Chapter 1

Data and Basic Techniques

COPYRIGHTED MATERIAL

The concept of data structure has several meanings. Within the context of this book, three meanings will be examined. We shall therefore examine what *basic data structures* are, and how abundantly they are used in the development of any algorithm and, also, as the very components of other higher level structures. As indicated, algorithms rely on these structures, but make extensive use of so-called basic techniques that we are also going to discuss. Internal and external structures will also be considered. *Internal structures* depend specifically on the algorithm and are only known to it, whereas *external structures* are used as interfaces between algorithms (software programs) and are thus known to (and usable by) all players of the computational chain (meshers, solvers, visualization tools, etc.). Hereafter, the way that these structures are implemented is not in question, but it will be seen that, more often than not, one or more arrays (a particularly simple structure) are employed.

Concerning basic structures, rather than giving an abstract description, which can be found in any good reference, we shall make an effort to consider the problem from a practical point of view, linking to questions about meshes and their utilization in numerical simulations. Within our specific context, a good understanding of what these structures are and what they are capable of doing when coupled with a few *basic techniques* can provide ideas for different contexts (in fact, this is somewhat the purpose of the presentation, to show that the structures and techniques used here with positive results can, with the same results, be used in other areas).

For external data structures, we shall briefly present the structures that we use to store meshes and solution fields, indicating that there is an existing library capable of manipulating them, accessible to all.

1.1. Basic data structures and basic techniques

These structures range from a totally basic level (a simple single-index array) to significantly higher levels of sophistication. Operations with the structures are based on a number (quite small)

of basic techniques. The algorithms are going to be built with the best use of these structures and techniques.

1.1.1. *Basic data structures*

It is virtually impossible to be exhaustive and there are numerous references on this subject, but we thought it might be useful to describe some of the basic structures intensively used in algorithms, starting with the simplest, the array. In a formal way, given a set of values (mainly integers and floating-point numbers[1]), a data structure is an organization in the memory allowing these values to be stored. With this organization, and according to its nature, access methods are associated (how the sixth value contained in an array can be accessed, how the next value is accessed, etc.) as well as manipulating methods (how value is removed or added to a list, etc.).

Before proceeding, let us indicate that the values stored in any such structure are often integers that characterize objects (a point, an edge, a face or a mesh, etc.) but do not only think of an array of coordinates or an array of solutions.

- Array

An *array* is a contiguous portion of memory in which values are stored successively one after the other. Accessing a value is direct, via its index (single-index array) or via its indices (vectors, matrices, etc.). In the case of a single-index array, the user sees the ith value of the array `Tab`, denoted by v_i, as $\boxed{v_i = Tab(i)}$ and the following value is given as: $v_{i+1} = Tab(i+1)$. In memory, if the array starts at address adr and if each value occupies b memory words, then v_i, the ith value, starts at address $adr + (i-1)b$.

The operations associated with an array are very simple and only relate to the reading of a value whose index is given, and the writing of a value V, at a given index, j, that is $\boxed{Tab(j) = V}$. The choice to carry out these readings and writings with any one such strategy will make it possible to build more elaborate data structures (see, immediately below, stacks, queues, etc.).

Before being used, an array must be sized by allocating a *size* to it. It is also necessary to know the starting index[2] (*start*) in the following and the end index (denoted *end*) or their equivalents in the case of several indices (this case will be revisited when we mention the use of such arrays on vector processors or in parallel). In many cases, an upper bound of the size can be estimated; in other cases, this is impossible and may lead to overflows[3] (therefore to more or less fatal errors) when a value is added to the array. This topic concerning sizing will be discussed further on.

1. A set of triangles, for example, will be described, for each triangle, by a set of integers and, for the coordinates of the vertices (nodes), by a set of floating-point numbers.

2. The first index is often set to 0 or 1 depending on the cultural habits of users and the language employed.

3. In other words: $end - start + 1 > size$.

• List

In a *list*, each value is associated with one (or two) pointers, which indicate(s) the index of the next (or previous) value or the indices of the next and previous values. This will then be referred to as a linked or doubly linked list. In practice, a list can be seen as an array with two or three fields (value, next link and previous link) or several arrays, one per field, which is an easier solution. This choice is the one retained and we denote by `Tab` the array of values, by `Prev` the array of the "previous" link and by `Next` that of the "next" link; then, if the ith value is at index j, that is $v_i = Tab(j)$, we have $v_{i-1} = Tab(Prev(j))$ and $v_{i+1} = Tab(Next(j))$.

The operations associated with a list are more elaborate and concern reads (to read a value) and writes (to add a value), but also deletions ("remove" a value). As mentioned below, it is assumed that start and end indices are known as well as the size corresponding to the list and one sets $Prev(start) = 0$, there is no predecessor, and $Next(end) = 0$, there is no next value.

Sequentially traversing the values is no longer trivial as it is in an array and is written as $v_{i+1} = Tab(Next(j))$ starting from index j of value v_i, as a process initiated by $i = start$ and $j = start$. This traversing operation is used to determine whether a given value is in the current list (and, if so, it should be added to the list). The list is iterated through its $Next(.)$ link field; if the value is found, the answer is obtained; if a $Next(.) = 0$ value is reached and this value is not the value sought for, then this value has not been found.

To add a value, one examines the last value v_{end} whose "previous" link does not exist (there is no "next" value and, by convention, $Next(end) = 0$). If V is the new value to be added, it is successively written that $end = end+1$, $Next(end-1) = end$, $Next(end) = 0$, $Prev(end) = end - 1$, $Tab(end) = V$.

The reverse operation, removing a value from the list, consists of ***breaking*** its links. Let j be the value index, it is successively written that $Next(Prev(j)) = Next(j)$ and $Prev(Next(j)) = Prev(j)$. The value is still there but no longer accessible when iterating through the links.

We shall see below (for example, for hashing) a clever and effective use of this list structure (then with a single link, $Next(.)$ to point to the next value).

Finally, to conclude on lists, we now give an illustration (indicating the three fields, the value or data, the two links and showing their evolution). Consider the following list ($end = 8$, there are eight values):

Value	v1	v4	v7	v2	v3	v8	v5	v6
Prev	0	5	8	1	4	3	2	7
Next	4	7	6	5	2	0	8	3

In this structure, a value is added at the end, namely $v9$ is the value, it naturally resides at position 9, in other words at position $end + 1$:

Value	v1	v4	v7	v2	v3	v8	v5	v6	v9
Prev	0	5	8	1	4	3	2	7	6
Next	4	7	6	5	2	9	8	3	0

The value $v4$ is then "removed"; thus it is no longer accessible when the linked list is traversed:

Value	v1	..	v7	v2	v3	v8	v5	v6	v9
Prev	0	5	8	1	4	3	5	7	6
Next	4	7	6	5	7	9	8	3	0

- Stacks and queues

These structures are nothing more than arrays whose manipulation (reading and writing) follows a particular strategy. A stack is a `LIFO`, last-in, first-out structure, whereas a queue is a `FIFO`, first-in, first-out structure. In practice, integers are going to be stored in one of these structures of integers that point to objects (points, edges, etc.) and the difference will reside in the way that the structure is filled up and emptied; in other words, on the one hand, the entities (hidden behind the integer indices) are not processed in the same order and, on the other hand, memory is not managed in the same way.

$\diamond$ For a *stack*, the index of the last element is conventionally not denoted end but top, as the top of the stack. We thus have a set of integers of allocated length (*size*), stored from 1 to top. Operations on this type of structure consist of extracting the value of the integer placed at index top and of processing the corresponding object (to this integer). The process performed may result in introducing one or more values (integers) into the stack or, conversely, not adding any value. Depending on the case it will follow that $top = top - 1$ (nothing is added) or a value is added to the index top (top is not altered) and, if further values are still to be added, they will be inserted at index $top = top + 1$, while the loop continues[4]. The purpose is to process every object described in the stack and processing ends when $top = 0$, the stack is empty. The steps of the operation (it is again assumed that the first index is $debut = 1$, choosing 0 only changes the terminal condition) are thus as follows:

4. Making sure that $top \leq size$.

Use of a stack [1.1]

(1) Process index top:
– do $top = top - 1$;
– if one or several values are to be added:
- do while: $top = top + 1$, insert the value at this index;
– otherwise, if $top = 0$, END;
– go to (1).

Let us give a simple example, a stack of 10 elements, denoted $1, ..., 10$.

```
 1 2 3 4 5 6 7 8 9 10
10 is processed and 11 and 12 are added, then:
 1 2 3 4 5 6 7 8 9 11 12
12 is processed and nothing is added, then:
 1 2 3 4 5 6 7 8 9 11
11 is processed and nothing is added, then:
 1 2 3 4 5 6 7 8 9
etc.
```

In the end, 1 is processed and if nothing is added, the operation is completed. A simple example concerns an edge subdivision algorithm. The indices of the edges of a mesh are stored in the stack (which are also described in an *ad hoc* structure), namely the edges are deemed too long. The operations on the stack are then started by taking the last edge; and if it is deemed too long, it is cut in half. The length of the two edges constructed is then calculated and according to the decision test, we pop ($top = top - 1$) or push by adjusting top if required (if one edge or both edges are to be stacked).

$\diamond$ For a *queue*, the index of the first element is obviously referred to as $start$ and that of the last by end (this is the end of the queue). We thus have a set of integers of allocated length (*size*), stored from 1 (or 0) to end. Operations on this type of structure consist of extracting the value of the integer placed at index $start$, then $start + 1$, etc., and of processing the corresponding object (to this integer). The process performed may result in introducing one or more values in the queue (integers) or, conversely, in not adding any values. The values to be added are placed at the end, therefore at index $end = end + 1$, as necessary. The aim is to process all the objects described in the queue and processing completes when the index $start$ reaches the index end. The steps of this operation are as follows, assuming $start = 1$:

Use of a queue [1.2]

(1) Process index $start$:

– do $start = start + 1$;

– if one or more values are to be added:
- do while: $end = end + 1$, insert the value at this index;

– otherwise, if $start > end$, END;

– go to (1).

