exist between numerical operations.

-	M_1	M_2	M_3	M_4	M_5
$+, -, \times$	1	1	1	1	1
$/$	6	18	59	42	32
$\sqrt{\cdot}$	11	39	129	207	81
exp	58	163	156	685	175
arctan	66	120	173	242	140
log	69	158	125	361	143
sin, cos	89	521 - 532	265 - 281	293-307	115
arcsin,arccos	98	173 - 184	209 - 234	538-558	259 - 310
a^x	157	522	398	1,741	402
tan	168	563	343	298	166

Table 2.1: Number of elementary cycles for some classical operations on a range of computer architectures.

Following these remarks, we propose here some ideas to optimize a program. As will be seen later, this approach concerns high level as well as low level functionality, some of these operations being simply common sense.

- analyze the predicate likely to give the desired information. If several predicates can be used, pick the best one (in terms of its degree).
- if a predicate has a high degree, look for another formulation of the problem in which this predicate is no longer involved.
- examine the operations used and keep track of costly operations,
- minimize the number of parameters of a function,
- avoid indirections as far as possible (pointers or, even worse, pointers to pointers),
- use arrays if possible (take care of multi indices arrays or matrices with more than 3 indices, for example),
- avoid small loops[8] (typically a loop $i = 1, 2$ is not legitimate) and in the case of nested loops, use the most judicious implementation.
- etc.

To conclude and without pursuing this discussion further, notice that optimizing a program can lead to a less elegant or less formal implementation (for example, when a recursive call is replaced by a loop).

[8]Theoretically, compilers should be able to perform this task in most cases.

2.10 Classical application examples

For the sake of simplicity, in this section we consider triangular meshes only, although most of the constructions described can be extended (more or less easily) to other kinds of meshes. The following examples are given to emphasize how to benefit from algorithms and data structures described in the previous sections when dealing with applications related to mesh generation.

Therefore, numerous examples are linked to frequently encountered operations in various tasks in mesh generation or mesh modification algorithms. The order in which examples are given is not strictly significant. Some of these examples are purely academic, others deal with more real-world applications.

Remark 2.7 *The following examples can be seen as a set of exercises. Starting from data assumed to be known beforehand and depending on the goal envisaged, the reader is welcomed, on the one hand, to examine the proposed solution and, on the other hand, to look for alternate solutions to the same problem.*

2.10.1 Enumerating the ball of a given vertex (1)

Given a mesh and a vertex of this mesh, the *ball* of this vertex is the set of elements sharing the vertex. We propose here a method which, for any vertex of a mesh, provides the list of the elements in its ball. Our interest is motivated by the fact that vertex balls are commonly used in numerous parts of mesh generation or mesh optimization algorithms (see Chapter 18, for example). The proposed method works without the knowledge of the adjacency relationships between the elements (see above for a definition of what these relationships are and, below, how to obtain them).

Let ne be the number of triangles in the mesh and let $Tria(1:3, 1:ne)$ be the array that stores the vertex indices of the mesh elements. Let np be the number of mesh vertices[9]. The array $Tab(1:np)$ is initialized to the value -1. Now, in view of a further usage explained below, we fill the arrays Tab and $List$ (of length $3 \times ne$) as follows :

Algorithm 2.15 *Construction of the ball of the mesh vertices.*

Procedure PrepareBall(Tria)
$ij \leftarrow 1$
FOR $i = 1, ne$
  FOR $j = 1, 3$
    $s \leftarrow Tria(j, i)$
    $List(ij) \leftarrow Tab(s)$
    $Tab(s) \leftarrow ij$
    $ij \leftarrow ij + 1$
  END FOR j
END FOR i

[9] The points are assumed to be sequentially numbered from 1 to np (thus, in a connected way, if this last property is not satisfied, np must be the largest number (index) of a point).

It is now possible to easily generate the indices of all the elements sharing a given vertex. Let P be the index of the considered vertex, then its ball is obtained as follows :

Algorithm 2.16 *Enumerating the ball of a vertex.*

Procedure BallPoint1(P)
$ij \leftarrow Tab(P)$
IF $ij \neq -1$ THEN
  $i = \frac{ij}{3} + 1$,
  (vertex P is the vertex of element i),
  $j \leftarrow ij - 3 \times (i-1) + 1$,
  (vertex P is the vertex of index j in element i),
  $ij \leftarrow List(ij)$ and back to IF.
ELSE END.

On completion of this procedure, the different indices i obtained in the algorithm are the indices of the elements[10] in the ball of the point P used as entry point while for each triangle of index i, the index j gives the position of point P.

Note that the above method consists of two algorithms. The first one is a preparation step which constructs the relevant tables. Once that has been done, the second one can be used repeatedly to access the ball of any vertex in the mesh.

2.10.2 Enumerating the ball of a given vertex (2)

Here, we consider a similar problem but now only one ball is of interest (*i.e.*, we consider only one vertex P) and, in addition, we assume that the neighboring relationships are available (in an example below, see how to compute this information).

Given a triangle, its three neighbors are given via a table $Neigh(1:3, 1:ne)$ (where ne is the number of triangles). Indeed,

$$k = Neigh(j, i)$$

means that element k is adjacent to element i and edge j of element i is the shared edge (while $k = 0$ if edge j of element i is a boundary edge). Also we assume that vertex j of triangle i is opposite edge j of this triangle (see Chapter 1).

Now, let k_0 be a triangle having a vertex P, the following algorithm computes the indices of the elements in the ball of P (we assume that P is not the index of a boundary vertex) :

Algorithm 2.17 *Enumerating the ball of a vertex.*

Procedure BallPoint2(P)
$k \leftarrow k_0$, $ltab \leftarrow 0$,
REPEAT

[10] In what follows, depending on the context, we will not differentiate between the index of an entity and this entity itself. For instance, point P and point of index P must be considered as two possible expressions of the same notion. Similarly, element k is the element of index k.

$ltab \leftarrow ltab + 1$,
$tab(ltab) \leftarrow k$
take j the index of P in triangle k,
take j_{next} the index following index j,
$k \leftarrow Neigh(j_{next}, k)$,
WHILE $k \neq k_0$.

On completion, *ltab* is the number of triangles in the ball of vertex P and the indices of the desired triangles are the k's in the array *tab*.

Exercise 2.19 *Examine the case where the vertex P in question is a boundary vertex. Modify the above scheme accordingly. (Hint : take care of the case where $Neigh(j, i) = 0$).*

Notice that the proposed scheme does not extend to solving the same problem when a tetrahedral mesh is considered, where a more subtle algorithm must be defined.

Exercise 2.20 *Construct the ball of a vertex using the adjacency matrices described in Section 2.7.2.*

2.10.3 Searching operations

The problem is to find the item (the box, the cell or again the element) of a structure (a grid, a quadtree or an arbitrary mesh) within which a given point falls.

Such problems are so-called searching problems or localization problems or again point location problems and are fundamental for various mesh generation methods. Let x, y be the coordinates of the given point.

Searching in a grid. Using a grid (Section 2.6.1) is a source of simplification by many respects. First, the indices of a box containing a point can be computed trivially. Second, it is easy to have access to the neighborhood of a given box and to that of a given point.

Let Δ_x (resp. Δ_y) be the size of the grid box in direction x (resp. y), the grid being constructed (see above) with the point x_0, y_0 as left bottom corner. Then

$$ind_x = \left\lfloor \frac{x - x_0}{\Delta_x} \right\rfloor \quad \text{and} \quad ind_y = \left\lfloor \frac{y - y_0}{\Delta_y} \right\rfloor$$

are the two indices of the box containing the point. Actually, ind_x as well as ind_y are integer values while the point coordinates could be floating-point values. Depending on the information stored in the grid, these indices can be used for various purposes (for instance, to find a point close to the point considered, any point in the box being a candidate, or, in the case of an empty box, any point in a non-empty box found in a certain neighbourhood of the initial box).

Searching in a quadtree. The easiest way to locate the quadtree cell (a PR-quadtree here according to Section 2.6.2) containing a given point is to start from the root of the tree and to use the values of the coordinates of the center of this cell to determine which one of the four children contains the given point. The center of a cell is easily obtained based on the box indices, we have :

$$x_c = ind_x\, \Delta_x + \frac{\Delta_x}{2} \quad \text{and} \quad y_c = ind_y\, \Delta_y + \frac{\Delta_y}{2},$$

where x_c, y_c are the coordinates of this center.

The process is then recursively performed until a leaf (a terminal node) is reached.

An alternative approach is based on the underlying binary encoding of the quadtree by which a cell can be defined by an index consisting of a series of 0 and 1. The root is the 0 cell while the four first children can be identified by the following indices (Chapter 5) :

$$(00, 01, 11, 10)$$

where 00 is the bottom left cell, 01 is the cell on the right of the previous one, 11 is the top right cell and 10 is the cell on the left of the previous (it is also the cell on top of the 00 cell). Actually, adding 01 to an index enables us to go to the cell on the right while adding 10 at the current index leads to the cell top of the initial cell (at the lower level, this effect will be obtained when adding 0001 and 0010 respectively, and so on).

Thus, binary operations can be used to locate a given point when a suitable system of coordinates has been defined.

Searching in a mesh. In this case, we assume that we are given a triangular mesh $\mathcal{T}$ covering a convex (planar) domain (for the sake of simplicity we consider this simple case only) and we want to find which element in $\mathcal{T}$ contains point P.

Let K be a triangle in $\mathcal{T}$ and let V_1, V_2, V_3 be its three vertices whose coordinates are denoted by x_i and y_i, $(i = 1, 3)$, then the signed surface area of K is :

$$S_K = \frac{1}{2} \begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix}. \tag{2.3}$$

Actually, we can define S_K as twice the above value so as to avoid a division (notice that, due to the numbering convention of Chapter 1, S_K is strictly positive if K is a valid element).

Let us define the virtual triangle K^j as the triangle K where the vertex V_j of K is replaced by the point P considered. Then, we can compute S_{K^j}, $(j = 1, 3)$ whose sign enables us to determine[11] where the point P is located with respect to the half-planes bounded by the lines supporting the three edges of K (note that 7 regions are defined in this way). According to the sign of S_{K^j}, we pass through the corresponding neighbor of K and we repeat this process until the three S_{K^j}'s

[11] We meet again here the barycentric coordinates.

are positive (assuming that P is distinct from all the mesh vertices) meaning that the visited triangle contains P.

Based on these observations, a searching algorithm is easy to design and implement. One has to select a triangle K_0 in the mesh and then follow the above scheme.

Rapid searching procedure in an arbitrary mesh. The previous algorithm can be very time consuming if a large number of elements needs to be visited between triangle K_0, the initial guess, and the solution triangle. This could lead in fact to a large number of area computations. Therefore, this algorithm could be combined with a grid (or a tree-like structure). A grid, or a tree-like structure enclosing the mesh is constructed and, for each cell one mesh vertex contained in it, if any, is recorded. In our previous examples, we showed that it is easy to find the cell of a grid or tree containing the given point. Also, a mesh element is associated with every point recorded in the cells. Hence, we can associate the given point with a close mesh point in the same cell. Any element, K_0, having this mesh point as a vertex can be used as an initial guess for the above searching procedure. In this way, the number of visited triangles is reduced and the number of necessary computations is reduced as well.

Remark 2.8 *Note that the grid (the tree structure) could be defined in various ways depending on the nature of the dataset. In this respect, for a grid, the number of boxes (indeed the values Δ_x and Δ_y as introduced above) and thus the occupation of the boxes are parameters that clearly affect the efficiency of the whole process.*

Intersection of a line segment with the elements of a mesh. Intersection problems are important components for some mesh generation techniques. One such problem is the following : given a mesh and a line segment between any two mesh vertices, construct the list of elements that are intersected by the given line segment.

Exercise 2.21 *Modify the searching technique in a mesh given above to solve this problem.*

2.10.4 Enumerating the set of edges in a mesh

In this section, we describe several examples of methods for creating the list of the edges in a mesh. Let na be the number of edges, which we will label from 1 to na in the process of building the lists. In general, na is not known beforehand, so that in practice, an upper bound $namax$ for na is needed to allocate memory resources for the arrays used.

An elementary method. Let ne be the number of elements in the mesh and let $Tab(1:2, 1:namax)$ be the table used to store all the edges. The following procedure enables us to fill the table Tab :

Algorithm 2.18 *Enumerating the edges in a mesh.*

Procedure TableEdge1()
$na \leftarrow 0$,
FOR $i = 1, ne$
  FOR $j = 1, 3$
    let e_1, e_2 be the indices of the endpoints of edge j of element i,
    $k \leftarrow 1$ and IF $found$ is a boolean, set $found = .FALSE.$,
    WHILE ($found = .FALSE.$ AND $k < na + 1$)
      IF [($e_1 = Tab(1,k)$ AND $e_2 = Tab(2,k)$) OR
      ($e_2 = Tab(1,k)$ AND $e_1 = Tab(2,k)$)], THEN $found = .TRUE.$
      ELSE $k \leftarrow k + 1$
      END IF
    END WHILE
    IF $found = .FALSE.$, the edge considered is a new one, THEn
      $na \leftarrow na + 1$ AND $TAB(1, na) \leftarrow e_1$, $TAB(2, na) \leftarrow e_2$,
    END IF
  END FOR j
END FOR i

On completion, na is the number of edges and Tab contains the mesh edges (in fact, the indices of the edge endpoints). Notice that the number of times the comparisons are performed in the inner loop is proportional to $ne \times na$. This method is very time consuming in terms of complexity; however, it can handle a non-manifold surface mesh in $\mathbb{R}^3$ (a mesh is said to be manifold if all of its edges are shared by exactly two triangles or belong to the boundary, see Chapter 1) without any changes.