Let us give a simple example, again, a queue of 10 elements, denoted $1, ..., 10$.

```
 1 2 3 4 5 6 7 8 9 10
1 is processed and 11 and 12 are added, then:
   2 3 4 5 6 7 8 9 10 11 12
2 is processed and nothing is added, then:
     3 4 5 6 7 8 9 10 11 12
3 is processed and nothing is added, then:
       4 5 6 7 8 9 10 11 12
etc.
```

In the end, there is only one element left, it is processed and if nothing else is added, the process is complete. If we return to the previous example of the edge subdivision algorithm, the indices of the edges (described in an *ad hoc* structure) of a mesh are stored in the queue, namely the edges deemed too long. The operations on the stack then initiate by taking the first edge; it is cut in half. The length of the two edges thus constructed is then calculated and according to the decision test, the next edge is addressed, or one or two of the new edges are added at the end of the queue before moving on to the next edge.

It is important to see that the choice of structure, to address the same algorithm, most certainly has an influence on the final result, on the cost of the process and the necessary memory resource[5].

• Grid

A grid[6] is a *spatial* structure in one, two, three, etc., dimensions that, in our case, essentially enable the localization of geometric data such as points and triangles. A grid, as seen below, is a cover of a domain by a set of cubes. The purpose is to distribute the data into these cubes, addressing only those deemed relevant, in order to accelerate a given process.

5. In a stack, the top (top) is used to add a value, while in a queue, the end index is simply incremented ($end = end + 1$).

6. This is also referred to as a “bucket”.

$\diamond$ An immediate application thereof is to quickly know which entities are close to a given entity. The simplest grid is a uniform grid (Figure 1.1), which covers the area of interest using cubes aligned with the axes, the cubes have a given size, δ in each direction ("small" segments, squares or cubes depending on the size of the space). In practice, δ is given and a number of cubes n is inferred, or this number (budget) n is given and δ is inferred. For some problems, it may prove interesting that cubes of different sizes are built depending on the direction, then one or more sizes δ_x, δ_y, etc., will be obtained, one per direction, therefore budgets n_x, n_y, etc. Regardless of the choice, it is very easy to find the cube that contains a given entity. We consider three dimensions and denote by $cube(i, j, k)$ the cube of index, the triplet (i, j, k). The *extrema* of the grid are determined and $(x_{min}, y_{min}, z_{min})$ denotes the bottom left corner of the latter and $(x_{max}, y_{max}, z_{max})$ its top right corner (and the cube thus defined is slightly dilated). One then has to find which cube contains a point of coordinates (x, y, z) (assumed to be within the grid, as it has been built for this purpose, see hereafter). It is trivially found that:

$$i = \frac{x - x_{min}}{\delta_x},\ j = \frac{y - y_{min}}{\delta_y} \text{ and } k = \frac{z - z_{min}}{\delta_z}.$$

To deepen the understanding of what can be done with a grid, we are going to consider a specific problem. Given a cloud of points (say a million points for clarity), how can a structure be built to make it easy to find points close to a given point. First, the points will be stored using a grid of a given size. This size, for example, $10 \times 10 \times 10$, necessarily implies that a given cube may contain several (possibly a significant number of) points. However, we can only do $cube(i, j, k) = v$, in other words, only one value v can be stored (here the index of a point P_v) per cube. The solution is to couple the grid with a (linked) list. The value v is therefore just the *entry point* in this list and it will serve to "contain" all the points of the cube under consideration. The resource requested is therefore the grid, 1,000 in our example, and the list whose size is, still in this example, 1,000,000, one link per point (the equivalent of the array $Next(.)$ seen above). With this device, all the points can be stored.

We set $cube(i, j, k) = 0$ for all the indices and the list initiated, denoted $Next(.)$ at 0 (there is no next), then the points are taken one by one. For a given point, of index w, its triplet (i, j, k) is found and the value $cube(i, j, k)$ is examined. If this value is zero, the cube is empty and one sets $cube(i, j, k) = w$. If $v = case(i, j, k)$ is not zero, the list will be iterated, starting at v, in the following way:

Insert an index w in a list [1.3]

Start $ind = w$.

(1) If $Next(ind) = 0$, then $Next(ind) = w$, END:

– otherwise, $ind = Next(ind)$, go to (1).

We now have the grid $cube(i, j, k)$ and the array $Next(.)$; we can thus come back to our subject, which is to find the points close to a given point for a given distance threshold, ε. The principle is simple, and we find the triplet (i, j, k) associated with the point being examined and $v = case(i, j, k)$ the index of a vertex whose link fields will be iterated. In this way, the cube (i, j, k) is examined but it is possible that it is empty ($v = 0$) and/or that there is a solution in

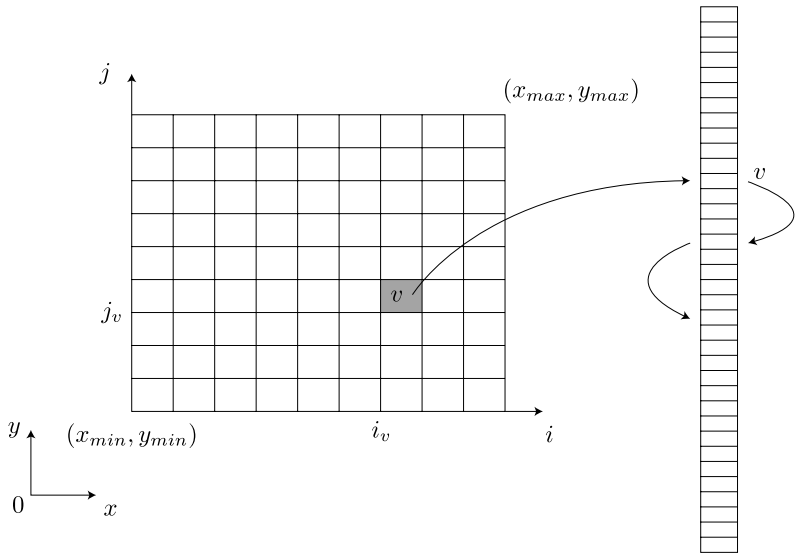

Figure 1.1. *A uniform grid (here in two dimensions) and the associated linked list*

a nearby cube. Therefore, we shall also look into, according to ε and sizes $delta_x$, $delta_y$, etc., the cubes neighboring the original cube likely to be relevant. To this end, the first index will *a priori* vary between $i - \Delta_i$ and $i + \Delta_i$, and the same is done for the other indices, with as a value for Δ_i, the appropriate integer, calculated based on δ_x and ε, that is $\Delta_i = \frac{\varepsilon}{\delta_x}$. Since the index, here the first one, must be comprised within the admissible range, one has to ascertain that $i - \Delta_i$ and $i + \Delta_i$ are within this range. The first possible index is 0, $x = x_{min}$ therefore $\dfrac{x - x_{min}}{\delta_x} = \dfrac{x_{min} - x_{min}}{\delta_x} = 0$, whereas the last is $\dfrac{x_{max} - x_{min}}{\delta_x} - 1$, where x_{max} is an upper bound for x. We thus set $i_{min} = 0$ and (with an integer calculation) $i_{max} = \dfrac{x_{max} - x_{min}}{\delta_x} - 1$ we have the constraint:

$$i - \Delta_i \geq i_{min} \quad \text{and} \quad i + \Delta_i \leq i_{max},$$

and thereof the two values are deduced:

$$\Delta_i^- = \min\left(i, \Delta_i\right) \quad \text{and} \quad \Delta_i^+ = \min\left(i_{max} - i, \Delta_i\right),$$

in conclusion, the first index will vary[7] between $i - \Delta_i^-$ and $i + \Delta_i^+$. The same occurs for the other indices. The following scheme can thus be proposed:

7. One could also have simply written a variation between $\max\left(i - \Delta_i, 0\right)$ and $\min\left(i + \Delta_i, i_{max}\right)$.

Find points in the neighborhood of a given point [1.4]

Build the grid $case(i, j, k)$ and its $Next(.)$ link fields.

Calculate the triplet (i, j, k) of point $P = (x, y, z)$ under examination:
– for $ind_i = i - \Delta_i^-, i + \Delta_i^+$, $ind_j = j - \Delta_j^-, j + \Delta_j^+$, $ind_k = k - \Delta_k^-, k + \Delta_k^+$:
- $v = case(ind_i, ind_j, ind_k) \rightarrow P_v$ the point of index v;
(1) If $||P - P_v|| \leq \varepsilon$, P_v is a solution:
- $v = Next(v)$, if $v \neq 0$, take P_v and go to (1);
– end for.

The points searched for are the P_v discovered in this algorithm.

◇ Another utilization of the same structure is the implementation of a quick *filter*. Given a number of points, find out if a (new) point is not too close to a given point (already stored in the structure). If that is the case, do not retain it (filter effect), but encode it into the structure. Note that the distance threshold, ε, depends on the point analyzed P, and each of the points (P_v already coded in the structure) against which it will be compared. With this in mind, ε should be written as $\varepsilon(P, P_v)$. The distance between P and a P_v is ideally computed within an underlying metric field (every point is associated with a size, this is an isotropic case) and the threshold of the filter can be expressed on the basis of a "unit" length (relative to this field). For instance, if the sizes at P and P_v are denoted by h and h_v, it will be possible to calculate the distance[8] in the field as $d_{h,h_v}(P, P_v) = 2\,\dfrac{d(P, P_v)}{h + h_v}$ or still $d^2_{h,h_v}(P, P_v) = \dfrac{d^2(P, P_v)}{h\,h_v}$ and a filtering criterion $d_{h,h_v}(P, P_v) \leq \varepsilon$ or $d^2_{h,h_v}(P, P_v) \leq \varepsilon^2$ will be employed with, now, a fixed threshold, for example $\frac{\sqrt{2}}{2}$.