Using an edge coloring scheme. In this example, we build the same table using a technique of edge coloring to ensure that every mesh edge is recorded only once. We assume that the mesh is manifold (if a surface mesh is considered) and that we can access the neighboring elements across each internal edge, *i.e.,* we have constructed a list $Neigh(j, i)$ which is the index of the neighbor of element i on side j. (See the next section). Let $ColorTab(1:3, 1:ne)$ be a table which stores a color value (0 or 1) for edge j of element i in $ColorTab(j, i)$. Then we can build $Tab(1:2, 1:na)$ as in the first example by :

Algorithm 2.19 *Edge coloring scheme.*

Procedure TableEdge2()
$na \leftarrow 0$,
FOR $i = 1, ne$
  FOR $j = 1, 3$
    let e_1, e_2 be the indices of the endpoints of edge j in element i,
    IF $ColorTab(j, i) = 0$ THEN, $na \leftarrow na + 1$,
    set $Tab(1, na) = e_1$, $Tab(2, na) = e_2$ and $ColorTab(j, i) = 1$,
    $k = Neigh(j, i)$, let j_k be the index of this edge in element k,
    set $ColorTab(j_k, k) = 1$, IF $k \neq 0$.

```
      ELSE, edge j of element i has already been visited.
      END IF
    END FOR j
  END FOR i
```

On completion, na is the number of edges and Tab contains the edges. The outer pair of loops of this method and the first one are the same. But the inner loop of this method is only executed $3 \times ne$ times. We reduced the amount of computation by using the large temporary table $ColorTab$. Note that this algorithm, serving as an example of *static* coloring, requires the input of the neighborhood relationships between the elements so as to know, for a given element, its (one, two or) three neighbors.

Notice also that this algorithm does not extend to three dimensions (as the coloring a vertex does not identify an edge (while a similar property holds for a face)).

By hashing. In the two previous examples, we constructed arrays for the edges of a graph which in fact consists in labeling each edge with a number. The arrays give a direct access to the endpoints of the edge from the edge number. But to determine the number of an edge from its endpoints, you have to search that table. In this example, we build a set of lists that provide the opposite access to edge data, *i.e.*, we construct a list that allows a direct access to the number of an edge from the edge endpoints. This construction is an example of *hashing* (Section 2.5.2). It works even in the non-manifold case where an edge is shared by more than two elements.

Let ne be the number of elements, and e_1 and e_2 be the endpoints of an edge a. We assume that we have a table $Sum(1 : 2 * np)$ where np is the number of vertices[12] in the mesh, a table $Link(1 : namax)$ with $namax > na$ the number of edges in the mesh[13] and a table $Min(1 : namax)$.

We first give the construction, then we add some comments. The construction consists in (after initializing all the arrays to 0) :

Algorithm 2.20 *Construction of the edges of a mesh.*

```
Procedure TableEdge3()
na ← 0
FOR i = 1, ne
  FOR j = 1, 3
    compute s = e1 + e2,
    IF Sum(s) = 0, na ← na + 1, Sum(s) ← na, Min(na) ← min(e1, e2),
    ELSE IF Min(l) ≠ min(e1, e2) with l = Sum(s), THEN
      (A) IF Link(l) = 0 THEN
      na ← na + 1, Link(l) ← na and Min(na) ← min(e1, e2),
      ELSE, consider m = Link(l),
```

[12] See the previous note about np.

[13] See the previous note about $namax$.


```
        IF Min(m) ≠ min(e1, e2), THEN set l = m and back to (A).
      END IF
    END FOR j
  END FOR i
```

As a result, na is the number of edges in the triangulation.

More precisely, for each element edge, we compute an index s as the sum of its two endpoint numbers giving an entry point in the table Sum. A zero value for $Sum(s)$ means that the current edge must be considered as a new edge (thus, it could be stored or processed as desired). Otherwise, one or several edge(s) with the same sum index have already been encountered. Hence, we just have to check if any of these edges matches the current edge (thanks to $Link$). This list traversal is done until the current edge is found (thanks to Min). Actually, if it is found, we proceed to the next edge, if not, it is inserted at the next available entry in $Link$.

Based on this construction, the edges can be retrieved using the following procedure :

Algorithm 2.21 *Retrieving the mesh edges.*

```
Procedure RetrieveEdge()
na ← 0
FOR s = 1, 2 × np
  IF k = Sum(s) ≠ 0, we find an edge such that e1 + e2 = s and
  min(e1, e2) = Min(k) and, while scanning the array Link we find
  all the edges having the same sum of indices s.
  Practically, these edges can be obtained as follows
    na ← na + 1, the pair s − Min(k), Min(k) is the edge na,
    WHILE l = Link(k) ≠ 0
    na ← na + 1,
    k ← l and the pair s − Min(l), Min(l) is the edge na.
    END WHILE
  END IF
END FOR s
```

Many variations of this example can be obtained by modifying the keys of the hashing (replacing the Sum and the Min by different encoding schemes) or by modifying the purpose of the algorithm. For instance, it is possible to obtain the list of the boundary edges. Note that different choices of hashing function lead to different numbers of collisions (the number of edges with the same key) which could dramatically affect the efficiency of the method.

Exercise 2.22 *Analyze how this technique could be used to improve the efficiency of the elementary method in the first example.*

2.10.5 About set membership

The question is here to decide (quickly) whether an edge is a member of a set of edges stored in an array.

Depending on how the edge table is constructed (see the examples discussing how to construct this table in the previous section), finding if a given edge is a member of this table can be efficiently solved provided a suitable data structure is used (conversely, a less suitable data structure leads to a time consuming method).

In practical terms, if a hashing technique has been used to establish the edge table (see the above procedure), checking whether an edge is a member of this table is easy. Let e_1, e_2 be the two indices of the edge, then :

Algorithm 2.22 *Check the existence of an edge e_1, e_2 in a mesh.*

Procedure ExistEdge(e_1, e_2)
compute $s = e_1 + e_2$,
IF $k = Sum(s) \neq 0$, one or more edges
such that $e_1 + e_2 = s$ exist.
IF $Min(k) = min(e_1, e_2)$, then edge e_1, e_2 belongs to the table.
ElSE, scan the table $Link$ to find
all edges having the same sum of indices s :
WHILE $l = Link(k) \neq 0$, $k \leftarrow l$ and analyze $Min(k)$.
ELSE, the edge is not stored.
END IF

provides the correct answer. Notice that the efficiency with which the question is answered depends on the way the edge table has been constructed.

2.10.6 Constructing the neighborhood relationships

We consider again a triangular mesh and we would like to construct the neighborhood relationships from element to element. Let $Neigh(1:3, 1:ne)$ be this table, then

$$k = Neigh(j, i)$$

means that element k is adjacent to element i and edge j of element i is the shared edge (while $k = 0$ if edge j of element i is a boundary edge).

To construct this table, a variation of the algorithm previously employed to construct the edge table can be used where additional information is associated with the edges at the time they are visited. In this respect, it is necessary, for a given edge, to know the element of which it is a member and to know its index in its element.

The first time an edge is visited, for element i and index j, these two values are stored. When, say for element I at index J, the edge is met again then, the two following relationships are completed :

$$Neigh(j, i) = I \quad \text{and} \quad Neigh(J, I) = i\,.$$

Exercise 2.23 *Discuss the non-manifold case (for a surface) where more than two triangles share a given edge.*

2.10.7 Static and dynamic coloring

In a previous procedure, we showed one application of static coloring applied to some items of a mesh with the purpose of deciding quickly whether such or such a situation occurs.

In this case, such a tool can be seen as a boolean operator where the status of an item is defined as $.TRUE.$ or $.FALSE.$ (or 0 or 1) depending on the situation. In some cases, a two-flag operator is inefficient and *dynamic* coloring can be used more effectively.

Let us give a very simple example. We would like to construct balls about each vertex using a technique like the second enumeration method for balls discussed earlier in this chapter. For this purpose, let $List_i$ be a list of the indices of elements in the ball around the vertex of index i, and let $ColorTab(1:ne)$ be a color table for the elements of the mesh. An initial solution (np being the number of internal vertices) can be as follows :

Algorithm 2.23 *Construction of the ball of points.*

Procedure BallPoint3()
FOR $j = 1, ne$
$ColorTab(j) \leftarrow 0$
END FOR j
FOR $i = 1, np$
define $List_i$ as the empty list
find an element of index k which belongs to the ball of vertex i
store k into $List_i$ and set $ColorTab(k) = 1$
WHILE element k has a neighbor of index j that has the vertex i
AND that $ColorTab(j) = 0$
$k \leftarrow j$; $ColorTab(k) = 1$;
END WHILE
FOR $j = 1, ne$
$ColorTab(j) \leftarrow 0$
END FOR j
END FOR i

This method uses two colors which must be maintained, which in turn requires additional processing of $ColorTab$ to reset its entries input to zero. One way to avoid this would be to maintain a list of the element indices that were encountered for the current i value and use it to reinitialize the relevant $ColorTab$ entries to 0. Another way is to use *dynamic* coloring method using colouring values 1 through np as follows :

Algorithm 2.24 *Construction of the ball of points.*

Procedure BallPoint4()
FOR $j = 1, ne$
$ColorTab(j) \leftarrow 0$
END FOR j
FOR $i = 1, np$
define $List_i$ as the empty list

```
    find an element of index k that belongs to the ball of vertex i
    store k in List_i and set ColorTab(k) = i
    WHILE element k has a neighbor of index j having the vertex i
    AND that ColorTab(j) < i
       k ← j; ColorTab(k) ← i;
    END WHILE
END FOR i
```

Here the vertex label i is used as a dynamic color code to simplify the management of the element status which is automatically updated without any explicit processing.

2.10.8 About the construction of a dichotomy

The dichotomy approach, illustrated by a small example here, is a useful approach to a variety of meshing problems. This example involves a partition of an interval $a < x < b$ which is a set of points x_i for $i = 1...N$ such that $a = x_1$, $b = x_N$ and $x_i < x_{i+1}$. Suppose that we are given a continuous function $f(x)$ defined on the interval $a < x < b$ and a tolerance value τ. We wish to construct a partition of this interval such that $|f(x_{i+1}) - f(x_i)| \leq \tau$ for each sub-interval.

The following method uses the dichotomy approach to build tables Tab and $Next$ which store the desired partition.

Algorithm 2.25 *Construction of a dichotomy.*

```
Procedure IntervalDichotomy()
Tab(1) ← a , Tab(2) ← b
i ← 1, ltab ← 2, Next(1) ← 2
REPEAT
   x = Tab(i) ; y = Tab(Next(i))
   IF |f(y) − f(x)| > ε,
   THEN ltab ← ltab + 1, Tab(ltab) ← (x+y)/2
   Next(ltab) ← Next(i), Next(i) ← ltab
   ELSE i ← Next(i)
UNTIL i = 2.
```

Note that this kind of algorithm has various applications and, in some sense, can emulate a recursive process (whereas this capability is not necessarily included in some programming languages).

Concluding remark

In Section 2.10, we have discussed a few examples of algorithms to illustrate how to use such or such basic structures or basic algorithms. Obviously, numerous other application examples can be found and, actually, meshing algorithms as well as mesh optimization algorithms or, in general, mesh manipulation algorithms or even mesh visualization algorithms can take advantage of using such or such basic ingredients in order to easily find or process the mesh entities that are involved in the whole procedure.

Chapter 3

A *comprehensive* survey of mesh generation methods

Introduction

Mesh generation has evolved rapidly over the last two decades and meshing techniques seem recently to have reached a level of maturity that allows them to calculate complete solutions to complex three-dimensional problems. Thus, unstructured meshes for complex three-dimensional domains of arbitrary shape can be completed on current workstations in reasonable time. Further improvements may still be expected, for instance regarding the robustness, reliability and optimality of the meshing techniques.

Early mesh generation methods employed meshes consisting of quadrilaterals in two dimensions or hexahedra in three dimensions. Each vertex of such meshes can be readily defined as an array of indices and these types of meshes are commonly referred to as *structured* meshes. By extension, any mesh having a high degree of ordering (for example a Cartesian grid) is said to be structured. More recent developments have tried to cope with the complex geometries (for instance in C.A.D. models involving multiple bounding surfaces) that were difficult to handle (*i.e.*, to mesh) with fully structured meshes. Nowadays unstructured meshes can be associated with finite element methods to provide an efficient alternative to structured meshes.

★
★ ★

The purpose of this chapter is to provide a comprehensive overview of the current techniques for both structured and unstructured mesh generation and to discuss their intrinsic advantages or weaknesses. These techniques will be further discussed in more detail in the relevant chapters of this book. First, a preliminary classification of existing meshing techniques is proposed. One section is dedicated to surface meshing as surfaces play an important role in unstructured mesh generation techniques. Finally, a brief outline of mesh adaptation approaches is given.

3.1 Classes of methods

Despite many conceptual differences (since mesh generation methods have been developed in different contexts and were aimed at different field of applications), the classification of these techniques into seven classes has been proposed, for instance, in [George-1991]. Although this classification reflects the main approaches published, it appears that several techniques can be gathered together (due to their intrinsic properties), thus leading to a modified classification into only five categories :

Class 1. *manual* or *semi-automatic* methods.

They are applicable to geometrically simple domains. Enumerative methods (mesh entities are explicitly user-supplied) and explicit methods (which take advantage of the geometric features of the domain) are representative of this class.

Class 2. *parameterization* (mapping) methods.

The final mesh is the result of the inverse transformation *mapping* of a regular lattice of points in a parametric space to the physical space. Two main approaches belong to this class, depending on whether the mapping function is implicitly or explicitly defined:

- *algebraic interpolation* methods. The mesh is obtained using a transfinite interpolation from boundary curves (surfaces) or other related techniques explicitly defined,
- *solution-based* methods. The mesh is generated based on the numerical solution of a partial differential system of equations (elliptic, hyperbolic or parabolic), thus relying on an analytically defined function.

Class 3. *domain decomposition* methods.

The mesh is the result of a top-down analysis that consists in splitting the domain to be meshed into smaller domains, geometrically close to a domain of reference (in terms of shape). Two main approaches have been proposed, the difference being the structured or unstructured nature of the mesh used to cover the small domains :

- *block decomposition* methods: the domain is decomposed into several simpler sub-domains (blocks), each of which is then covered with a structured mesh (obtained for instance using a mapping technique, as seen above).
- *spatial decomposition* methods: the domain is approximated with a union of disjoint cells that are subdivided to cover a spatial region object, each cell then being further decomposed into mesh elements. Quadtree and octree-based techniques are representative of this class.

Class 4. *point-insertion / element creation* methods.

The related methods generally start from a discretization of the boundary of the domain (although this feature is not strictly required) and mainly consist in creating and inserting internal nodes (elements) in the domain. Advancing-front (element creation) and Delaunay-based (point insertion) approaches are two methods belonging to this class.