An algorithm very similar to the previous one will be used with, in addition, a more subtle path, ideally in *spiral*, around the initial cube. The purpose of this is, in the event of a rejection, to detect more quickly a (the first) conflict situation and, thus, to stop exploring the points of the structure. Since the likelihood of finding a conflict decreases as one moves away from the cube in which the analyzed point falls, we shall start by analyzing the points identified in this cube before "revolving" around (unlike the previous case) by gradually moving away from it. It remains to be clarified which cubes should be analyzed, namely those likely to contain a conflicting point. It is assumed that the size h of the point P that is being filtered is known. The points P_v potentially in conflict are progressively discovered, but their size h_v comes to be known when such a point is discovered. As a result, how far the search should be extended remains *a priori* unknown (what is, along the width, the number of cubes to explore). Therefore, we shall rely only on the h of point P, with a security coefficient to assess the width of the area to be explored. On the other hand, the approximated computation of distances will be correct in accordance with the sizes involved.

8. It has been seen, on many occasions, in previous volumes, how a distance in a field can be calculated (approximated); we retain here an inexpensive and reasonable approximate formula for the present case.

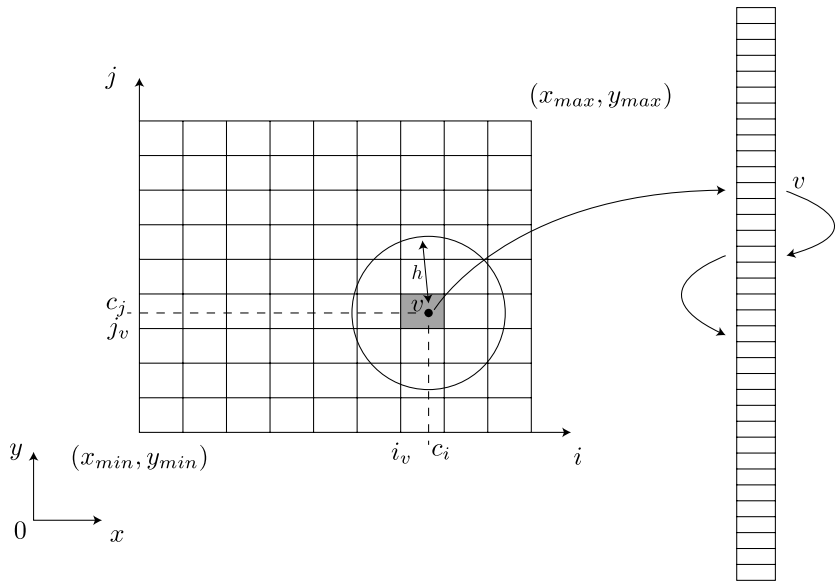

Figure 1.2. *A uniform grid (here in two dimensions) and the associated linked list. The point P (black circle) to be filtered with its size h and point P_v of the cube (supposedly non-empty) of indices, and those determined by P, P_v as the point acting as point of entry into the linked list*

The indices (i, j, k) (therefore integers) of point P of coordinates (x, y, z) are calculated, as well as (c_i, c_j, c_k), the relative coordinates of P. In more detail, we have: $c_i = \frac{x - x_{min}}{\delta_x}$ while $i = int(c_i)$ the integer immediately less than or equal to c_i. The half-width is estimated at i, that is $l_x = \frac{h}{\delta_x}$. It follows that the first index at i to be explored is $\min_i = int(\max(0, c_i - l_x))$ and that the last one is $\max_i = int(\min(c_i + l_x, i_{max})$. For other directions (at j and k), we have exactly the same approach. It now remains to determine how the set of cubes will thus be defined in order to minimize the number of computations. The trivial solution, which is the loop from $\min_i$ to $\max_i$ (likewise at j and k) is obviously ruled out.

The list in the example in Figure 1.2 focuses merely on the relevant cubes. Figure 1.3 shows two strategies, among many others, of possible paths. The strategy to be retained must both minimize costs and, obviously, be relatively easy to "program" (think of the three dimensions). The aforementioned trivial solution, although immediate in its written expression, is not a good choice. On the left, the distance between the initial cube and the other cubes to find the path seems optimal. Let us look first into the cube (i, j) and then its four neighbors per edge (the cubes $(i-1, j), (i+1, j), (i, j-1)$ and $(i, j+1)$). Then, we consider the four neighbors per corner (cubes $(i-1, j+1), (i-1, j-1), (i+1, j_1)$ and $(i+1, j+1)$); as such, we have just examined the *crown* of $rank$ 1. At the top level, the same idea is applied (at j the two cubes

involved, which are $(i-2, j)$ and $(i-2, j)$, etc., for $rank$ 2). It is clear that it is not easy to build this path (imagine the three dimensions). To the right of the figure, the trajectory is more simply defined, even if it is not entirely optimal. We look again first at cube (i, j) and then its neighbors in the first crown, $rank$ 1 (starting with $(i-1, j)$ for example and then rotating). Next, the adjacent (active) cubes of the second crown are taken, $rank$ 2 (starting with $(i-2, j)$ for example and then rotating). Then the following crowns are addressed if they are relevant. This strategy seems to be much easier to program[9].

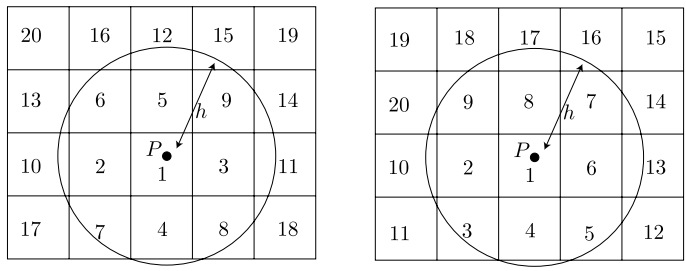

Figure 1.3. *Two possible paths around the cube containing point P (two-dimensional case)*

In short, the starting cube is located. If it is non-empty, we find the index v of the point that it contains. The point P is filtered with P_v; if there is conflict, the process terminates and P is set aside. Otherwise, the links associated with v are iterated through and P is filtered with the points encountered; any collision stops the filter and P is set aside. Then, if the process continues, the surrounding cubes are traversed following the pathway strategy defined. For each cube, if it is non-empty, the index v that it contains is found and the same process is repeated.

If, at the end of visiting all of the cubes deemed *a priori*, the point P has not been filtered, it is inserted in the grid, which is tantamount to either add it to its own cube (if it was empty) or to add it at the end of the list of that same cube. The following diagram will try to explain synthetically what has just been discussed:

9. And this will be left to really interested readers!

Filter a point [1.5]

Build the filtering grid $cube(i, j, k)$ and its $Next(.)$ link fields.

Compute the triplet (i, j, k) of point $P = (x, y, z)$ being examined. Set $v = cube(i, j, k)$.

If $v \neq 0$, take point P_v:

(1) if $||P - P_v|| \leq \varepsilon$, P_v is in conflict, END:
– $v = Next(v)$, if $v \neq 0$, take P_v and go to (1);

End if $v \neq 0$.

"Spiral" path following the crowns. Set $rank = 0$;

(2) $rang = rang + 1$.

Do for indices i, j and k of the "rank" crown:

Take $v = cube(i, j, k)$, if $v \neq 0$ take P_v;

(3) If $||P - P_v|| \leq \varepsilon$, P_v is in conflict, END:
– $v = Next(v)$, if $v \neq 0$, take P_v and go to (3).

End Do for indices.

Go to (2) while not completing.

Insert P into the filtering grid.

It should be noted that this type of filter is perfectly adequate for Delaunay-based triangulation (meshing) algorithms. It makes it possible to rule out (not to insert) a point too close to a vertex and it controls the size of the (current) edges of the mesh under construction. For a frontal-type method, such a filter avoids proposing a point that is too close to a vertex existing during the advance of the front. For tree-based methods, the tree itself serves as a filter (see below) thereby resorting to a grid is, in general, superfluous.

◇ For one last use, think of a mesh or re-mesh algorithm, to quickly find a (mesh) element containing a given point. The grid will then be used to initialize a localization algorithm. It is the triplet (i, j, k) associated with the point that is addressed and the index of a (current) mesh vertex $v = cube(i, j, k)$ is obtained. Each vertex of the mesh has to simply be associated with an element of the latter to initiate the localization algorithm (as seen in Volume 1 for simplicial meshes). The use of the grid makes it possible, in principle, to be relatively close to the solution and thus enhances the performance of the search.

Most of what has just been seen can be extended to the anisotropic case more or less. The geometry of the cubes is not really adaptable, but, on the other hand, the lengths can be evaluated in accordance with anisotropy.

Many other uses, in our field or in others, are possible. In conclusion, let us indicate that a grid, as described above, is by nature uniform (cubes with equal sizes defined according to directions) and that there is the risk that one (or more) cube(s) "contain(s)" a significant number of points (here), while others, if not most of the other cubes, are seldom filled up or even empty. If the distribution of points into the cubes is poorly balanced, most of the advantages of the grid will be lost. As a matter of fact, in such a case, the performance gain is reduced, the mere presence of a single cube containing almost all of the points (and this case is real) may lead to quadratic behavior (Figure 1.4). As a result, a more flexible and adaptive structure is to be considered, such as a *quadtree*- or *octree*-type tree (see below).