Class 5. *constructive* methods.

The final mesh of the domain is the result of merging several meshes using topological or geometric transformations, each of these meshes being created by any of the previous methods.

Remark 3.1 *Needless to say, this classification is necessarily arbitrary. However, while not unique, it does account for the different approaches published. Other methods not included in this classification exist, which are designed to handle specific situations.*

A difficult task consists of clearly identifying the method capable of providing an adequate mesh, depending on the field of application. Basically, the geometry of the domain and the physical problem direct the user towards which method to apply.

On the other hand, the emphasis can be put on the type of meshes created by any of the proposed methods. From this point of view, two classes of meshing techniques can be identified, depending on whether they lead to structured or unstructured meshes. The following sections provide additional details on this aspect.

3.2 Structured mesh generators

In this section, we briefly describe the main approaches generally used to create structured meshes. While not claiming exhaustivity, the techniques mentioned here are representative of the current and latest developments in this field.

The basic idea common to all structured mesh generation methods consists of meshing a canonical domain (*i.e.*, a simple geometry) and mapping this mesh to a physical domain defined by its boundary discretization. Numerous types of such transformations exist and have been successfully applied to computational domains, for instance parametric space for surfaces (Bézier patches, B-splines), Lagrange or transfinite interpolation formula, quasi-conformal transformations, etc.

The first problem to solve is *where* to place the mesh points in such a way as to achieve a *natural* ordering appropriate to the problem considered. A trivial observation shows that simple domains such as squares and discs, in two dimensions, have an intrinsic curvilinear coordinate system. In this sense, the mapping techniques described below provide a basis for mesh generation.

Curvilinear coordinates. The physical domain discretization requires some level of organization to efficiently compute the solution of the P.D.E.s. This organization is usually provided through a Cartesian or cylindrical coordinate system. More precisely, the grid points are defined using coordinate line intersections,

which allow all numerical computations to be performed in a fixed (square or rectangular) grid. Hence, the Cartesian coordinates used to represent the P.D.E.s have been replaced by the curvilinear coordinates.[1] A constant value of one curvilinear coordinate (and a monotonic variation of the other) in the physical space corresponds to vertical or horizontal lines in the *logical* space.[2]

Theoretically, two procedures can be used to generate a system of curvilinear coordinates, algebraic interpolation techniques [Gordon,Hall-1973] and solution-based techniques [Thompson-1982a]. From the computational point of view, the classical algebraic method is usually faster than the differential equation methods.

3.2.1 Algebraic interpolation methods

A simple, though efficient, way to achieve a structured mesh is to use a sequence of mappings to reduce the possibly complex domain to simple generic shapes (e.g. a triangle, a quadrilateral, a hexahedron, etc.). After a structured mesh has been defined in the logical space, the mapping function is used to generate a mesh conforming to all domain boundaries. This technique has proved useful for two dimensional domains as well as in three dimensions.

The mapping function(s) and the mesh point distribution in the logical space can be chosen arbitrarily. However, it may be of some interest (and sometimes more efficient) to force the boundary discretization in the logical space to match the given domain boundary discretization.[3] The control of the mesh point distribution in the parametric space makes it possible to control the density of mesh vertices in the real domain (for instance to obtain a finer mesh in regions of high curvature).

Remark 3.2 *In general, the domain discretization must be a convex polygon (polyhedron, in three dimensions)*[4] *to guarantee the validity and the conformity (according to Definition 1.7) of the resulting mesh.*

The problem of finding a proper mapping function is equivalent to finding a specific function of the curvilinear coordinates. This function contains coefficients that enable the function to match specific values of the Cartesian coordinates on the boundary (and possibly elsewhere). Algebraic grid generation is thoroughly discussed in [Shmit-1982] and [Eriksson-1982]. Figure 3.1 shows an example of an algebraic mesh generated by the method described in [Baker-1991b].

To emphasize the algebraic method feature, we simply mention one particular mapping function, the transfinite interpolation scheme. This approach was first investigated by [Gordon,Hall-1973] and, then, by [Eriksson-1983], among others. Its most significant feature is its ability to control the mesh point distribution and particularly the slope of the mesh lines meeting the boundary surfaces [Baker-1989a]. In this chapter, we do not pursue the notion of transfinite elements and refer the reader to Chapter 4. However, we describe its application to the mapping of a unit square, in two dimensions.

[1] Note: the mapping of the physical space onto the *logical* space must be one-to-one.
[2] Also called transformed or parametric space.
[3] This feature makes it possible to conform exactly to the given domain boundaries.
[4] or at least close to a convex shape.

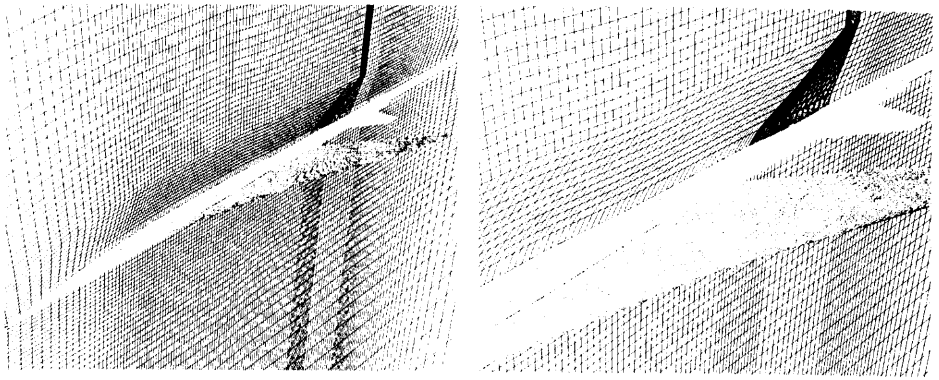

Figure 3.1: *Single block algebraic grid for a fuselage plus two lifting surfaces (data courtesy T.J. Baker, Princeton University).*

Unit square mapping by transfinite interpolation. Here, we are concerned with a continuous transformation which maps the unit square $(\xi, \eta) \in [0,1] \times [0,1]$ one-to-one onto a simply connected, bounded two-dimensional domain. The mapping can be seen as a topological distortion of the square into the domain. The problem is to construct the mapping function that matches the boundary of the domain, and more precisely, the boundary discretization of this domain.

Let $\phi_i(\xi, \eta), i = 1, 4$ be the parameterization of the side i of the real domain, for which four such sides have been identified, and let a_i be the corresponding edge endpoints (corners). For the sake of simplicity, we have assumed that the discretizations of any two opposite edges of the domain have the same number of points[5]. A discretization of the unit square is constructed, analogous to that of the real domain (*i.e.*, each side of the square conforms to the discretization of the corresponding real side, in terms of relative distances between successive points). A quadrilateral mesh is then formed in the logical space by joining the matching points on opposite edges, the internal nodes thus being the line intersections.

Then, the mapping function takes the lattice of points in the parametric space (unit square) and maps it to the physical space (real domain) using transfinite interpolation based on the Lagrange interpolation formula as follows:

$$\begin{aligned} F(\xi,\eta) &= (1-\eta)\phi_1(\xi) + \xi\phi_2(\eta) + \eta\phi_3(\xi) + (1-\xi)\phi_4(\eta) \\ &- ((1-\xi)(1-\eta)a_1 + \xi(1-\eta)a_2 + \xi\eta a_3 + (1-\xi)\eta a_4). \end{aligned} \tag{3.1}$$

Figure 3.2 shows a mesh of a domain mapped by applying a transfinite interpolation formula.

The same technique can be applied to map a right triangle onto a triangular-shaped domain and can be extended to three dimensions as well (cf. Chapter 4).

[5] One can always obtain such a situation by adding more points along a boundary edge, if needed.

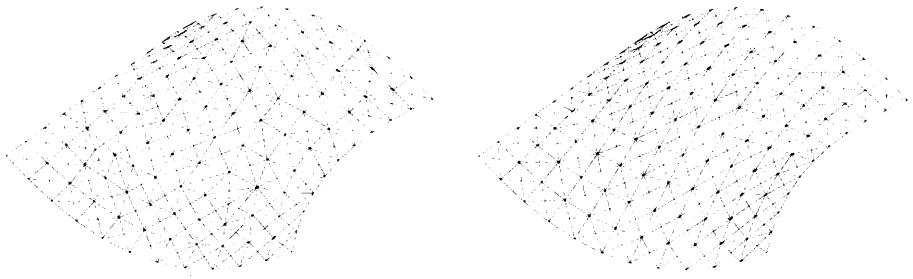

Figure 3.2: *Surface mesh obtained using a transfinite mapping of the unit square (right: with element shape control.)*

Remark 3.3 (Sequential mappings) *Another algebraic method which is similar to the coordinate transformation method introduced above is based on the combination of a sequence of mappings. It becomes possible to reduce a complex domain to a simple generic shape by introducing several simple conformal mappings successively.*

Remark 3.4 (Blending approach) *The association of several meshes, each one being generated separately as a simple domain, to form a global smooth mesh using a weighted combination of functions is the attractive key feature of the blending grid technique. Although very promising, this approach is by no means easy to implement for arbitrary complex configurations. To some extent, this technique prefigures the multiblock approach.*

3.2.2 P.D.E.-based methods

Since the problems of concern are formulated in terms of partial differential equations, it seems obvious to link the coordinates to the solutions of a system of P.D.E.s. If the coordinates vertices are specified on the boundary of the region, the equations must be elliptic, as they would be parabolic or hyperbolic if the specification concerned only part of the domain boundary. Hence, the important step of finding a mapping between a Cartesian and a boundary-fitted curvilinear coordinate system is to clearly identify the equations. Of this kind of technique, the elliptical equation method is the most popular (cf. [Eiseman,Erlebacher-1987] for a general survey of P.D.E.-based methods).

Elliptic method. The main advantage of the elliptic equation method is that it preserves grid orthogonality in the vicinity of the boundary. Another property is the inherent smoothness over the entire domain that prevails in the solution of

elliptic problems. Moreover, boundary discontinuities do not propagate far inside the domain. A drawback is that the coordinate system is the solution of a system of partial differential equations, thus resulting in more computing time than other methods of generating a grid. This technique has provided some interesting results, especially in computational fluid dynamics (CFD) simulations for transonic flow over airfoils [Thompson-1982b], [Thompson-1987].

The most simple elliptic system is the Laplace system defined as

$$\nabla^2 \xi^i = 0 \qquad i = 1,3 \tag{3.2}$$

which can be obtained from the Euler equations for the minimization of the integral

$$I = \iiint \sum_{i=1}^{3} |\nabla \xi^i|^2 dV$$

where the quantity $|\nabla \xi^i|$ represents in a certain way the grid point density along the coordinate line for a variation of ξ^i ($\xi^1 = \xi$, $\xi^2 = \eta$, ...). The smoothing effect of the Laplacian tends to closely or equally space the lines according to the boundary curvature.

Remark 3.5 *The strong smoothing effect of the Laplace transform may lead to an undesirable node point distribution. To overcome this problem, control functions can be introduced in Equation 3.2.*

Parabolic and hyperbolic methods. The mesh generation procedure can also be based on parabolic or hyperbolic P.D.E.s. Equations of the parabolic method can be derived from the elliptical method by modifying the proper terms. The parabolic technique is useful to generate a mesh between two boundaries of a multi-connected domain, with two boundary specifications. The hyperbolic approach tolerates only one boundary specification and is mainly interesting for use in calculations over unbounded domains or for generating orthogonal meshes.

For instance, the solutions of the following set of equations

$$x_\xi x_\eta + y_\xi y_\eta = 0\,, \qquad x_\xi y_\eta - x_\eta y_\xi = V(\xi,\eta)\,,$$

where $x_\xi = \frac{\partial x}{\partial \xi}$, defines a hyperbolic system [Steger,Sorenson-1980], where the first equation corresponds to the orthogonality condition and the second equation defines the local cell area based on a specified distribution $V(\xi,\eta)$. The main drawback of the hyperbolic-type approach is its inability to create meshes for internal flow problems.

3.2.3 Multiblock method

Wrapping a mesh around a complex domain boundary may be a tough and even intractable problem to solve. One way of getting around this difficulty is to consider a multiblock scheme. In this approach, the whole domain is decomposed into several simpler sub-domains (*i.e., blocks*), each of which is then covered automatically with a structured mesh, resulting, for instance, from an algebraic technique

or a P.D.E. methods. This feature makes the multiblock approach particularly interesting for parallel computations.

Remark 3.6 *In general, the resulting mesh is structured at the block level but no longer at the global domain level. Therefore, this approach usually leads to unstructured meshes.*

Several possibilities of implementing the multiblock technique arise depending on which constraints are required between the blocks (for instance what degree of continuity or conformity is desired).

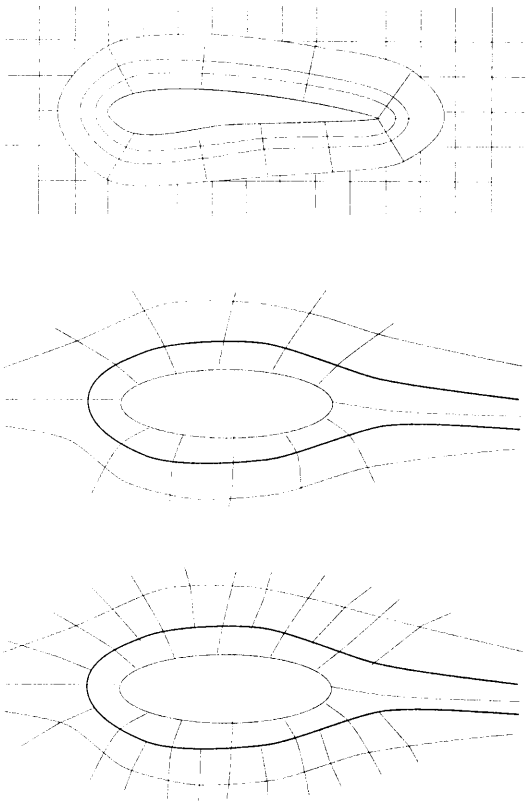

Figure 3.3: *Several implementations of the multiblock approach : overlapping (top), composite (center, mesh lines are continuous across the boundary) and patched (bottom, conforming surfaces of the boundaries, however discontinuous mesh lines).*

Overlapping. If no special attention is paid to the block interfaces, each block can be meshed separately for each component of the domain. The resulting mesh is a system of overlapping sub-meshes. Although the meshes are easy to generate, the main drawbacks of the technique are the transfer of information between neighboring meshes and the accuracy of the interpolation which can prevent the stability of the method.