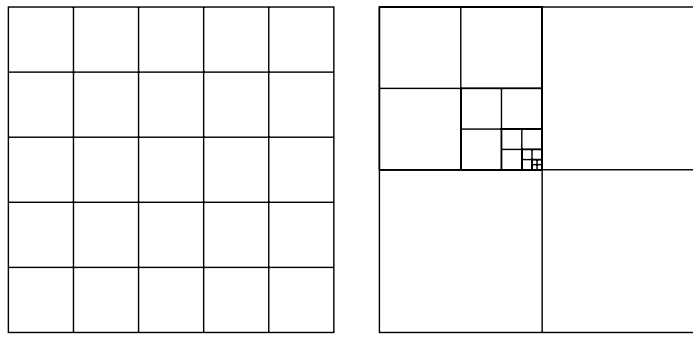

Figure 1.4. *The separation potential of a grid with* 25 *cubes and a tree of the same number of cubes. The smallest cube of the grid (of all in fact) has a size equal to* $\frac{1}{5}$*, that (or those) of the tree has (have) a size equal to* $\frac{1}{64}$*. This discrepancy is, naturally, the greatest possible, thus, in practice, it is not actually achieved. However, the value reached is better than that related to a grid*

◇ Hash grid: We may not effectively build the grid (thereby a two- or three-index array, which are memory-hungry) but describe it via a single index by way of a hash. One virtually has a grid such as hereafter and a point P is associated with its triplet (i, j, k) in a conventional manner. Then, a hashing function is used to build a single index, denoted by ijk. For example, let $ijk = i + j + k$ up to a *modulus*. In this way, the size of the grid is controlled, although the size of its cubes might be extremely small, giving a good degree of separation and an excellent resolution; the price to pay is the increase in length of the linked list that will take longer to iterate. This aspect (the resolution) will be explained in Chapter 2 when mesh reconnecting and merging problems will be addressed.

• Tree

Of all the possible types of trees, two of them can be distinguished, namely *quatree-* or *octree-*types of trees, that will be seen as spatial structures (similarly to the grids discussed above). The goal is the same here, to store, localize or filter points. The use of such a structure, more sophisticated than a simple grid, will allow, for a reasonable budget, that no cubes (of the quadrants or octants) be abnormally filled. Furthermore, in situations where the geometry is fine, the cubes are refined and otherwise, larger sized cubes are kept. It should also be noted that the conventional rule for balancing the tree, applied for use for meshing purposes (Volume 2, Chapters 4 and 5), is not required here. This will avoid behaviors of a quadratic nature by controlling the depth of the tree in order to control the maximal number of points linked to a given cube. In practice, this maximal number of points is chosen and the maximal depth is deduced therefrom as well as the maximal number of cubes in the tree (which will give the necessary memory budget).

To further describe this technique, two dimensions are considered, and thus adopt a quadtree and follow the very clear presentation given in the previously study [Löhner-2008]. A list of possible variations of this process will also be proposed. The tree is seen as a two-index array, namely an array denoted $quad(.,.)$. The second index is that of the quadrant. The first index can take several values and, according to them, it allows the history of the quadrant to be described and the points it contains to be referenced. Therefore, if no more than four points are tolerated[10] per quadrant, the array is written as $quad(1:7,i)$ and the precise meaning of $quad(j,i)$ is as follows:

– $v_7 = quad(7,i)$ indicates that the quadrant has been subdivided ($v_7 < 0$), is empty ($v_7 = 0$) or gives the number v_7 of points that it contains ($v_7 > 0$);

– $v_6 = quad(6,i)$ indicates the position of the quadrant i inside its parent v_5;

– $v_5 = quad(5,i)$ gives the index of the parent quadrant ($v_5 > 0$);

– $quad(1:4,i)$ is defined according to the value of v_7:

- if $v_7 > 0$, $quad(1:4,i)$ gives the indices of the points contained in the cube; a zero index indicates that the cube is not already saturated;

- if $v_7 < 0$, $quad(1:4,i)$ gives the indices of the 4 children of quadrant i.

Besides this array (of integers), an array (of real numbers) $bounds(.,.)$ will be defined that associates with each quadrant (octant), *aligned with the axes*, the minimum and maximum abscissa (the ordinate, and the height) of its vertices. In two dimensions, we thus have four values, denoted x_i, y_i, X_i, Y_i for quadrant number i, that is to say that $x_i = bounds(1,i)$, etc. In three dimensions, there will be six such values. These bounds will be used to locate a point given by finding the cube (cell, or leaf) in which it is contained.

10. A natural value since the quadrants are likely to be subdivided into four, but another value may be taken for example 10. The array will then be organized as $quad(1:13,i)$. For indices 13, 12 and 11, we find once more the analog of values 7, 6 and 5 as they are defined in the text and for indices 1 to 10, either at most the 10 points stored are found or only the four children. In three dimensions, for an *octree*, the natural value is obviously 8.

All of this information will be used to build (with the given points) the tree and then to use it to carry out operations related to the considered process (localization of a point, insertion of a new point and filter of a point relatively to the existing points in the tree).

Let us indicate that there are other ways to define a tree, in particular, by storing only what is strictly necessary in the structure itself and by adding pointers. Examples of this alternative construction method will be discussed below.

◇ Localization of a point

Consider the structure with its two arrays. To find the cube containing a point, the tree has to be iterated starting from its root and *descending* through its various levels down to the desired *depth* corresponding to the solution cube. The array $bounds(.,.)$ is used to know whether the point being examined is inside a given cube. The array[11] $quad$ then makes it possible, if necessary thus if $v_7 < 0$, to travel downwards in the structure via one of the values $quad(1:4,.)$ to find the cube at the terminal level containing the dot. This gives a particularly simple algorithm. Starting from the root, it searches in which of the (4) children the point is located and the process iterates as long as it is incomplete. This gives the following steps, denoting $P = (x, y)$ the coordinates of the point being analyzed.

Localization of a point [1.6]

$iel = 1$

– (1) if $quad(7, iel) < 0$, if among the four children, $quad(1:4, iel)$, find the one that contains P:

- recover $x_{iel} = bounds(1, iel)$, $X_{iel} = bounds(2, iel)$
  $y_{iel} = bounds(3, iel)$ and $Y_{iel} = bounds(4, iel)$ then;
- if $x_{iel} \leq x < \frac{x_{iel}+X_{iel}}{2}$, two cubes are discarded;
- if $y_{iel} \leq y < \frac{y_{iel}+Y_{iel}}{2}$, a single cube, i, is retained, do $iel = i$ and go to (1).

– The solution is cube iel, END.

It should be immediately noted that testing the range of x and y requires only two operations (three in three dimensions). We also observe that a problem of inaccuracy in the computation does not really have any consequence; in the worst case, the wrong cube is found.

◇ Insertion of a point

To insert a point in the structure, it has first to be localized: this is the previous algorithm. Let i be the quadrant index found, and the insertion of the point (denoted e in Figure 1.6) is done according to the following scheme.

11. Or its three-dimensional equivalent.

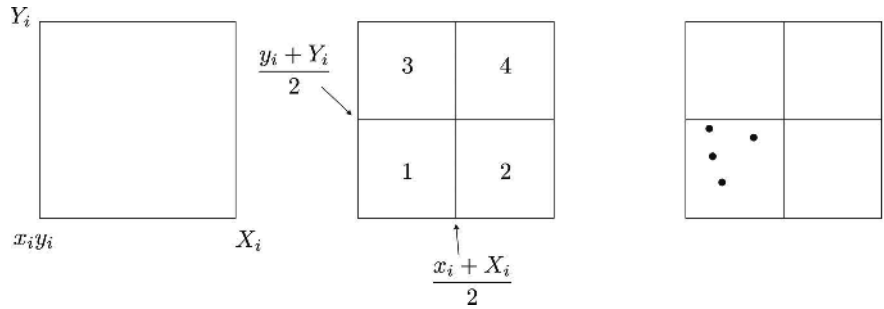

Figure 1.5. *A cube with its bounds, its four children with the new bounds and the local numbering of its children. On the right, the four points "fall" into the first child*

Insertion of a point [1.7]

– (1) if $quad(7, i) = 4$ (the quadrant is saturated):
- subdivide the quadrant into 4, rank the 4 points in the desired child;
- localize the point e in the desired child, let i be its index, go to (1).

– Otherwise $quad(7, i) = quad(7, i) + 1$ and $quad(quad(7, i), i) = e$, END.

Figure 1.6 shows how to insert into the structure, a point that falls into a saturated quadrant that will have to be subdivided, this is the case $quad(7, i) = 4$ of the scheme where i is the index of this quadrant.

The memory is intended for m cubes and in the array $quad$, *a minima* was initialized $(7, 1 : m) = 0$ during the construction phase of the tree (see hereafter). It is assumed that n cubes have already been created; the first free cube is thus that of index $n + 1$. The cube, of index i, containing the point is saturated, as such it must be subdivided before attempting to insert the point.

The children will therefore be stored at the first free indices, that is $n+1, n+2, n+3$ and $n+4$. The cube of index i should forward to these indices, namely $quad(1, i) = n+1, ..., quad(4, i) = n+4$ and should marked as subdivided, that is $quad(7, i) = -1$. In the reverse direction, children must point to their parent, namely $quad(5, n + 1) = i, ..., quad(5, n + 4) = i$. Then, we localize points a, b, c and d in the desired children. In the example in the figure, the cube $n + 1$ contains a and therefore $quad(7, n + 1) = 1$, the next cube is empty, the next one contains b and c, thus $quad(7, n + 3) = 2$, etc. The array $bounds(., .)$ is updated in parallel. Once done, the point e is then inserted, it falls into the cube $n + 2$ which is unsaturated, and it can thus accommodate it.

◇ Construction of a tree

With the two operations above (localization and insertion), it will be shown how to build a tree to store a cloud of given points.

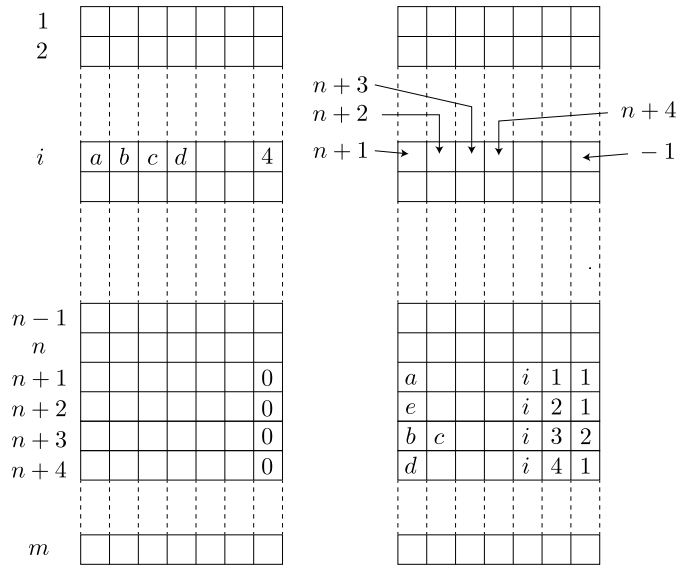

Figure 1.6. *Updated array $quad(.,.)$ related to the attempt to insert a point, denoted e, in a saturated quadrant, of index i (already containing points of index a, b, c and d)*

The root (the first quadrant (octant)) is defined from *extrema* $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ slightly dilated. This root might be a square (cube) or a rectangle (parallelepiped). The quadrants (octants) will therefore be squares (cubes) or not. For the subdivision of cubes, we shall rely on the middle of the edges. The construction of the tree consists of defining its cubes down to the desired level and to associate them with their bounds (the arrays $quad(.,.)$ and $bounds(.,.)$), while satisfying the constraint: no more than four points per cube. We start by setting $quad(7, i) = 0$ for all the expected values for i (from 1 to m). The root is initialized:

– $i = 1$, $quad(1:7, i) = 0$;

– $bounds(1, i) = x_{min}, bounds(2, i) = x_{max}, bounds(3, i) = y_{min}, bounds(4, i) = y_{max}$.

Then, the insertion algorithm is sufficient to build the tree, that is simply:

Construction of the tree [1.8]

– Do for all points of the cloud:
  - Insert the point using Algorithm [1.7].
– End do.

The index of the quadrants is found during subdivisions by a simple increment of 1 in the numbering order seen above (Figure 1.5 in the middle).

◇ Filtering of a point

Similarly to, and for a comparison with, a grid, it will be shown how to filter a point relatively to a set of points stored in a tree structure. With each point a size h is associated and the relevant quadrants of the tree will be visited to find out if the point under examination $P = (x, y)$ is too close to a given point for a threshold ε. Except for a few details, the same method[12] allows the points in a certain neighborhood of a given point to be found. The idea is similar to a grid to locate the cube containing the point being analyzed and then to examine, if necessary, the neighboring cubes in a given neighborhood. The cubes neighboring a given cube are of several types depending on their degree of "kinship". *Siblings* will be found (derived from the subdivision of the same cube) as well as more or less distant *cousins* (derived from the subdivision of siblings or cousins of the parent). In other words, it will be necessary to travel through the tree, *a priori* in both directions, downwards (moving from a child to the child of a child, etc.) or upwards (moving from a parent to the parent of a parent, etc.). Stepping downwards or upwards into the tree is trivial via either $quad(1:4, i)$ or $quad(5, i)$.

Filter a point or find the points located within a given neighborhood [1.9]

– Locate the point P in the tree, that is, i the index of the cube found.
– Examine cube i and depending on the case:
  - for a filter:
    . for the P_v discovered in the cube, if $||P - P_v|| \leq \varepsilon$, the point is filtered, END;
  - to find a neighborhood:
    . for the P_v discovered in the cube, if $||P - P_v|| \leq \varepsilon$, the point P_v is a solution.
– Move to relevant neighboring cube, while not terminating.

We have the same remark as for a grid, if the h of the point analyzed is known. Those of the points, P, discovered in the tree are not known until the time of their discovery, hence a coefficient of security should be provided. Nevertheless, the tree was built from supposedly consistent points (in terms of their size h) and, in addition, the risk of having a cube too full is prevented by the subdivision rule for the saturation criterion and, even if several neighboring cubes are impacted, the number of points (and therefore of computations) to be examined is lower than in the case of a grid.

Figures 1.7 and 1.8 try to illustrate the utilization of a tree for filtering a point given with respect to the points stored in the structure. Geometrically, the domain surrounds a central object. The outer boundary is very coarsely meshed while the boundary of the central object is finely meshed. The result is the tree shown in Figure 1.7 (on the left). A magnification of the central region is shown on the right side of the figure. The point P to be filtered can be seen, and is

12. If ε defines the neighborhood, the same algorithm is used by fixing $h = 1$ for every point.

represented by a red circle. Its size h determines the area to be analyzed by the red circle, the encompassing box of this circle was also added (aligned with the axes).

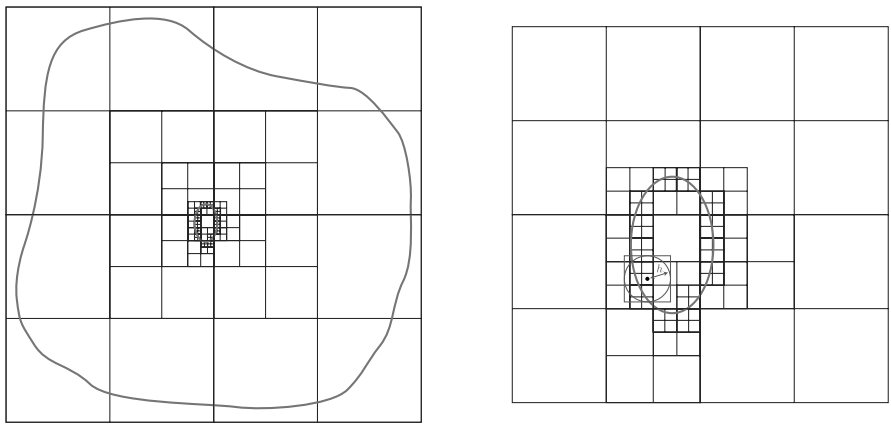

Figure 1.7. *On the left, in blue, the boundary of the domain with its two connected components and the associated tree. On the right side, the central object is enlarged, the point to be filtered (red circle) is the center of the circle of radius h*

There is a case where to travel from a cube to one of its neighbors, it is necessary to go back to the root and then move downwards into a another branch of the tree, but this (long) journey can be avoided using a simple trick. To this end, the depth p of the tree has to be known (see immediately below). This depth makes it possible to calculate the size (sizes) of the smallest cube in the tree, namely a factor $\dfrac{1}{2^p}$ of $x_{max} - x_{min}$ and $y_{max} - y_{min}$. The two increments are then defined $\delta_x = \dfrac{x_{max} - x_{min}}{2^{p+1}}$ and $\delta_y = \dfrac{y_{max} - y_{min}}{2^{p+1}}$. From of the bounds of the cube and its position in its parent, in the middle of Figure 1.5, rather than moving up and down in the tree, a location query will be initiated on a fictitious point. For example, to find the cube to the right of the cube numbered 2 of bounds (x_i, X_i, y_i, Y_i), we shall build as a fictitious point the point $(X_i + \delta_x, y_i + \delta_y)$. Similarly, to find the left neighbor of the cube numbered 1, we build as a fictitious point the point $(x_i - \delta_x, y_i + \delta_y)$, etc.

Before continuing, let us figure out how to calculate the depth of the tree. It is necessary to associate an additional value[13] to the cubes, their depth. Therefore, let $p_i = quad(8, i)$ be the depth of the cube i. We initialize $p_1 = 0$ for the root and set $p = 0$. At each subdivision, we have $p_{child} = p_{parent} + 1$ and the depth p is adjusted as $p = \max(p, p_{child})$.

In the case of a grid, to determine which cubes to examine and to define how to travel them, the width of the relevant area had been calculated and a spiral trajectory constructed (or almost).

13. For example, by giving the array a size of eight values in the first field.

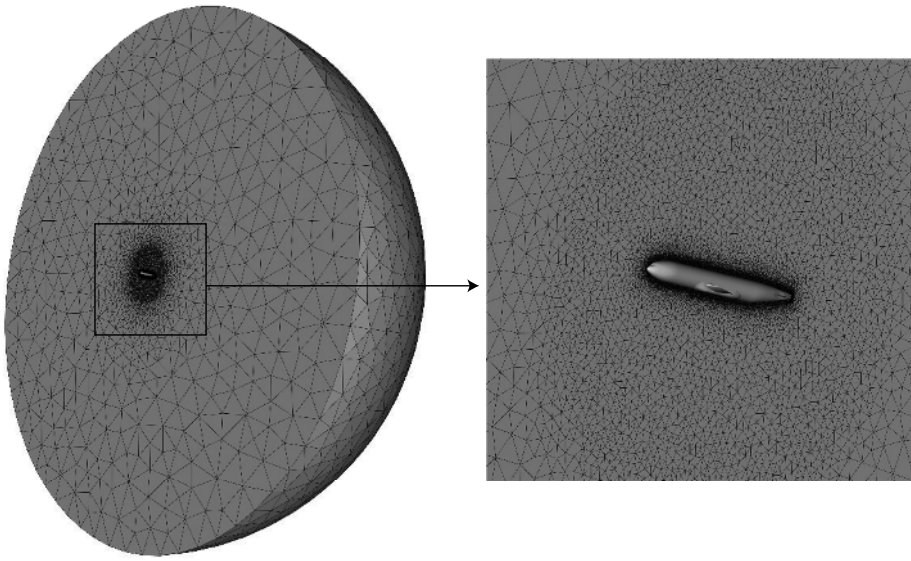

Figure 1.8. *A real example (the tree of which would be close to that of the previous figure) where a large number of points (from the boundary of a domain) is located inside a very small part of the volume. For a grid, one (or a few) cube(s) is (are) largely full, and the others (although of the same size) being almost empty. For a tree, the cubes really utilized (and with adapted size) will be in the neighborhood of this half-plane (right-side enlargement) in its cube (visible on the left)*

The regular structure (the grid) had been used to define this trajectory. With a tree, we are going to look at how these operations are implemented. The non-regular structure of the tree, the existence of (geometrically) neighboring quadrants, but of different sizes (different levels) that are not topologically neighbors (sharing no common edges), makes the spiral path more difficult around a given quadrant. It should be possible to navigate the tree upwards, travel through a neighbor, at this level, and move downwards. In order to avoid these operations, we shall propose to rely only on the localization algorithm. A quadrant is considered and it is virtually associated with a circle whose center is the barycenter of the quadrant and radius, a length in accordance with the size of the quadrant. By increasing this radius, the circle will intersect the quadrants in the vicinity of the initial quadrant. The circle should simply be sampled with a few points. Its points are then localized in the tree and the quadrant that contains them is determined; this quadrant is intended to be processed (by referring to the vertices that it points to). To avoid addressing the same quadrant multiple times, a coloring is used. Indeed, the fact that the radius increases does not mean that the affected quadrants are always different, a large quadrant could be found for several radius values. It should be noted that this method has the advantage of solving the problem of finding neighboring cubes and that neighborhood information between sibling quadrants does not need to be stored.