Patched. An additional constraint to the overlapping multiblock technique requires the separate meshes to conform to the surfaces of their common boundaries, even if mesh lines are not continuous. In this patched approach, the interpolation procedure is easier than that required by the previous approach and mesh refinement can vary in specific regions without propagating elsewhere.

Composite. If mesh lines are required to be continuous across the boundaries, thus propagating mesh refinement throughout the entire domain, we obtain the composite multiblock method. This technique requires a global vertex numbering. Its main advantage is the improved accuracy that results.

General scheme. The composite multiblock decomposition method can be summarized as follows :

Step 1. Decompose the computational domain into simple blocks.

- Use a global vertex numbering
- Define the block interfaces to ensure conformity

Step 2. Discretize the block interfaces. The requirements are to

- obtain good geometric approximation
- ensure the mesh lines are continuous across the boundaries
- ensure the adequacy of each block with respect to the local mesh generation process (for instance, with an algebraic method, the number of points on opposite edges must be identical),

Step 3. Mesh each block separately (create internal points)

Step 4. Construct the final mesh by merging the sub-meshes.

3.2.4 Product method (topology-based method)

Semi-automated procedures sometimes give additional meshing capabilities to the user. Actually, the mesh of a complex domain of cylindrical topology in d-dimensions can be easily obtained from a $d-1$-dimensional mesh of a section, the *source* (for instance, a cylinder can be defined using a circle and a direction in three dimensions). More precisely, a point leads to a series of segments and a segment results in a set of quadrilaterals. The efficiency of the method is related to the ability of the user to define the reference mesh.

In three dimensions, the purpose of the two-dimensional reference mesh is to provide a pattern from which to extrude the final mesh. The number of layers (slices) and their positions (*i.e.*, the node locations along the reference line) can

be supplied implicitly (the discretization of the line is given) or explicitly (using a stretching function for instance). The reference mesh is then extruded along the direction to create the desired number of three dimensional element layers between an upper and lower bound. Based on the type of two-dimensional element (triangles or quadrilaterals), the resulting quasi-regular mesh is defined from either wedge or hexahedral shaped elements (cf. Figure 3.4) apart for some peculiar configurations leading to degenerate elements.

Remark 3.7 *In general, the final mesh is not structured as the reference mesh is not necessarily structured. However the connectivity of the resulting mesh is derived from that of the reference mesh.*

The main drawback of this approach is the possibility of degeneracies (for instance when part of the domain boundary is coincident with the axis).

General scheme. Schematically, the product technique can be written as follows :

Step 1. Identify and mesh the domain of reference (section)

Step 2. Extrude the reference mesh based on

- the specified direction
- the desired number of layers

Step 3. Optimize the mesh.

Figure 3.4 illustrates the principle of a product method by depicting a source mesh composed of quads and the resulting hex mesh. For the sake of clarity, only one layer of elements, between two sections, is displayed.

3.3 Unstructured mesh generators

In general, structured meshes for arbitrary complex geometries are difficult to obtain in a fully automatic manner. An alternative to a structured mesh consists of using simplices (triangles or tetrahedra). This feature gives the mesh maximum flexibility to address complex geometries as well as to control mesh point distribution. As previously mentioned, unstructured meshes are mostly composed of simplicial elements and the major automatic mesh generation methods produce such elements. However, there also exist some methods resulting in unstructured meshes consisting of quads or hex. Nevertheless, such methods are more tedious both to design and to implement.

In this short survey, we will focus mainly on first type methods (simplicial methods) while the others will be discussed after.

Unstructured mesh generation is a task that may appear both easy and difficult at the same time. The first point is related to the fact that theoretical issues can be used to help the algorithm design for some approaches. The second point, *a*

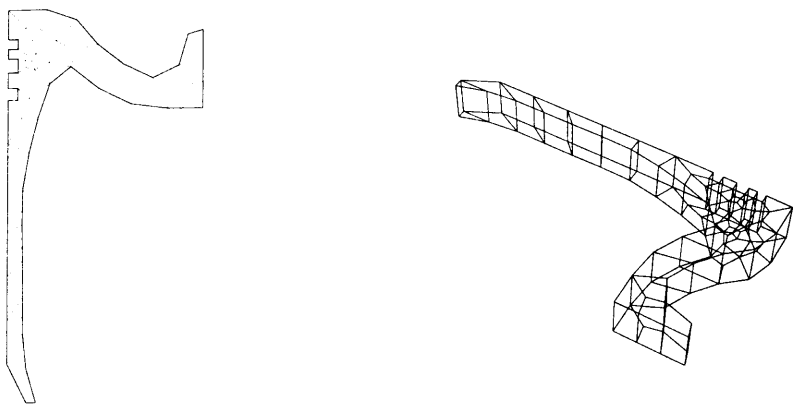

Figure 3.4: *A two-dimensional source mesh (section) and one layer offset in the resulting three-dimensional mesh.*

contrario, is that we don't have such results for some other methods. In addition, whatever the case, the mesh generation technique must be robust and reliable since complex geometries and delicate situations must be envisaged.

After these remarks, we now introduce the main approaches that enable us to create two and three dimensional unstructured meshes. To this end, we give the main characteristics of the various methods while the following chapters will give a detailed discussion on the same methods.

The generation of unstructured meshes involves the creation of points and the relevant connectivities. This is usually achieved through different stages that can be summarized as follows :

Step 1. Definition of the domain boundaries

Step 2. Specification of an element size distribution function

Step 3. Generation of a mesh respecting the domain boundaries[6]

Step 4. Optimization of the element shapes (optional).

The boundary discretization (which represents a polygonal (polyhedral) approximation of the boundary of the physical domain) can be achieved as a separate procedure or simultaneously with the creation of the mesh (in which case, the boundary integrity must be guaranteed by the mesh generation technique).

The element size distribution function can be defined in two ways, implicitly or explicitly. In the first case, either the size of an interior element is deduced from the boundary discretization by interpolation, or, if a control space is supplied, for which the element size is defined at each vertex, the value at any point can be

[6] or the boundary discretization.

computed by interpolation between vertices (cf. Chapter 1). On the other hand, a function $f(x, y, z)$, defined over the entire (three-dimensional) domain, can be either constructed analytically to define explicitly the element size distribution or user-supplied (academic test case).

In general, good-quality meshes cannot be obtained directly from the meshing techniques. Therefore, a post-processing step is required to optimize the mesh with respect to the element shapes. Regardless of the mesh generation method used, topological and geometrical mesh modification techniques are required to obtain a high-quality mesh suitable for finite element computations (cf. Chapters 17 or 18 for more details about the optimization procedures).

Three approaches can be identified in the context of automatic methods including the spatial decomposition based methods, the advancing-front and the Delaunay type methods. After a survey about these methods, we will turn to some other approaches.

3.3.1 Spatial decomposition methods

Spatial decomposition methods were applied to mesh generation purposes about two decades ago [Yerry,Shephard-1983]. In such approaches, the resulting hierarchical tree structure (quadtree and octree in two and three dimensions respectively) serves as a neighborhood space as well as a control space (cf. Chapter 1) used to prescribe the desired element sizes (the element sizes are related to the cell diameter).

General principle. At first, the domain is enclosed in a bounding box (one cell). The domain is then approximated with a union of disjoint, variably sized cells, representing a partition of the domain. The cells are recursively subdivided until each terminal cell is no larger than the desired element size (local value of the element size distribution function). A covering up of a a spatial region enclosing the object (a bounding box) is then obtained. Each terminal cell is then further decomposed into simplices (triangles or tetrahedra), thus leading to a suitable finite element mesh of the domain. The stopping criterion can be based on the curvature of the model entity or supplied by an adaptive analysis error estimate.

This type of method is usually capable of proceeding either directly from a given discretization of the domain boundary or, more generally, by generating the boundary representation of the domain using simple queries to a geometric modeling system (*i.e.*, a C.A.D. system).

General scheme. Schematically, a classical quadtree/octree-based technique involves the following steps :

Step 1. Initializations.

- boundary discretization (or analytical description of the boundaries),
- definition of the size distribution function (if available).

Step 2. Tree decomposition.

- Initialization : the tree representation is derived from a box enclosing the domain
- Recursive subdivision of the box up to a satisfactory criterion

Step 3. Tree balancing : limit the difference between neighboring cells to only one level (the so-called 2 : 1 rule).

Step 4. Cell meshing using predefined patterns (internal cells) and local connections (boundary cells)

Step 5. Optimization : topological and geometrical modifications.

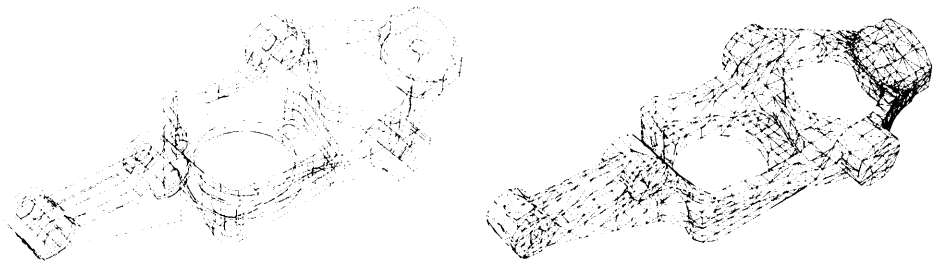

Figure 3.5: *Original CAD model (Patran geometric modeler) and octree-based mesh before optimization (data courtesy of Mac Neal-Schwendler Corp.)*

Main features. The tree decomposition technique produces a set of cells that must have a size that is compatible with the element-size distribution function. As the size of the cells in the tree is directly related to the expected local size, the element size resulting from the method will have a size close to the targeted value. In contrast to other mesh generation methods, there is no particular difficulty at the time the field points are created. As the field points are chosen at the quadrant (octant) corners, this stage is simple and does not require some specific checks (such as a filtering stage used to detect points that are too close together, etc.). However, this strategy of point location induces some rigidity. In other words, the point distribution may conform to the size function but the exact location of these points is not necessarily optimal. Hence, the extent of optimization required after the mesh completion may be relatively great.

The simplicity of the method implies in most cases its robustness. The only problem regarding convergence is related to the cases where it is not so easy to distinguish two entities which are rather close (two very close points which are not directly connected). This is why the case of the corners (points where the incident entities may form a rather acute angle) must be considered with special care.

Boundary discretization. The boundary mesh of the domain boundaries could be an input data of the problem or not.

In the case where this mesh is supplied, it is necessary to identify the points and the characteristics of this mesh (corners, ridges, discontinuities, etc.). On the other hand, it is not necessary to provide this surface mesh with an orientation, which is not the case for an advancing-front technique, for instance. As the spatial decomposition introduces some points (including some in the boundaries), it is necessary to check that the input mesh is a fine enough geometric approximation of the boundaries (to avoid difficulties when creating a point on these boundaries).

In the case where the boundary discretization is not supplied, such a mesh will be automatically created based on the tree structure. In this case, we assume that a geometric modeler is available and is used by means of a series of queries. Indeed, it is necessary to know the point on the boundary closest to a given point, to find the intersection between the boundary and a cell, etc.

Curvature-based refinement. This approach is carried out to improve the accuracy of the geometric approximation of the domain boundary. Hence, finer meshes are achieved in highly-curved regions and coarser meshes in regions of low curvature. The geometric approximation error[7] can be related to a fraction of the desired mesh size as represented by the cell size [Shephard *et al.* 1996]. Hence, the length of a mesh edge is related to the level of the cell in the tree structure.

Tree decomposition. Starting from a box enclosing the whole domain, a tree is developed by recursive partition of its cells. The initial tree includes four cells in two dimensions and eight such cells in three dimensions. Each cell is analyzed so as to determine whether it conforms to a stopping criterion. If not, it is subdivided into four (resp. eight) cells of identical size. At completion, a covering-up of the enclosing box is obtained.

The stopping criterion used in the method is related to various facts based on the application in question. The most widely used criteria state that :

- all cells include at most one boundary point,
- all cells with no point inside include at most one edge (face) of the boundary discretization,
- the size of all cells conforms to the size map,
- etc.

This strategy implies that the decomposition tree enables us to *separate* two close boundary entities. Therefore, for instance in two dimensions, two edges belonging to two opposite but close domain sides will belong to two different cells. This means that at least one point will be created inside the domain in each region. Hence, the tree acts as a neighborhood space and as a separator.

[7] The maximal gap between the mesh edge or face and the curve or the surface.

Tree balancing. Using this tree construction approach results in adjacent cells which can differ greatly in terms of size. Therefore, as the element sizes are related to the cell sizes, a smooth enough size gradation from element to element will be obtained if the size variation from cell to cell is bounded by a factor two (this is the well known 2 : 1 rule). Prescribing such a rule also results in another positive property. In fact, it allows us to know in advance the possible combinations of the cells and then those of the elements resulting from such combinations (at least for the internal cells[8]). It is then easy to define *a priori* all the patterns (the so-called *templates*) that will be subsequently employed to mesh the internal cells at a very low cost.

For efficiency reasons, tree balancing can be carried out during the building of the tree. When a cell subdivision is done, the tree balancing is verified and the adjacent cells are subdivided or not depending on the case, then, in turn, the neighboring cells are considered and such a process is recursively applied, etc.

Filtering step. Due to the way the tree is constructed, say following an arbitrary order (*i.e.*, with no special attention paid to the geometry of the domain),it is possible to have two points related to the intersection of the boundary with a cell side quite close to a cell corner or even to another intersection point (related to another cell side). To prevent the creation of necessarily bad quality elements using these points, a point filtering step is applied. Provided the domain topology is preserved, some points can be merged with one another. Actually, this task may be tedious, particularly in three dimensions where the tree structure could be affected (for instance, when modifying the position of a cell corner).