Bear in mind that the encompassing cube can be defined (aligned with the axes) and associated with the circle (sphere) being searched for. This can accelerate the computations. A point in an *a priori* relevant cube can be evacuated in a single simpler computation, the difference in x (in y) with this cube.

◇ Filtering or insertion of a point

In Delaunay-based methods or frontal methods, points are proposed in order to be inserted. These points have to be filtered (discarded) or selected as candidates for the insertion. We then follow the algorithm above for this specific purpose.

In Delaunay-based methods, the insertion of a point needs to quickly find the element in which it is situated. Its localization in a tree cube makes it possible to find a vertex of the mesh being built. An element should then be associated with every vertex to initialize the search process with the element associated with the vertex found; this is the notion of *seed* which we will revisit.

◇ Anisotropy

Similarly to a grid, a tree is inherently isotropic. On the other hand, distance calculations can be carried out by taking into account an anisotropic field.

◇ Immediate or more advanced variants for the definition of a tree

The definition of a *quadtree*, as seen above, can be made significantly more compact and therefore less memory greedy and thus makes it possible to store on a laptop computer a tree containing dozens or hundreds of millions of cells.

We consider the situation where only downwards searches are performed in the tree. This avoids having to memorize parents. Next, the coordinates of the cell vertices are not stored (the array $bounds(.,.)$). They will be calculated on the flight at the expense of redundant calculations, but with significant memory gain. For the purpose of further compacting the structure, a minimum amount of information will be stored in every quadrant. These are of two types, intermediate quadrants and terminal quadrants. A intermediate quadrant should only allow access to its children. Only one index is needed that points to the first child, the others being consecutively stored in memory. Terminal quadrants (which do not have children) allow effective access to the entities (for example, vertices) stored in the tree via, once more, a single index. In the event that several entities are to be stored in the tree, they will be stored consecutively in the memory starting from that first index. The same index, therefore only one, can contain the information. If the index is positive, the quadrant is terminal and the index points to the entities are stored. If the index is negative, the quadrant is intermediate and the absolute value of the index points to the first child of the cell.

Everything that has just been seen in two dimensions can be extended to three dimensions.

This section ends by mentioning a problem mentioned above for the arrays, but also common to all these structures, which is the case where their initial sizing proves to be insufficient. For

example, an array has been sized to store up to 1,000 values and storage for another value is required. This point will be discussed in section 1.1.2 where several solutions will be proposed.

1.1.2. *Basic techniques*

It is impossible to be exhaustive, here too, but it is useful to describe some of the basic techniques[14] used intensively in algorithms (here rather destined to meshes). In our experience, we have observed the role played by a reduced number, all in all, of techniques which, for the most part, are surprisingly simple and, in principle, (should be) known to everyone.

• Coloring and dynamic coloring

The purpose of coloring is to quickly know the status of an entity (a point, an element, etc.) vis-a-vis a given situation. To find out if a point has undergone any processing (a move, etc.) or if an element belongs to a certain set (a cavity, a sub-domain, etc.), it can be assigned a color, c, via an array (a marker) of colors. In these two examples, a Boolean number (0 or 1, true or false, yes or no) can be used, more precisely, two colors only, but it remains to be known whether it has been updated if the entity marked is modified. To clarify, reconsider the example of the construction of a queue, Algorithm (1.2), by stipulating a few rules. The aim is to find a list of elements in a mesh that do not verify Delaunay's criterion vis-a-vis a point. One starts by identifying the element, iel, containing this point and then builds, per neighborhood, the queue sought for, the array $Tab(.)$. Thereby we set $start = end = 1$ and $Tab(start) = iel$ and the following algorithm is unwound:

Construction of a queue of elements [1.10]

(1) Process the element of index $start$:

– do $start = start + 1$;

– if one or more neighbors are to be added:

- do while: $end = end + 1$, insert the element at this index;

– otherwise, if $start > end$, END;

– go to (1).

A subtlety is hidden in this algorithm. Since it is proceeding by neighborhoods, a neighbor of an element may already be in the queue. It is therefore necessary to verify this case and to not consider (storing in the queue) such elements[15]. The simplest method is to color the elements to indicate if they are or are not already in the queue. First method, a Boolean, the elements already taken into account are marked with the value 1, the others take the value 0. By simply looking into this value (color), it can be seen if a particular neighbor must be considered. However, at the end of the construction and to achieve another one, the situation has to be to re-established,

14. Some of which have already been described in Volumes 1 and 2.

15. Except for compressing the queue, if it has not burst in the meantime, during postprocessing to eliminate duplicates.

namely to reset the elements that have been marked with a 1, to 0. Thus, a color array is needed (0 or 1) and an array that memorizes the numbers of the elements whose color has been modified. We call $Mark(.)$ and $Modi(.)$ these two arrays, set $i = 0$ and initialize $Mark(.)$ to 0, then the above algorithm becomes clearer in (with, as above $start = end = 1$ and $Tab(start) = iel$):

Construction of a queue of elements with boolean coloring [1.11]

(1) Process the element of number iel and index $start$ in the queue and mark it: $Mark(iel) = 1$, then do $i = i + 1$ and $Modi(i) = iel$.

– Do $debut = debut + 1$.

– If one or more neighbors are to be added:

- do while for iel the number of the neighbor: if $Mark(iel) = 0$, $end = end + 1$, insert the element at this index.

– Otherwise, if $start > end$, END.

– Go to (1).

– Reset $Mark(.)$: Do for $j = 1, i$, $Mark(Modi(j)) = 0$.

The lines in red indicate how coloring is achieved. The algorithm is correct, but has an obvious weakness, the need to manage the array $Modi(.)$. To tackle this problem, we shall resort to dynamic coloring. The same algorithm unfolds by initially setting $c = 0$ and $Mark(.) = c$, where c is seen as a color, then the following algorithm is obtained:

Construction of a queue of elements with dynamic coloring [1.12]

– Do $c = c + 1$.

(1) Process the element of number iel and index $start$ in the queue and mark it: $Mark(iel) = c$.

– Do $start = start + 1$.

– If one or more neighbors are to be added:

- do while for iel the number of the neighbor: if $Mark(iel) \neq c$, $end = end + 1$, insert the element at this index.

– Otherwise, if $start > end$, END.

– Go to (1).

Again, the lines in red indicate how the coloring is used. Updating array $Mark(.)$ is implicit, therefore unnecessary; the array $Modi(.)$, on the other hand, has become useless, the algorithm is easier, and therefore faster.

With this example, which can be applied to many situations, we have shown how a simple technique, dynamic coloring, can be beneficial.

• Resorting to randomness

This is still a very simple technique that, when used, enables the performance to be improved (speed and/or efficiency) in many algorithms. A random choice is applied to choose in which order[16] the entities (for example points) whose indices are contained in an array will be addressed. This technique also allows choosing (randomly therefore) the process to apply in case there are several possible choices.

In the first case, it must be taken into account that access to memory can penalize performance since we are going to "hit" the memory almost everywhere and certainly not in a sequential manner. As such, a good tradeoff has to be found between the expected improvement and possible cache misses that might occur.

Finally, it should be kept in mind that, although they comprise of random aspects, algorithms remain deterministic.

• Sorting

The sorting that will take place in our domain mainly concerns lengths (of edges) and qualities (of elements). Sorting therefore involves floats referred to by an array of indices and can be used to order these indices so that corresponding floats be ordered, for the chosen *criterion*, from the smallest to the largest or the other way around. Thereby, the first index points to the smallest value (actually to the corresponding entity), and the last to the largest or vice versa. There are a number of sorting methods and references on this subject are numerous (any computer science-related course should be a good start). Rather than paraphrasing any such source, we would rather give *in extenso* a sorting program that we use frequently, in `Fortran` then in `C`; the sorting is carried out in ascending order for the given criterion (array CRITER or `criter` depending on the case and notation conventions).

We thus give these two programs and the reader is left[17] to understand the hidden mechanics of this particular sort that exchanges values. It should be noted that the array (of indices) as well as that of the values are updated during the process.

◇ A sorting program written in `Fortran`

```
      SUBROUTINE TRIRE3(CRITER,ARRAY,N)
C  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
C  GOAL: SORT THE ARRAY ARRAY(1:N) ACCORDING TO CRITER(1:N)
C  ---
C  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      INTEGER N,ARRAY(N)
      REAL    CRITER(N)
      INTEGER I,L,R,J,TAB
      REAL    CRIT
```

16. So to speak, since there is none.
17. Pencil in hand!

```
C
      IF ( N .EQ. 1 ) RETURN
C
      L = N / 2 + 1
      R = N
    2 IF ( L .LE. 1 ) GO TO 20
      L    = L - 1
      CRIT = CRITER(L)
      TAB  = ARRAY(L)
      GO TO 3
   20 CRIT       = CRITER(R)
      TAB        = ARRAY(R)
      CRITER(R)  = CRITER(1)
      ARRAY(R) = ARRAY(1)
      R = R - 1
      IF ( R .EQ. 1 ) GO TO 10
    3 J = L
    4 I = J
      J = 2 * J
      IF ( J .LT. R ) THEN
        GO TO 5
      ELSE IF ( J .EQ. R ) THEN
        GO TO 6
      ELSE
        GO TO 8
      END IF
    5 IF ( CRITER(J) .LT. CRITER(J+1) ) J = J + 1
    6 IF ( CRIT      .GE. CRITER(J) ) GO TO 8
      CRITER(I)  = CRITER(J)
      ARRAY(I) = ARRAY(J)
      GO TO 4
    8 CRITER(I)  = CRIT
      ARRAY(I) = TAB
      GO TO 2
   10 CRITER(1)  = CRIT
      ARRAY(1) = TAB
      END
```

◇ The same sorting program written in C

```
void heap(int siz, double *criter, int *array)
{
  int    i,j,r,l,tab;
  double crit;

  if ( siz <= 1 )  return;


  l = siz/2+1;
  r = siz;
```

```
  while ( r != 1 ) {
    if ( l > 1 ) {
      //--- save state l
      l       = l-1;
      crit = criter[l];
      tab  = array[l];
    }
    else {
      //--- save state r and put 1 in r
      crit       = criter[r];
      tab        = array[r];
      criter[r]  = criter[1];
      array[r] = array[1];
      r = r-1;
      if ( r == 1 ) break;
    }
    j = l;
    i = j;
    j = 2*j;
    while ( j <= r ) {
      if ( j != r ) {
        if ( criter[j] < criter[j+1] )
        j = j+1;
      }
      if ( crit < criter[j] ) {
        criter[i] = criter[j];
        array[i]  = array[j];
        i = j;
        j = 2*j;
      }
      else
break;
    }
    criter[i]   = crit;
    array[i]  = tab;
  }
  criter[1]   = crit;
  array[1]  = tab;
  return;
}
```

One might ask how can a sort be parallelized in an attempt to see how the program aforementioned must be transformed, with what effort and at the cost of which algorithm complexification (which is already not very clear sequentially).

- Renumbering

Some methods for renumbering nodes or mesh elements are presented in Chapter 3, where the goal is to optimize certain properties in matrices built to perform calculations, using the finite element method. It should not be considered as a basic technique; this is an algorithm in its own.

On the other hand, renumbering methods based on filling curves are indeed basic techniques that can be used in many algorithms. Chapters 4 and 5 of Volume 1 and Chapter 9 of Volume 2 to which we refer the reader cover their relevance. Nonetheless, we shall return to this point in Chapter 3 of this volume.

One should already recall the fact that the latter type of renumbering can be used to obtain at the same time a random part and a sequential part (via the underlying sorting) in processing the data thus renumbered.

• Dynamic threshold

Making use of sorting allows data to be processed in a specific order. If the sorting criterion is a length, think of edges, sorting will provide a means to consider the edges from the smallest to the largest or conversely. If the sorting criterion is a quality (in elements), it will be possible to address the elements from the worst to the best. In this example as in the previous case, the use of a *dynamic threshold* is an alternative (which, as a result, avoids a sorting).

The principle is simple; if ε is the target threshold, the entities will be processed in several stages with an initial threshold set to $\alpha\,\varepsilon$ with $\alpha > 1$, then, over the stages, the value of α will be reduced until it reaches $\alpha = 1$. This strategy often pays off, in terms of efficiency and time. With regard to the criterion, the worst entities are first addressed and, in overall, the gain is significant, more significant than if the entities were considered from the first to the last.

• Hashing

A recurring question is to know whether any entity, an edge (a segment), one face (a triangle, etc.) is present in an array (of entities) of edges, faces, etc. Since an edge is defined by a couple of indices and a face by such a triplet, the question becomes: is there a couple (a triplet) given in an array of couples (triplets)? A direct method is to compare the couple (triplet) with every couple (triplets). This method is quadratic and therefore inadmissible. The *hashing* provides a solution which, in practice, is somewhat linear. Another application (already seen in Volume 1) is to perform a hashing to quickly construct the array of edges (faces) of a mesh, which is a broader problem than the previous one, but that includes it. The hashing can also be used as a quick means to establish the neighborhood relationships (by edges or faces, depending on the space dimension) between the elements of a mesh by constructing the edge (faces) array for this purpose[18]. Beside these examples, there are obviously many other applications for hashing[19].

Hashing has been extensively described in Volume 1 (Chapter 4), consequently only a quick description will be carried out here. Consider the example of the construction of the array of edges in a mesh. The point is therefore to manipulate pairs of indices (those of the ends of each edge). The question is to find out if a given couple is already in the array or needs to be added, without having to iterate the whole array and compare, one by one, the pairs of indices.

18. In fact, a bit more information has to be memorized than for the simple construction of this array.

19. A widespread idea in teams is that anything and everything can be hashed to full advantage; this will be seen further on.

To do this, any pair (i, j) will be associated with two values, one *key* and a *failsafe value*. The simplest key is $key_{ij} = i + j$, a possible failsafe value is the integer $\min(i, j)$. Two identical couples (i_1, j_1) and (i_2, j_2) with therefore, $i_1 = i_2$ is $j_1 = j_2$ or $i_1 = j_2$ and $j_1 = i_2$ (if we have the notion of orientation) necessarily have the same key (and the same failsafe value). On the other hand, two couples having the same key are not necessarily the same, we just have $i_1 + j_1 = i_2 + j_2$. As a consequence, in order to find identical couples, all of the couples with the same key have to be iterated by verifying the value of the failsafe value at the same time. As several couples may have the same key, it is necessary to define a linked list. The key is the entry point in this linked array and iterating over the links allows the decision. The couple is found, and the edge may already exist in the array; otherwise, it has to be added to the array (it should be noted that, in this way, we are counting the edges).

The sizing of the arrays requires knowing the number of edges (or an upper bound) for the edge array strictly speaking and depends on the key function for the link array. The choice of the key function depends equally on the size of the array, and on its filling rate as well as on the number of collisions (thus the length of the chaining for a given key value).

• Resizing a resource (now insufficient)

To unroll an algorithm, the necessary memory resources must be allocated, namely by properly sizing the arrays (regardless of how, in the end, they are implemented) that are used. For some arrays, a good estimate can be obtained, namely an upper bound or (we know) the exact value of the necessary size. For others, this is not the case. However, even if in general, we have a more or less accurate idea of the necessary resource, there is always the possibility of overshooting what was predicted. What can be done concerning the way arrays are allocated? Two schools can be identified, one, slightly old perhaps, where only a single array, known as a *super-array* is allocated, via the allocation system, is available (in the language being used, `Malloc` in C) and then the useful arrays are managed by "hand" (which is tantamount to defining their sizes and (starting) addresses in the super-array) and the other in which each array is allocated via the allocation system. Independent of the method for managing memory, any program normally constituted must detect any overflow, avoid it and properly terminate[20] by notifying the user (by means of a message or an error code).

It should be noted that the perception of a problem of insufficient sizing is not the same for everyone and depends on the context in which it occurs, a context that conditions the nature of the solution to be provided.

In super-array mode, the size of the latter is either a default value[21], a value specified by the user or, still, *all of the available memory*. In the latter case, it may have an adverse effect on the cost. For example, if arrays (thereby of very large sizes) are to be initialized, this time (useless) may not be marginal. Otherwise, we see systems that are only concerned with the parts

20. Certainly not with a *segmentation fault* or any other funny name announcing the thing or the likely effect of the thing.

21. Decided at best by software designers based on what they know about what users most often do and what is the usual size of their applications.

of the arrays actually used and the fact that arrays are oversized is transparent. Still in super-array mode, if, for a prescribed size, the algorithm fails due to insufficient size having "lost" a reasonable amount of time (a few minutes), one trivial solution consists of restarting the process by increasing the size. A few minutes (of computation) were wasted but the solution only took a handful of seconds and, typically, the algorithm designer did nothing and the algorithm is necessarily simpler. As a matter of fact, developing an automatic system (transparent to the user) capable of resizing the resource and then of replacing the current data back into that new resource to be able to continue the process is not trivial. For some arrays, they merely have to be recopied (the copied values remaining relevant), for others, for pointers for example, the fact of copying in itself can result in these pointers being wrong in the new memory environment. Therefore, the re-creation of a consistent system is a difficult (programming) operation[22].

In the other mode where several arrays are allocated (`malloc` in C), if an array overflows, it is reallocated (`realloc` in C), for example, to two times its current size[23]. Two cases may then come across. The new array is simply the previous one that was enlarged, however it starts, in memory, at the same location. Therefore, its contents does not need to be copied. The other way around, this copying is necessary. It should be noted that all of this is automatic (the system knows how to do it) but the issues due to the presence of pointers (which would become erroneous) remain to be solved.

1.2. Internal data structures

An internal structure is specific to an algorithm and, as a result, depends on the nature of the latter. It is based on a set of elementary structures known as basic structures (array, list, stack, queue, pointer, grid, tree, etc.) and on the techniques associated (access, updating, etc.) with these components. As the name suggests, an internal structure is not known to the exterior. Moreover, this will fuel the classic debate about whether a quantity must be stored in the structure (at the cost of increased memory space) or computed on the spot; this debate identifies the difference in cost between a memory access and a computation.

1.2.1. *What should be stored in an internal structure and how?*

Independent of the situation, the internal data structure contains a specific number of compulsory arrays, essentially an array of coordinates (those of the vertices) and an array of elements (the indices of their vertices), but there is a way to do it differently. There are two other kinds of necessary arrays. Some are ephemeral and will be of no interest here. The others will be shortly used end-to-end during the course of the algorithm; the latter are the ones that we shall talk about, noticing that some resources are commonly (utilized by) several methods.

22. It should be noted, nonetheless, that this technique for recovering on the spot a lack of resources can be seen in some software programs.

23. A "small" resource will not be allocated on the spot every time it is requested. The doubling of the current resource makes this approach very effective.

The definition of an internal structure is paramount and, in general, not unique. Several possible solutions will therefore be considered to store a particular entity when there is a choice to be made. In Volume 1, regarding triangulations, it is deemed *self-evident* that such triangulation (of simplices) is described as a list of elements and, for each one of them, a list of vertices, their coordinates being stored in another array. In Volume 2, with regard to meshes, other elements are seen (including high-order elements) but the same form of organization, a list of elements, is kept and, for example, per element, a list of vertices and nodes (non-vertices if any) whose coordinates are in another array. This choice of organization, which is all natural and particularly simple, can be discussed, that is what follows.