Element creation. The quadtree-octree method idea consists of using the tree structure for internal point creation as well as mesh element creation. As previously seen, the tree balancing rule reduces the number of transition patterns from cell to cell. Predefined patterns, the so-called *templates* can then be used to quickly fill up the internal cells of the tree. A more general triangulation method, [Shephard,Georges-1991], is necessary to fill up the boundary cells (*i.e.*, those that are intersected by the object boundaries). In this case, it is necessary to conform to the domain topology as well as to the domain geometry. For instance, in two dimensions, one must be sure that an edge related to a given geometric entity actually links two intersection points.

Following this approach, it should be noted that the cell corners and sides will be members of the final mesh where they will be element vertices, edges (faces in three dimensions).

External element removal. The mesh resulting from the previous stage is a mesh of the enclosing box. To obtain a domain mesh, all elements exterior to this domain must be removed. To this end, one needs a coloring algorithm (see Chapter 2) similar to that used in a Delaunay type method.

[8] A cell is said to be internal (with respect to the bounding box) if it is not intersected by the domain boundary.

Optimization. Meshes as completed by this method are globally good quality meshes. Nevertheless, since the internal points correspond to the cell corners (apart from the boundary points), a certain rigidity may be present. Moreover, the tree construction does not explicitly pay attention to the intersections of the boundaries with the cells. In other words, the boundary may intersect a cell close to one side of it. As a consequence, ill-shaped elements can be constructed, for instance rather flat elements. Thus, the classical optimization procedures (Chapter 18) by means of topological and geometrical local operators can be used.

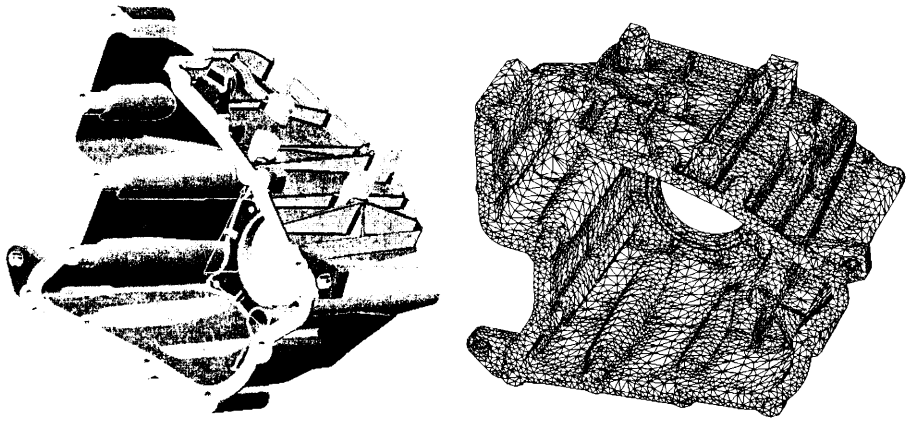

Figure 3.6: *Initial C.A.D. model (CATIA, courtesy of Dassault Systèmes) and octree type mesh (ibid.).*

Numerical issues. Quadtree-octree type methods are relatively easy to implement. However this apparent simplicity may hide some numerical difficulties, for instance, point localization problems (in a cell), intersection problems (from the model and the tree cells), and tree traversal procedures. However, the global complexity (in terms of memory occupation and CPU time) is of order $\mathcal{O}(n)$, where n stands for the number of elements.

3.3.2 Advancing-front method

First suggested by [A.George-1971], this type of method has undergone significant improvements proposed by [Lo-1985], [Löhner,Parikh-1988] and more recently by [George,Seveno-1994], [Löhner-1996b] and [Peraire,Morgan-1997] and [Rassineux-1995]. In this approach, the main idea is to construct the mesh element by element, starting from an initial *front*(*i.e.*, a domain boundary discretization supplied as a list of edges in two dimensions and a list of edges and faces in three dimensions). The technique proceeds by creating new points (or using a set of a priori created points) and connecting them with some points of the current front so as to construct the mesh elements. Thus, the yet unmeshed space is then gradually *nibbled* since the *front* moves across the domain. The front can be simply defined

as the set of mesh entities (edges or faces, thus entities of $d-1$ or d dimensions) separating the part of the domain already meshed from the region not yet filled. The technique is iterative, an entity of the front (edge or face) is selected and a mesh element is formed either by connecting this entity to an existing point or to a newly created point so as to form a new good quality simplex. At each element creation, the front is updated and then dynamically evolves. This iterative process terminates when the list is empty (the domain is then entirely meshed).

General scheme. A classical advancing-front technique can be summarized as follows :

1. Initialize the front with the domain boundary entities which can be sorted based on a given criterion ;
2. Define the element size distribution function which could be provided as input data or constructed as the best from the available input informations ;
3. Select the next entity from the front (based on a specific criterion). This leads to the following :
   - create an optimal point P based on the entity,
   - determine whether a mesh vertex V exists that should be used instead of P. If such a point exists, set V to P,
   - check for element intersection, element size, ..., to validate the above choice,
   - once a correct point has been identified, add the corresponding new element, update the mesh data structure and update the front ;
4. Then as long as the front is not empty, return to 3 ;
5. Optimize the mesh (if needed).

Critical features. Recurrent problems of any advancing-front method include the way to select a front entity, the (optimal) points identification and the element validation checks once a candidate point has been analyzed. All these operations must be made using robust and efficient algorithms since the convergence of the full process is strongly related[9].

Boundary discretization. The mesh of the domain surface (input data of the problem) composes the initial front. Each connected component of the boundary is orientated in a consistent way that allows the domain to be precisely defined with respect to its position around the boundary. In two dimensions, this leads to defining in which way the polygonal segments of the boundary are considered.

[9] In two dimensions, a theoretical issue about simple polygon triangulation (cf. Chapter 6) insures the convergence of an advancing-front method. However, this nice result does not extend to three dimensions.

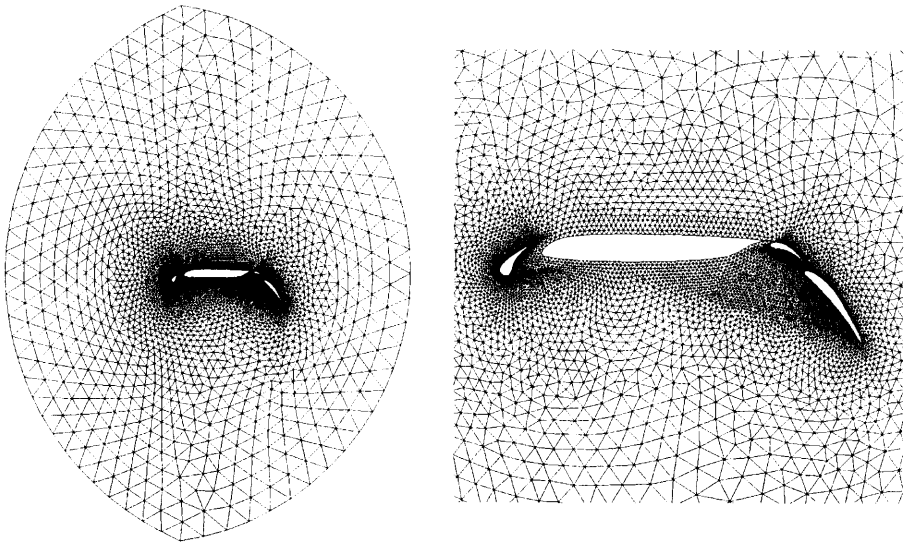

Figure 3.7: *Two-dimensional advancing-front mesh of a multi-airfoil without mesh optimization. Left: entire computational domain, right: partial enlargement around the wing body.*

In three dimensions, the face orientations are such that all the faces (for a given connected component) have their normals with the same orientation. The case of a *non manifold* surface, in three dimensions, is more tedious and requires specific attention (for instance, the *non-manifold* face of the surface can be repeated).

Remark 3.8 *Since the boundary discretization is the initial front, boundary integrity is part of any advancing-front method and thus is preserved.*

Front analysis. The mesh elements are created based on the front entities. The selection of a front entity can be related to various criteria, based on the targeted solution. This operation is not, in general, a purely local process (indeed, it is necessary to anticipate the front evolution to prevent some bottlenecks at a later stage). In two dimensions, one strategy resulting in a nice mesh gradation consists in selecting the smallest front edge (using, for instance, a *heap* structure). In three dimensions, it could be worthwhile selecting the faces at some neighborhood of the last created elements so as to minimize the necessary intersection checks.

Internal point creation. In general, mesh quality is a function of the internal point distribution. Provided that the desired element size is supplied everywhere (cf. Chapter 1, the notion of a control space), several strategies can be used to find the location of a point from a given front entity. An *optimal* point is defined at the place where the element composed of the front entity and this point is regular (equilateral in two dimensions). In three dimensions, such a position belongs to the normal passing through the face centroid G at a distance from G dependent on

the size function that is desired for the element. This location is then, if necessary, iteratively adjusted using an optimization procedure [Seveno-1997].

Candidate vertices. The above optimal point P is not necessarily inserted in the mesh. In fact, another vertex of the current mesh can be seen as a potential candidate due to its proximity to the point P previously computed. This proximity notion depends on the distance from point to point, according to the size specification $h(P)$ at point P when this information is provided (or has been constructed). In this respect, any point in a sphere centered at P, with radius $h(P)$ is *ipso facto* part of the candidate points. The set of such points is ordered based on the distance to P after which these points are analyzed so as to determine which one will be used to construct an element regarded as optimal with respect to the selected front entity.

Validation. The question here is to select among the admissible points the best candidate to construct an "optimal" element (in terms of aspect ratio (shape) and/or size). In fact, one needs to check that a guest element (if retained as a mesh element) has no intersection with the front and does not include any other mesh vertices. These checks must be rigorously implemented and the number of intersection tests must be reduced so as to minimize the cost of the full mesh generation process. To this end, specific data structures (binary trees or *quadtrees-octrees*) can be used to reduce the number of tests necessary. A point leading to an invalid element or leading to any intersection is removed from the list of admissible points.

Then, among the candidate points now identified (for which all the tests have been successful), the one that will result in the best quality possible (in shape or size) must be chosen. Moreover, it is of the utmost importance to analyze precisely the configuration obtained after any valid element creation to decide whether or not it may lead to a delicate or blocked configuration.

Convergence issue. Since the advancing-front method is based on local operations, convergence problems may be encountered, especially in three dimensions. A wide variation in size for the elements between two neighboring fronts may lead to intersection (overlapping) problems and the algorithm may have difficulty when meshing such configurations. As no theoretical results can guarantee that the method will complete a mesh in three dimensions, it is sometime useful to cancel some iterations and thereby removing some elements and points in the mesh. For efficiency reasons, such operations that enable us to overcome some local bottlenecks must be reduced as far as possible.

Front updating. Once an optimal point has been inserted in the mesh, one or several elements are created. The external faces of such elements (those separating the already meshed domain and the not already meshed regions) are stacked into the front list while the edge (face) of the former front used in the construction is removed from this list.

Mesh optimization. Meshes completed by an advancing-front method are in general good quality meshes. Nevertheless, due to the local aspect of the algorithms, it may be useful to optimize the resulting meshes. In such a case, the classical optimization procedures (cf. Chapter 18) are used.

Remark 3.9 *The theoretical complexity of an advancing-front method is established to be in* $\mathcal{O}(n\log(n))$, *where* n *is the number of simplices in the final mesh. In practice, efficient data structures are necessary to achieve this result, as pointed out in [George,Seveno-1994], [Löhner-1996b].*

Surface meshing. A two-dimensional implementation of the advancing-front technique can be adapted to create surface meshes, provided that the surfaces are mapped onto a parametric space (logical space) or by using a direct approach. The aim is to obtain a nice approximation of the real surface by means of a piecewise surface (the mesh) in such a way as to obtain sufficiently good regularity (for instance a G^1 continuity).

In the first approach, the method uses the fundamental forms of the surface and completes an anisotropic mesh in a planar domain, the parameter space. This mesh is then mapped onto the physical space in such a way as to bound the gap between the triangle edges and the surface. This is done thanks to the so-called tangent plane metric or the metric of the maximal radii of curvature (cf. Chapters 13 and 15) that make it possible to compute the lengths of the segments in the parameter space using the geometry of the real surface (for instance, see [George,Borouchaki-1997]).

Remark 3.10 (Variant) *The same technique can be applied to the generation of quadrilateral or hexahedral elements. In two dimensions, it is then usually known as the* paving *technique, [Blacker,Stephenson-1991], and in three dimensions, the* plastering *technique, [Blacker,Meyers-1993].*

3.3.3 Delaunay technique

The computational-geometry properties of the Delaunay triangulations have been investigated for many years [Delaunay-1934]. Even before this date, in 1850, Dirichlet proposed a method to decompose a domain into a set of convex polyhedra [Dirichlet-1850]. However, the application of these approaches to mesh generation has only more recently been explored [Hermeline-1980], [Cendes *et al.* 1985], [Cavendish *et al.* 1985], [Baker-1986], [Weatherill-1985], [Mavriplis-1990]. The earliest strategies used predetermined sets of points as the Delaunay construction provides a suitable technique to connect these points, although it does not provide a mechanism to generate points. Moreover, the Delaunay triangulation of a domain may not preserve boundary integrity[10] which is a key requirement for mesh generation procedures and thus, this point must receive a special attention. Most of the current procedures for point insertion are based either on

[10] *i.e.*, does not conform to a given boundary discretization.

Bowyer-Watson's algorithm [Bowyer-1981], [Watson-1981] or Green-Sibson's algorithm [Green,Sibson-1978].

For a given set of points $\mathcal{S} = \{P_k\}$, $k = 1, N$, a set of regions $\{V_k\}$[11] assigned to each of these points can be defined, such that any location within V_i is closer to P_i than any other of the points :

$$V_i = \{P : |P - P_i| \leq |P - P_j|, \forall j \neq i\}.$$

The regions are convex polyhedra, the Voronoï regions or cells. Joining all the pairs P_iP_j sharing a common segment of a Voronoï region boundary results in a triangulation of the convex hull of $\mathcal{S}$, the so-called *Delaunay triangulation.* The set of triangles (resp. tetrahedra) defining the Delaunay triangulation satisfies the property that the open circumcircle (resp. circumsphere) associated with the nodes of the element is empty (*i.e.*, does not contain any other point of $\mathcal{S}$). This condition is referred to as the in-circle (resp. in-sphere) criterion and is valid for n-dimensional space.

In this approach, an initial mesh is constructed, for instance from a bounding box [12] enclosing the boundary discretization (list of edges and/or faces) of the domain. All boundary points are inserted iteratively into the initial triangulation of the bounding box, thus leading to a Delaunay triangulation with no internal point. The boundary connectivity constraint is not taken into account in this construction scheme. Hence, it is necessary to ensure that the entities of the boundary discretization are present in the resulting Delaunay triangulation and, if needed, to retrieve the missing entities by modifying the triangulation. Local mesh modifications are applied to remedy the situation and to finally obtain a mesh of the bounding box of the domain conforming the given discretization. Then, the exterior elements are removed (using a coloring scheme) and internal points can be created and inserted in the current mesh. Finally, the resulting mesh can be optimized.

General scheme. Provided with a size distribution function and a boundary discretization of the domain under interest, the global procedure for the mesh generation using a constrained Delaunay method can be outlined as follows :

Step 1. Initializations :

- input the boundary entities,
- construct an initial triangulation $\mathcal{T}_B$ of the bounding box of the domain.

Step 2. Insert all boundary points into $\mathcal{T}_B$.

Step 3. Construct an empty mesh $\mathcal{T}_e$ (no interior points) starting from $\mathcal{T}_B$:

- recover the missing boundary entities (boundary integrity),

[11] known as the *Dirichlet tessellation* or the Voronoï cells.

[12] Introducing a bounding box is not strictly required, but is a source of simplification. Without this trick, the discussion becomes more subtle and makes it necessary to include several situations (rather than just one).

- identify the connected components of the domain.

Step 4. Internal point creation and point insertion (*i.e.*, enrich $\mathcal{T}_e$ so as to obtain a mesh $\mathcal{T}$).

Step 5. Mesh optimization.

In the above scheme, some steps deserve a special attention as they are connected to the robustness of the method and influence the quality of the resulting meshes.

Delaunay kernel. Let $\mathcal{T}_i$ be the Delaunay triangulation of the convex hull of the set of points $\mathcal{S} = \{P_k\}$, $k = 1, i$, where $i = 1, N$. The insertion of $P = P_{i+1}$ in $\mathcal{T}_i$ results in the triangulation $\mathcal{T}_{i+1}$. This can formally be written as

$$\mathcal{T}_{i+1} = \mathcal{T}_i - \mathcal{C}_P + \mathcal{B}_P \,,$$

where $\mathcal{B}_P$ denotes the set of elements formed by joining P with the external edges (resp. faces) of the set of elements $\mathcal{C}_P$, whose circumcircle (resp. circumsphere) contains P. This insertion procedure is known as the *Delaunay kernel.*

Boundary integrity. Boundary recovery is commonly performed before the insertion of internal points, just after the insertion of boundary points. An alternative approach recovers the boundary integrity in the final stage of the mesh generation process, although it may result in a poor quality mesh, thus requiring a final optimization stage [Mavriplis-1995].

One technique of recovering boundary integrity consists of inserting a number of additional boundary points until the triangulation conforms to the boundary, although the initial boundary discretization is obviously not preserved. An alternative (and more elegant) approach consists of modifying the Delaunay triangulation using local mesh modifications operators to conform the boundary (cf. Chapter 18). This procedure matches exactly the specified boundary discretization. In two dimensions, if a boundary edge is missing, but its two endpoints belong to two adjacent triangles, an edge swap is used to recover the missing edge. In three dimensions however, the implementation of the procedure is more complex and more tedious. Moreover, additional points (the so-called *Steiner* points) are often required to enable the boundary recovery procedure to be performed [George *et al.* 1991a].

Field point creation. Various approaches have been investigated to create internal points. One strategy consists of inserting the new mesh points at the circumcenters of the elements [Holmes,Snyder-1988], [Weatherill,Hassan-1994]. This technique results in meshes for which the (dihedral) angles are bounded. However, the resulting meshes can be irregular and the mesh gradation is not well handled. An alternative approach consists of driving the point creation by the boundary point distribution.

It is assumed that the point distribution on the surface matches the geometric (curvature) and finite element requirements. This distribution is then extended into the domain using an interpolation scheme. One way is to create the points along the internal mesh edges, first in the empty mesh and then in the current mesh, so as to conform to the desired element-size distribution function [George *et al.* 1991b]. In this approach, the point creation is controlled by a background mesh (actually the empty mesh).

A third technique uses point sources considered as control functions with elliptic partial differential equations [Weatherill,Hassan-1994]. Another technique consists of using an advancing-front point-placement strategy to create the internal nodes. The front is then defined as the transition region between well-shaped and badly-shaped elements [Rebay-1993], [Frey *et al.* 1998].

Once created, the internal points are then inserted randomly in the current mesh using the Delaunay kernel procedure.

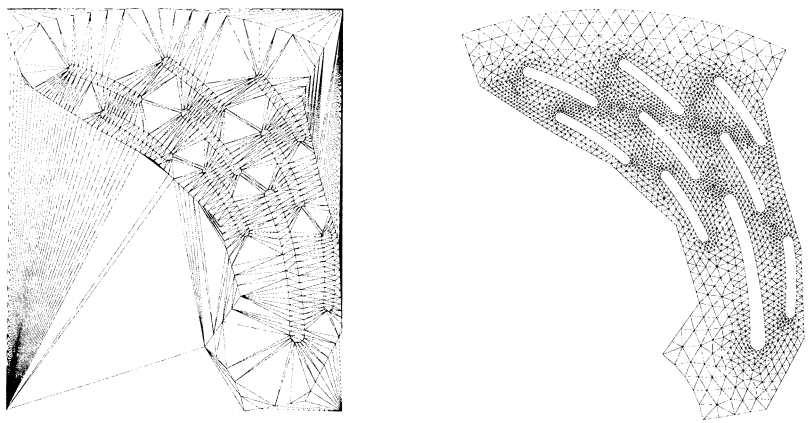

Figure 3.8: *Delaunay mesh of a mechanical device. Boundary mesh with no internal point (left-hand side) and resulting mesh (right-hand side).*

Robustness. In two dimensions, this technique is very robust and reliable [13]. In three dimensions however, the lack of a theoretical proof for the boundary recovery, that can be translated in terms of computer procedures, greatly diminishes the robustness of the method, which relies explicitly on heuristic techniques.

The Delaunay method results in a very efficient mesh generation technique. Most of the operations can be achieved in constant time (provided adequate data structures, cf. Chapter 2). The overall complexity (space and time requirements) is $\mathcal{O}(n \log(n))$. In practice, this method appears to be one of the most efficient meshing techniques available today, as speeds in excess of 250,000 elements / minute on current workstations have been reported by several authors.

[13] It is easily proven that the boundary integrity can always be recovered.

3.3.4 Tentative comparison of the three classical methods

Without seeking to provide a classification, it could be of interest[14] to observe the meshes the three methods previously discussed (*quadtree*, advancing-front or Delaunay) are likely to complete using the same description of a domain. This is quite easy to do since all these methods assume the same input data, a boundary mesh, and complete the same type of meshes (simplicial meshes).

Figure 3.9 displays different meshes of the same computational domain resulting from the quadtree, advancing-front and Delaunay type methods. Table 3.1 adds some statistics about the number of vertices np and the number of elements ne, the shape quality of the mesh $\mathcal{Q}_M$ and the quality value of the worst element $\mathcal{Q}_{worst}$ in these meshes (Chapters 1 and 18).

method	np	ne	$\mathcal{Q}_M$	$\mathcal{Q}_{worst}$
quadtree	1,246	2,171	1.25	1.88
advancing-front	2,557	4,795	1.1	1.61
Delaunay	2,782	5,528	1.16	1.82

Table 3.1: Statistics relative to the different meshes depicted in Figure 3.9.

Observing these examples indicates a non-negligible variation in size for the meshes (the number of elements is in a ratio of 2 or more) while the mesh qualities are about the same (close to 1 !). The CPU costs are all less than one second. Thus, at least in two dimensions, all the methods produce rather similar meshes (in terms of quality). Notice that the element sizes inside the domain are only related to the boundary discretization provided as input data. The quadtree type method, based on a gradation rule of type 2 : 1, results in larger elements inside the domain (say, far away from the boundary) and thus produces fewer elements than the two other methods. Advancing-front and Delaunay meshes are rather similar in terms of size.

A more subjective view could produce some differences regarding the *aesthetics* of the created meshes.

3.3.5 Other methods

Various other techniques have been developed for unstructured mesh generation, although none of them seems to be widely used today. However, two classes of techniques offer promising capabilities in specific fields of applications : hybrid methods, which are useful for CFD (Computational Fluid Dynamics) computations, and partitioning methods, which can be used in parallel applications.

Hybrid methods. This approach combines features of structured meshes (in general in the vicinity of the domain boundaries) and unstructured meshes (for instance, [Weatherill-1988], [Kallinderis *et al.* 1995], [Khawaja *et al.* 1995], and

[14] Moreover, such an idea is not really discussed in the literature and, in addition, implies that several methods are available in the same place.

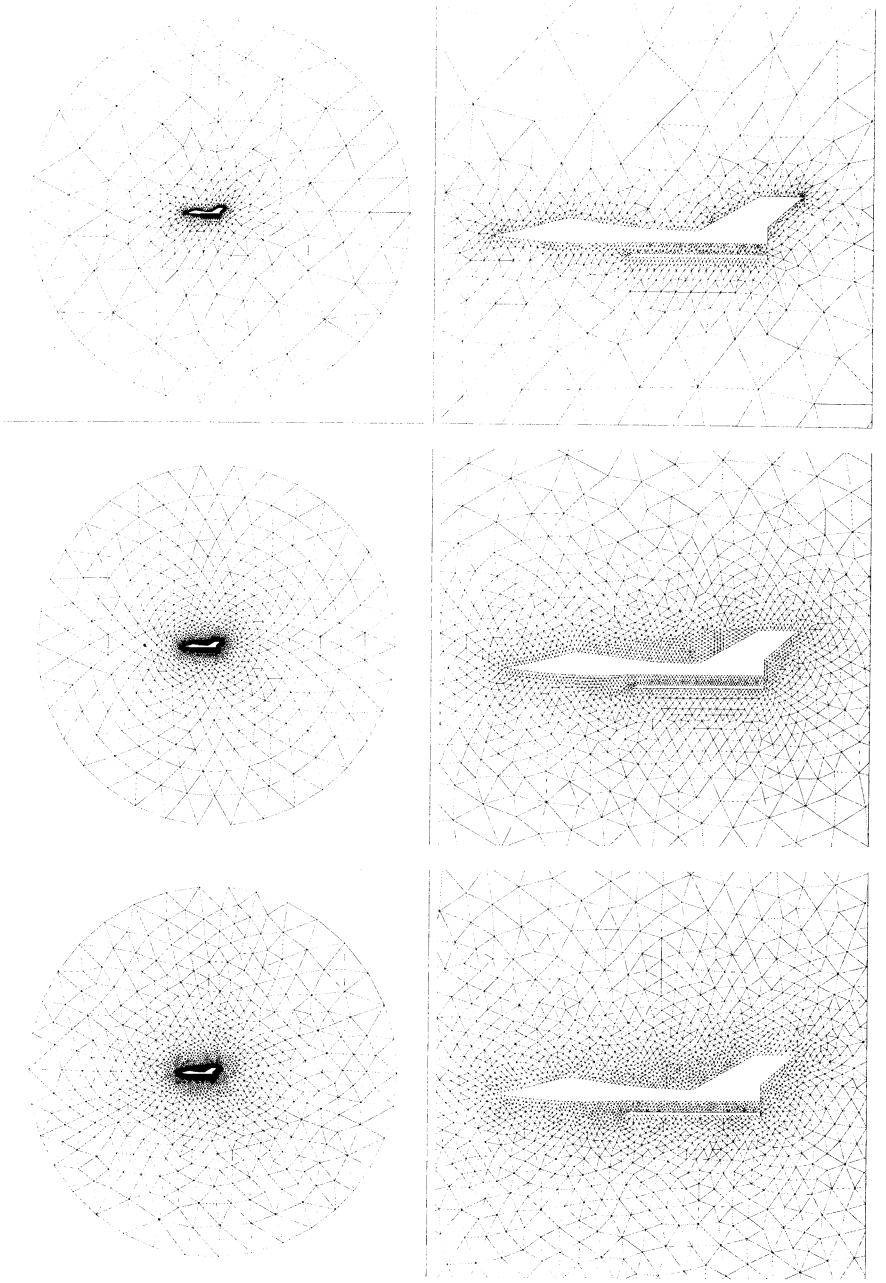

Figure 3.9: *Overview of the different mesh generation methods when applied in a domain (used, for instance, for a C.F.D. problem). Quadtree type mesh (top), advancing-front type mesh (middle) and Delaunay type mesh (bottom) including a close-up view around the fuselage.*

[McMorris,Kallinderis-1997]). In hybrid prismatic/tetrahedral meshes, the initial surface triangulation is the outer prismatic surface. In general, layers of prisms are used to resolve boundary layers and wakes, while tetrahedral elements cover the remaining part of the computational domain. A hybrid type mesh combining elements of different orientations seems more flexible to accommodate the different flow features. The most common technique employed for generating prismatic elements is a *marching* method that starts from a surface and propagates towards an outer boundary. The marching direction vectors are the normal at the surface vertices and the marching distances along these vectors (*i.e.* the stretching of the nodes along the direction) are dependent on the physics of the problem (for instance related to the Reynolds number), [Garimella,Shephard-1998]. The grid is built one layer at a time in an iterative process. The unstructured mesh is generated using a classical unstructured mesh generation method (Delaunay, advancing-front, octree, etc.).

The tedious part in developing this type of mesh generator lies in the management of the interfaces between the structured and unstructured meshes. For instance if a Delaunay algorithm is used, the boundary integrity constraint may be relaxed (or even omitted). A promising technique consists of using a *buffer* layer for the transition from the prismatic to the tetrahedral elements (for instance, pyramidal elements with quadrilateral bases can be introduced). An alternative method consists in allowing the outer layer of prismatic elements to be broken down during the boundary recovery stage.

Partitioning methods So far, this chapter has only dealt with unstructured mesh generation methods that generate simplicial meshes. Quadrilateral meshes may be desirable in some applications (structural mechanics for instance) where they result in increased computational performance and numerical accuracy. In this context, partitioning methods offer a way to design automated algorithms for producing well-structured quadrilateral or hexahedral meshes.

Most of the first partitioning algorithms subdivided the domain recursively until simple elements (*i.e.*, patterns) or very simple transition meshes remained [Cavendish-1974], [Schoofs *et al.* 1979]. This class of techniques is known as recursive partitioning. Another approach consists of separating the mesh generation in two phase : automatic subdivision of the domain into subregions and meshing of the resulting subregions. The first stage is based on the identification of suitable subdivisions and uses the medial axis (surface) of the domain or the Voronoï diagram of its edges (faces) [Tam,Armstrong-1991], [Armstrong *et al.* 1995]. In the second stage, an algebraic method (or a similar method) can be used to mesh the various regions resulting from the partition.

Quadrilateral meshing. Two approaches (direct and indirect) may be adopted to generate quadrilateral meshes for domains of arbitrary shape.

- Direct methods.

Among the direct methods, essentially two approaches have been investigated : a domain decomposition technique followed by quadrilateral sub-domain filling by

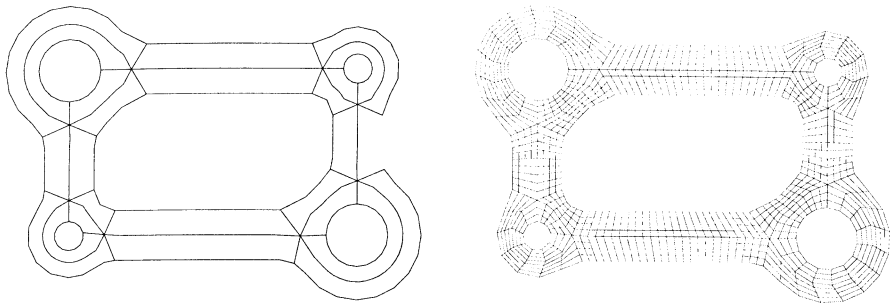

Figure 3.10: *Partitioning method based on medial axis subdivision : skeleton of the domain (left-hand side) and quadrilateral mesh of the domain (right-hand side).*

means of an algebraic method [Armstrong *et al.* 1995], [Talbert,Parkinson-1991] and the quadrilateral paving techniques [Blacker,Stephenson-1991]. The first approach is domain decomposition sensitive and relies on the quasi-convex nature of the resulting sub-domains. The domain decomposition algorithms usually require local or global knowledge of the domain. In particular in this last case, the skeleton fully defines and allows an accurate decomposition of the domain. The second method consists in paving the domain from the boundary to the interior and managing the front collisions. By its very nature, the performances of this method are closely related to the boundary discretization.

When a constant isotropic metric field is specified, these two classes of methods are likely to lead to the same results. In fact, in an advancing-front method, the front shape tends toward the skeleton. On the other hand, if a generalized metric map is specified, the second method is more likely to respect the field.

- Indirect methods.

Given a triangular mesh of the domain, the indirect approaches aim at combining triangles to form quadrilaterals [Lo-1989], [Johnston *at al.* 1991], [Lee,Lo-1994], [Zhu *et al.* 1991], [Rank *et al.* 1993], [Lewis *et al.* 1995] and lead to two related merging processes. The triangle merging procedure is either driven by the quadrilateral quality [Borouchaki,Frey-1998] and may lead to mixed triangular - quadrilateral meshes, or starts from the boundary and moves to the interior of the domain, ensuring an even number of vertices when two fronts collide and results in pure quadrilateral meshes (if the boundary discretization has an even number of vertices). This second method requires a topological classification of the front collisions.

The resulting quad meshes can be enhanced using a specific optimization procedure capable of optimizing the shape or the size quality of the elements[15] (cf. Chapter 18).

[15] While bearing in mind that merging two good quality triangles does not necessarily lead to a good quality quad.

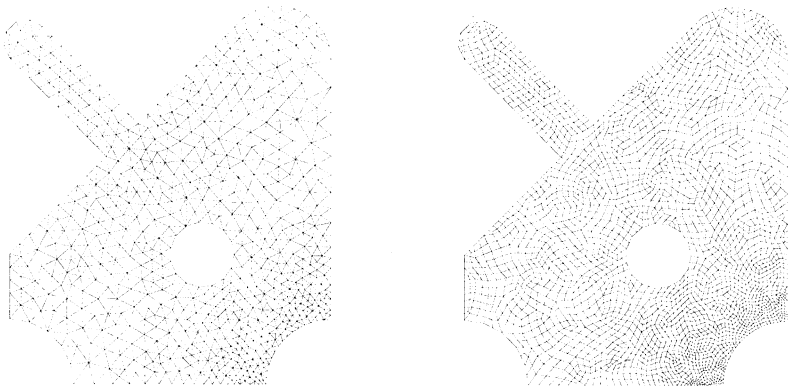

Figure 3.11: *Triangular to quadrilateral conversion, indirect approach. Original triangular mesh (left-hand side) and optimized quadrilateral mesh (right-hand side.*

Mesh generation by local optimizations. In the context of moving (evolving) mesh methods (such as those encountered in forming calculations for instance), it is often desirable to allow the mesh topology to change progressively rather than to build a new mesh. The remeshing is required to avoid element distortions due to large deformations and to adapt the mesh dynamically to the new conditions [Coupez-1997].

An initial mesh of the domain can be optimized for any kind of criterion, for instance one related to the volume of the elements. Starting from a very crude mesh (for instance obtained by connecting a node to each face of the boundary of a non-convex domain, thus leading to overlapping elements), the optimization stage will attempt to minimize the sum of the absolute value of the element volume. The improvement process tends to optimize an element shape quality function (cf. Chapter 18), and internal nodes can be introduced to remove locked configurations and to optimize the mesh.

3.4 Surface meshing

Surface meshes play an important role in numerical simulations using finite element methods and the quality of the geometric approximation may affect the accuracy of the numerical solutions. In this context, a surface mesh is usually intended to be the boundary description of a domain used in a three-dimensional finite element analysis. Therefore the surface mesh must conform to specific properties, related for instance to the geometry of the surface it represents or to the behavior of the physical phenomenon. The aim is to create an optimal piecewise planar approximation of the original surface in which the maximal distance between the original and the approximating surface does not exceed a given tolerance.

Depending on the surface definition, three techniques have been investigated : mesh generation via a parametric space (if a CAD modeling system has been defined, for instance), mesh generation for implicitly defined surfaces (e.g. isosurfaces) and, if the sole data is a given surface discretization (*i.e.*, a surface triangulation), surface mesh optimization (which proves useful, for instance, in the study of large deformations in structural mechanics).

3.4.1 Parametric mesh generation

A regular surface parameterized by u, v can be defined using a function σ as :

$$\sigma : \Omega \subset \mathbb{R}^2 \longrightarrow \Sigma \subset \mathbb{R}^3, (u, v) \longmapsto \sigma(u, v), \tag{3.3}$$

where Ω is a domain of $\mathbb{R}^2$ and σ is a smooth enough function. The goal is to achieve the final surface mesh via a triangulation in the parametric (logical) space. The two-dimensional mesh generation is governed by a set of metrics related to the intrinsic properties of the underlying surface. These metrics correspond to sizing or directional specifications. The control induced in this way is then explicitly written as a criterion about the lengths of the mesh edges, see [Dolenc,Makela-1990], [Samareh-Abolhassani,Stewart-1994], among many others.

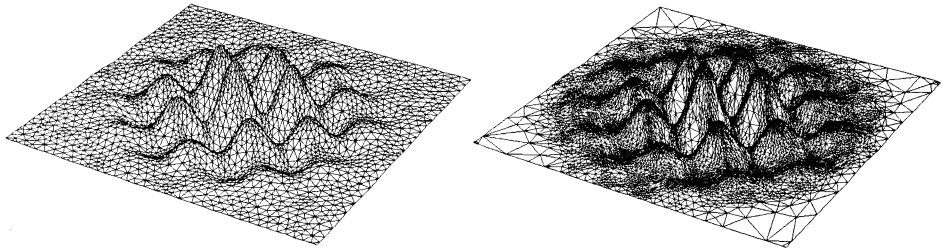

Figure 3.12: *Parametric surface meshing, analytical example. The surface is defined as :* $z(x, y) = 2.5e^{-0.1(x^2+y^2)} \sin(2x) \cos(2y)$ *in the domain* $[-6, 6] \times [-6, 6]$ *of the Oxy plane. Left-hand side : constant-size surface mesh, right-hand side : geometric surface mesh.*

General scheme. Let Ω be the domain in $\mathbb{R}^2$ corresponding to the parameterization of a surface Σ. The parameter space is supplied with a Riemaniann structure, used to govern the meshing process, which is constructed according to the nature of the expected mesh (isotropic, anisotropic, specified sizes, uniform sizes, etc.). Actually, the metric of the first fundamental form of the surface[16] is involved, which is defined by interpolating the discrete metric map associated with a given mesh of Ω. The issue here is to mesh Ω with unit length mesh edges (with regard to the metric) and in such a way that the resulting elements are of good quality.

[16] and even that of its second fundamental form.

The domain meshing process consists of three stages. The two first concern the parameter space and consist in meshing the boundary of Ω and then in meshing Ω using this boundary mesh as input data. The final stage consists in mapping this mesh onto the surface.

Meshing a surface boundary. The discretization of the curves defining the surface boundary enables us to construct a geometric support using a well-suited mathematical representation. This support is approximated by a polygonal segment whose constitutive segments are unit length segments. This polygonal segment is the sought mesh (Chapter 14) unless a map that is not necessarily of compatible size must be adopted. This mesh is constructed using a mesh of the boundaries of Ω and, at completion, we have the boundary discretization ready.

Meshing the domain. To complete the surface mesh, we construct, using a suitable method (Delaunay or advancing- front, in general), a mesh whose vertices are the points of the boundary discretization together with a series of internal points such that unit edges and good quality elements are obtained (with respect to an appropriate metric). This construction is made in Ω using the discretization of the sides of this region as previously created.

Mapping onto the surface. Mapping the mesh in Ω onto surface Σ is rather easy, one just needs to apply function σ. The vertex positions are the image by σ of the vertices in the parametric space. The connections are those of this planar mesh.

Remark 3.11 *This technique can be applied to surfaces defined using several patches, each of which corresponds to a parametric space. In this case, the meshing procedure starts by meshing the interfaces between the patches so as to insure mesh conformity. In this way, a patch-dependent mesh is obtained. Notice that this constraint can be overpassed (Chapter 15).*

3.4.2 Implicit surface triangulation

Recent years have witnessed increasing interest in the use of implicit functions for defining geometric objects, for instance in the field of Computer-aided geometric design [Requicha-1980], [Ricci-1972], [Wyvill *et al.* 1986], [Pasko *et al.* 1995] or in applications where the domains involved are obtained using tridimensional scanning and sensing devices (e.g. biomedical imaging systems). They are called *implicit* because they represent subsets of $\mathbb{R}^3$ that are not specified explicitly by their boundaries or parameterizations.

An implicit algebraic surface can be defined as the set of points (x, y, z) in $\mathbb{R}^3$ that conform to an equation such that :

$$f(x, y, z) = 0\,. \tag{3.4}$$

A geometric object is considered as a closed subset of $\mathbb{R}^3$ with the definition $f(x, y, z) \geq 0$. The boundary of such an object is a so-called implicit surface. The

defining function f may be defined or approached in different ways, depending on the field of application.

Scheme of the method. There are relatively few papers[17] on implicitly defined surfaces (cf. [Ning,Bloomenthal-1993] for an overview).

The classical scheme as proposed by [Allgower,Schmidt-1985] and now well recognized in most of the approaches is based on two operations :

- a sample of the function values at the vertices of a covering-up set of the domain,
- the connection of these vertices in order to obtain a mesh.

The sampling step aims at creating a set of points belonging to the implicit surface. This task is delicate as, in general, one has to solve non-linear equations. The aim of the connection step is to construct a topology similar to that of the surface and, in addition, such that a well-suited surface approximation is obtained (from the geometric point of view).

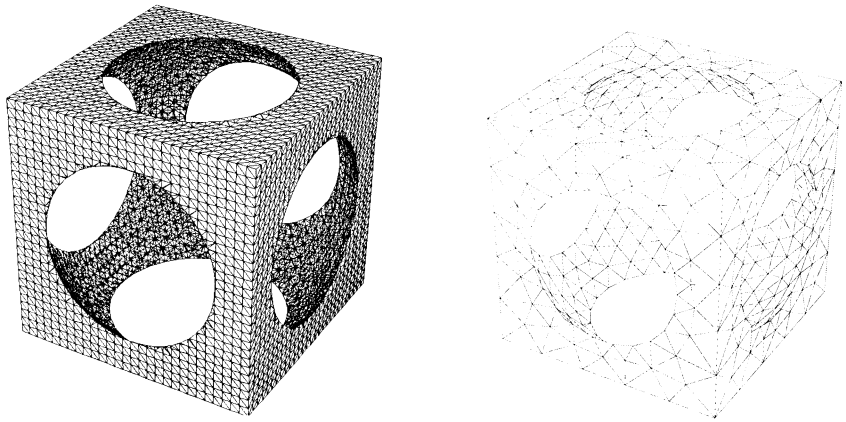

Figure 3.13: *Example of implicit surface meshing, the domain is defined as the extrusion of a sphere from a cube using CSG primitives. Left : original triangulation based on a regular grid partitioning, right : optimized geometric surface mesh.*

Various approaches. A popular technique consists in sampling the basis function in the space and uses a numerical method to find the zeros of this function. The mesh vertices will then correspond to the roots resulting from this algorithm. In practice, a spatial decomposition of the domain is developed (for instance, using an *octree*) and the function is approximated locally (in each cell) by a piecewise surface. The global conformity of the mesh is insured by the cell subdivision rule (using the same idea as in an *octree* type method, see above). The cell size may depend on the local curvature or on some other explicitly defined parameters. There

[17] In comparison with the literature on parametric surfaces.

are two classes of algorithms depending on the nature of the data. In the case of discrete data, the implicit function is not exactly accessed since the sole values available are those at the vertices of the covering-up. A linear interpolation can then be used to compute the values of the function everywhere in the domain and thus find the intersections between the function and the edges of the covering-up.

A simple numerical technique for constructing such a covering-up uses an octree [Bloomenthal-1988]. Lorensen and Cline [Lorensen,Cline-1987] introduced an algorithm [18] which is now commonly used for constructing a polygonal representation of a constant density surface using voxels, and numerous algorithms to guarantee topological correctness of the polygonization of isosurface have been proposed since then. Also a Delaunay mesh of the convex hull of the points in the sample can be used to construct this covering-up [Frey,Borouchaki-1996]. The meshes obtained by any of these techniques are then optimized (for instance according to size specifications) using classical mesh modifications operations (cf. Chapter 19). This is due to the fact that no special attention is paid to the quality of the triangles at the time they are constructed since the only concern is to suit the surface.

3.4.3 Direct surface meshing

This approach consists of applying a classical meshing technique directly to the body of the surface, without using any kind of mapping [Shephard,Georges-1991], [Nakahashi,Sharov-1995], [Chan,Anatasiou-1997], [McMorris,Kallinderis-1997]. In such approaches, the mesh element sizes and shapes are controlled by analyzing the surface variations. At first, the curves representing the surface boundary are discretized, then the surface mesh is created using any unstructured meshing technique. The difference between the various approaches proposed lies in the (iterative) algorithm used to find an optimal point location on the surface. An optimization stage is usually required to improve globally the element shapes after the generation of the initial surface mesh.

3.4.4 Surface remeshing

Although the surface meshing approaches described in the previous sections seem appropriate, in many cases the domains are not defined in terms of analytical functions but rather by means of a surface triangulation. Such applications include : numerical simulations where the surface results from measurements, biomedical engineering where the domain is provided by a sensing or scanning device, numerical simulations that involve remeshing (e.g. forging problems, fluid-structure interaction problems, etc.). In this context, we consider the tedious problem of generating geometric finite element meshes given an arbitrary surface triangulation representing the surface, possibly supplied with geometric specifications (ridges, singular points, etc.) [Löhner-1996a], [Frey,Borouchaki-1998].

[18] the so-called *Marching Cubes*.

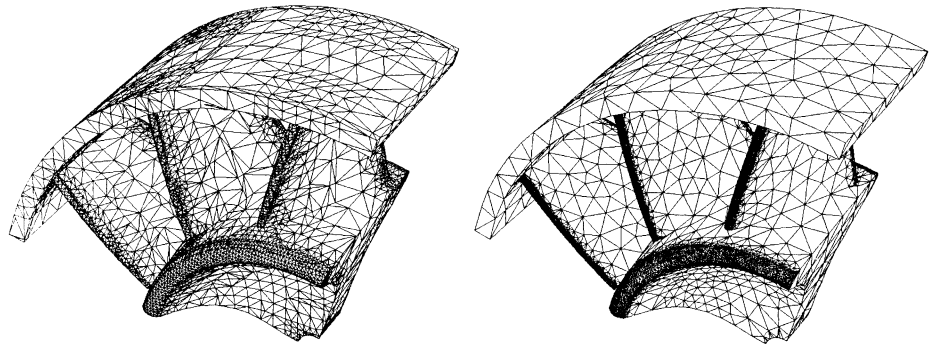

Figure 3.14: *Example of direct surface meshing (octree-based method with curvature-based refinement). Left-hand side : original triangulation (data courtesy of Mac Neal-Schwendler Corp.), right-hand side : optimized surface mesh.*

Problem statement. We are concerned with a case where the surface to be meshed is defined through an initial triangulation enjoying some geometric properties[19]. From this triangulation, a mathematical support (with a continuity G^1 in general) is constructed which will be used to collect the required information by a system of queries. The problem then involves constructing a new mesh, conforming to the given specifications, by means of successive modifications applied to the initial triangulation. The required information are as follows :

- the placement of a point on the surface,
- the surface property collection at a local level (discontinuities, minimum radius of curvature, main curvature radii, normals and tangents, etc.).

The remeshing procedure uses local modification operators that can be of a topological nature (to control the geometric approximation) or of a metric nature (subdivision of edges that are too long, vertex removal, vertex relocation, etc.).

Control of the geometric approximation. The initial surface triangulation is optimized in accordance with the geometry to obtain a so-called *geometric* mesh (regarding the geometric approximation of the surface) and also to the element quality, in other words, a mesh which best fits the surface geometry. In such a mesh, the maximal gap between an edge of the discretization et the real surface is bounded (provided by a tolerance threshold value). Additional constraints about the shape quality, to obtain a finite element mesh (regarding the element shape and sizes) can be added.

Remark 3.12 *The problem is to start from an initial surface triangulation which contains a reasonably small number of elements and still represents an accurate polyhedral approximation of the surface. Usually, the given initial surface triangulation needs to be geometrically simplified (with respect to a geometric tolerance), prior to building the geometric support.*

[19] If a C.A.D. model is available, the surface is known using a series of queries to the modeler.

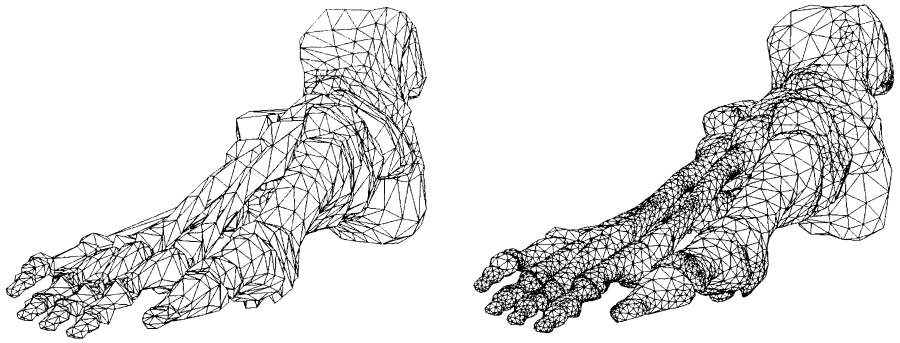

Figure 3.15: *Surface mesh optimization.Polyhedral biomedical iso-surface reconstruction from volumetric data (public domain data, Naval Air Warfare Center Weapons Division). Initial triangulation (left-hand side) and optimized surface mesh for a tangent plane deviation lesser than 37 degrees (right-hand side).*

Governed surface remeshing. The given triangulation initializes the current mesh. In this mesh, we identify the singularities (corners, ridges, C^0-discontinuities). A size map is constructed by evaluating in a discrete manner the intrinsic properties of the surface (Chapter 19). The edges in the current mesh are then analyzed one at a time so as to obtain unit mesh edges. This leads to :

- subdividing each edge judged too long (having a length greater than "one") into sub-edges of unit length (or a value close to one),
- removing any edge that is too short (provided the topological consistency is preserved).

Point enrichment or removal are combined with edge swap (cf. Chapter 18) in such a way as to enhance element quality. This process is repeated as long as one edge is judged invalid (with respect to the given specification). At completion, The surface mesh conforms to the intrinsic size map (*i.e.*, the geometric map) or to any other given map.

3.5 Mesh adaptation

Solution-adaptive meshing is a very promising technique that improves the numerical accuracy of the solution at a lower computational cost. It relies on a more efficient (optimal) point distribution (in terms of their number and their location), and also on the shape of the element in the mesh (isotropic or anisotropic shape, for instance). A new mesh is then constructed (using one of approaches discussed in this chapter or by using a remeshing procedure), then the process is repeated, the new mesh being assumed to better capture the physics of the problem in hand. The successive iterations aim at optimizing this distribution based on an *a posteriori* error estimate. Starting from an initial mesh, an initial solution is computed.

It is then analyzed by an error estimate and this analysis is written in terms of size specifications used in turn to govern the mesh adaptation. The solution accuracy is also strongly related to the interpolation step from the computational mesh (the previous iterate) and the current mesh.

3.5.1 Mesh adaptation scheme

Despite several differences between the possible approaches that are suitable for adaptive meshing, the following steps are usually representative of an adaptive meshing strategy.

- Construction of an initial mesh $\mathcal{T}_i$,
- Computation of the initial solution u_i on $\mathcal{T}_i$
- (A) Estimation of the local error in u_i,
- (Re-)Construction of a mesh $\mathcal{T}_{i+1}$ according to the estimated error values :
  - construction of the control space $\mathcal{CS}_i$ associated with u_i,
  - construction of the governed mesh $\mathcal{T}_{i+1}$ with respect to $\mathcal{CS}_i$,
- Solution transfer on $\mathcal{T}_{i+1}$,
- Resumption of the solution procedure (return to (A) with $i = i + 1$).

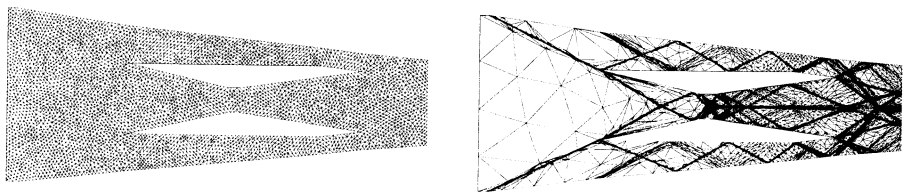

Figure 3.16: *Adaptively generated mesh for the computation of a supersonic flow (Mach 3) over a Scramjet. Left-hand side : initial mesh (4,000 points), right-hand side : final mesh (90,000 points).*

3.5.2 Mesh adaptation techniques

Adaptation methods can be broadly classified into three categories. The first category consists of a local or global modification of an initial mesh so as to adapt it to the computational requirements. The second category includes the global methods that reconstruct the whole mesh at each iteration step. Finally, some other methods combine these two approaches. The adaptation is first made at a local level during a few iterations, then the mesh is entirely reconstructed prior to being locally updated again.

In this classification, we can distinguish between r-methods, h-methods, p-methods and the coupling of these last two resulting in hp-methods (cf. Chapters 21 and 22).

Adaptive remeshing. The mesh on which a solution has been computed becomes the *background mesh* for the next iteration step. A discrete element-size specification function is defined at the vertices of the background mesh, for instance from a Hessian-based criterion or any type of error analysis method. A new unstructured mesh is then generated by a classical mesh technique governed by this new size map [Peraire *et al.* 1987], [Shephard *et al.* 1988], [Löhner-1989], [Mavriplis-1990].

Mesh modifications. Local remeshing (refinement, coarsening) is an alternative (and sometimes less computationally expensive) approach to mesh adaptation, based on mesh optimization techniques. The current mesh is modified in the regions where the discrepancies between the current and the specified element size are too large [Rivara-1984b], [Cendes,Shenton-1985a], [Rivara-1991]. Subdivision refinement (*h-refinement*) can be used to produce nested meshes (*i.e.*, containing a subset of the initial mesh vertices), thus allowing an accurate transfer of variables from one mesh to another (in multigrid approaches, [Désidéri-1998]).

3.6 Parallel unstructured meshing

The requirement to develop fast and reliable unstructured mesh generation algorithms is common to several computational fields of application. Moreover, large meshes (in excess of one million elements or even several million elements, for instance in some CFD problems or wave propagation or crash problems) are now frequently managed in these disciplines (for instance in computational fluid dynamics simulations). In order to benefit from parallel architectures, the whole simulation process (including mesh generation, numerical solving, adaptive remeshing and visualization) need to be efficiently parallelized. Parallel computing is then a way to solve very large size problems (irrespective of the cost).

The unstructured mesh generation techniques commonly used are intrinsically scalar as they create one point or one element at a time. Parallelism can be achieved if the points to be inserted are sufficiently far apart (the neighborhoods associated with these points are distinct). As pointed out by [Shostko,Löhner-1992], *distance* is the enabling factor for parallelism. Moreover, parallel mesh generation is a tedious task as it requires the ability to decompose the computational domain into sub-regions that can be meshed separately on different processors. This is referred to as the *partitioning* stage.

3.6.1 Parallelism and meshing processes

The strategy of parallel mesh generation can be divided into two categories :

- the mesh generation method includes parallelism,
- the mesh generation process is parallel while based on a serial mesh generation method.

The first approach introduces parallelism at the mesh generation method level while the second consists in using a given meshing method in parallel. This approach leads to meshing each sub-domain separately after the definition of the various domain interfaces and after a mesh of these interfaces have been completed.

The second approach avoids the specific tests about the boundaries of the different meshes (in terms of conformity). On the other hand, meshing the interfaces is relatively tedious [Shostko,Löhner-1992].

The mesh generation methods resulting in unstructured meshes as seen in this chapter are basically scalar methods. Indeed, in general, they allow the creation of one point or one element at a time. It is possible to include some degree of parallelism if the points they try to insert are sufficiently far away (in other words, when considering one point, there exists a certain neighborhood with no intersection with another one). Distance is then the key factor in meshing parallelism. On the other hand, a solution method based on a domain decomposition technique requires the construction of the meshes of several sub-domains whose union is a covering-up of the whole domain. Then, each sub-domain is dealt with in one processor. One of the difficulties is therefore to transfer the data values associated with one sub-domain (and then belonging to one processor) to another sub-domain (another processor). The efficiency of the method then depends on the load balancing between the different processors (Chapter 23).

3.6.2 Domain decomposition

The aim of domain partitioning is to minimize the amount of inter-processor communication as well as to balance the computational load per processor. The partitioning process can be subdivided into three classes depending on how the domain is split [deCougny-1997] :

- partitioning of an initial (coarse) mesh [Wu,Houstis-1996],
- partitioning of the domain (and not a boundary mesh) using a spatial decomposition method [Saxena,Perucchio-1992],
- direct partitioning (also called pre-partitioning) of the mesh of the domain boundaries (e.g. surface mesh in three dimensions) [Galtier,George-1996].

Once the partitioning has been completed, sub-domain meshing is performed in parallel with or without inter-processor communication.

***A posteriori* partitioning.** Provided with a mesh of the domain under interest, an *a posteriori* partitioning method consists in splitting this mesh into several