• How can a triangle be defined

From a geometric point of view alone, a triangle (of the first-degree in two or three dimensions) is completely defined by the data of its three vertices, thus there are two ways to describe it:

– via three integers that are the indices of its vertices and a global array of coordinates. The index i therefore refers to the coordinates $coor(1:2,i)$;

– via three pairs of coordinates (and here, there is no need for a global array of coordinates), therefore we just have the set $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3).

The first method has the advantage of being compact in memory because the coordinates of a vertex shared by several elements are only stored once. Conversely, in the second method, the coordinates are duplicated as many times as a given vertex appears in an element. In addition, in the first method, any change to a coordinate is implicitly propagated to all the elements concerned. In the other method, the coordinates should be changed in all the elements involved, in order for the mesh to remain consistent. However, for the benefit of this method, there is no indirection, which can speed up some algorithms (for example, for visualization).

A third method, close to the second and well adapted to *vector* processing, consists of defining six arrays, two per vertex, one storing the first coordinate, the other the second, that is:

– an array for the coordinate x of the first vertex $(s1)$ and therefore for all the (ne) elements: $[xs1_1, xs1_2, ..., xs1_{ne}]$;

– an array for the y-coordinate of the first vertex and therefore for all the (ne) elements: $[ys1_1, ys1_2, ..., ys1_{ne}]$;

– *ditto* for the second vertex $(s2)$, namely $[xs2_1, xs2_2, ..., xs2_{ne}]$ and $[ys2_1, ys2_2, ..., ys2_{ne}]$;

– *ditto* for the third vertex.

All geometric operations involving the coordinates of the elements can therefore be carried out globally with these six large vectors, and a vector machine will process the values by groups of 64 as quickly as a scalar machine would process a single one. See also below, in a more abstract way, the link between memory (reading and writing) and the organization of the information stored in a structure (internal, external or arbitrary).

• How can a mesh be defined

At a minimum, the internal mesh data structure contains the coordinates of the vertices and the list of indices of the vertices of every element. However, to be effective, it optionally contains other information, possibly in large amounts. Some is of general interest, other is more specific and inherent to a particular method, as it will be seen further on. Of general interest, we find the array of neighbors (by two-dimensional edge, by three-dimensional face) and a germ (or seed) that indicates for every vertex an element of its ball. For a high-degree mesh, it is the list of nodes that will be considered and their coordinates.

• A brief conclusion

Let us retain that there is no single manner to define an internal mesh structure which, we recall, is known only to the construction, modification, or visualization, etc., meshing algorithm under consideration. It depends thereof. Let us point out that apart from the natural information (coordinates, list of elements), the structure contains all the information (value, array, etc.) enabling the algorithm to be efficient (robust and fast). Finally, the external mesh structure will preserve only what is strictly necessary and will be known to external access, this is the one that operates as interface, for example between the mesh world and the computation world.

1.2.2. *Internal structures, method by method*

We shall now method (mesh or optimization) by method look at the information that is specific to them by choosing the way[24] they are structured.

• Frontal-type method

The frontal method, in its conventional version, Volume 2, consists of starting from a front (the discretized boundary of the domain) and building the elements by relying on the items (edges or faces depending on the size) from the front and introducing new vertices. The front thus evolves (toward the interior of the domain) and the method terminates when the front is empty. The crucial operation is to know whether the point that is added or the element that is built is valid. This is determined by the detection of the absence of intersection between the current mesh and the items sought to be built. It is therefore essential to locate the point (the element under consideration) with respect to existing items (in principle, we are in a vacuum). A *grid* is therefore an element of the answer, the points are coded inside the grid and we quickly know what the potential conflicts are (filter effect). With the points (already inserted), a *germ* (an element of the ball, open or closed, of the point) is associated and from this germ the balls are found, which requires knowledge (of the array) of the *neighbors* as well as coloring. Next, the *front* is going to be manipulated (and developed), thereby the most flexible structure possible has to be implemented to enable insertions (a new item to be stored) or deletions (a front item that became inactive), thinking of a queue is then quite legitimate.

24. Following the previous discussion.

• Delaunay-like methods

The analysis of a Delaunay-like method and its various steps makes it possible to identify useful and utilized data as well as the basic techniques to be implemented. Therefore, the Delaunay criterion (inherent to these methods) concerns the balls circumscribed to the elements. Thus, it is not surprising that every element is associated with the *center* of its ball and the *radius* of the latter[25]. The construction of cavities, Volume 1, being done by looking at the neighbors of a (starting) element, the array of its neighbors will naturally be found, by element. Localization problems (finding out in which element a given point is located) lead to the use of a *grid* (or a *tree*) as well as, for every vertex, the data of a *germ* (an element of the point ball). A grid is also useful for filtering a point with respect to the existing vertices. A point marker (used to avoid losing one during insertion) and an element marker (useful for building cavities), therefore two arrays of *colors*, are also implemented. On the other hand, throughout the process, it is necessary to quickly know if an edge (face) is boundary or not (thereby constrained or not). This information will obviously be found by implementing an array of these entities based on hashing.

• Tree-based method

In the tree-based method (*quadtree* or *octree* depending on the size), seen not as a mere localization structure, but as a mesher (Volume 2, Chapters 4 and 5), two types of entities are used – vertices and cells (quadrants or octants). During the construction of the tree, the cells are cut in accordance with the mesh of the domain boundary. Each cell then contains a pointer to its parent and 4 (8) pointers to its children in order to be able to iterate through the tree by moving upwards and downwards. Terminal cells, or leaves, point to a linked list that contains the geometric entities (vertices, edges, triangles, or quadrilaterals) of the surface that intersect these cells. In practice, a mesh is composed of an array of coordinates and vertices and the index of the root cell, the latter pointing recursively to all the others. The structure is thus not required to store the exhaustive list of all its elements (cells) but only the first. During the insertion operations of a triangle into the tree, the list of terminal cells intersecting this triangle is found and it is added to the linked list of each cell. During the refinement and balancing phases, the cells are sliced according to geometric criteria (two vertices far too close inside the same cell) or to respect the balancing rules. Once the tree is finished, the mesh elements are obtained by slicing the terminal cells and they are stored in a conventional way.

• Optimization method (local)

An important part of a local mesh optimization algorithm, Volume 2, consists of processing the balls of the vertices (for moves) and the shells of the edges (for flips). The construction of these topological entities is achieved based on the neighborhood relationships (per face) between the elements; the array of *neighbors* is thus necessarily accompanied by a coloring of the elements to avoid inserting the same element several times in the searched array (ball or shell). Optimization strategies are based on an *a priori* quality criterion, involving the elements. A quality criterion is also defined involving vertices (taking into account the quality of the elements of

25. Which could be calculated on the spot at the expense of an extra time cost, but with immediate gain in memory or that could be replaced by a calculation of determinants, the latter being a solution that we consider less robust.

their ball). Keeping and maintaining these two qualities will result in avoiding recomputing them on the spot (a relatively expensive operation) and spending too much time with a vertex or an element of already good quality. Coloring can be used for the same purpose. In addition, sorting the quality array allows the algorithm to be directed (with or *versus* the use of dynamic thresholds, as seen above).

• Internal structures

The different structures are listed here (essentially seen as arrays) whose relevance was shown in the analysis of methods. The reader will easily see, for a given array, what methods are involved.

We denote by ne the number of elements, np the total number of vertices (and/or nodes), noe the number of nodes per element, nf the number of faces of an element and by dim the dimension of the space.

$\diamond$ $Coor(1:dim, 1:np)$, the array of the coordinates of the vertices (and non-vertex nodes).

$\diamond$ $Elem(1:noe, 1:ne)$, the array of the elements giving, for each one, a list of its nodes.

$\diamond$ $Neighbor(1:nf, 1:ne)$, the nf neighbors of a given element. A zero value indicates a boundary face, and there are no neighbors on this face. It should be noted that the mixed mesh problem emerges, and the number of faces per element is not constant.

$\diamond$ $Grid(0:.,0:.,0:.)$, three-index array (three-dimensional) complemented by its link array, the localization grid to help localize items, but also for filtering purpose. Here, the indices start obviously at value 0.

$\diamond$ $Tree(.)$, the tree itself according to the method retained to build it.

$\diamond$ $Germ_{element}(0:np)$, the index of an element (germ or seed) for each vertex. The elements are numbered from 1 to ne and the fact that the array starts at 0 allows finding a color for a virtual or non-existent element (which, precisely, has 0 for index, see above the neighbors array) without having to explicitly verify that the number is zero.

$\diamond$ $Color_{vertex}(1:np)$, the dynamic coloring of the vertices.

$\diamond$ $Color_{element}(0:ne)$, the dynamic coloring of the elements. This coloring is useful for the construction of cavities, but, in plain words, in the search for the (topological) balls of the vertices and edge shells. The index starts at 0 for the reason given above.

$\diamond$ $Center(1:dim, 1:ne)$, the coordinates of the center of the element ball.

$\diamond$ $Radius(1:ne)$, the radius of the (geometric) element ball.

$\diamond$ $Front(1:dim,.)$, the array of the faces of a front. This array can be seen as a queue, sorted or not.

$\diamond$ $Cont_{edge}(1:2,.)$ and $Cont_{face}(1:3,.)$, the arrays containing the constrained edges and faces (here, triangular), as arrays accompanied by a hash.

$\diamond$ $Sub-Dom(1:ne)$, a value per element to indicate membership to a sub-domain (a connected component). This information can be reduced by giving only one germ per sub-domain, the elements being obtained by neighboring (as long as a boundary is not crossed).

$\diamond$ $Edge_{edge}(1:2,.)$ and $Face_{face}(1:3,.)$, the array of edges (of the triangular faces), if necessary only.

$\diamond$ Etc., depending on the specific needs of the algorithm.

A few remarks on these arrays. The grid can be replaced by a tree to balance the filling of the cubes. The arrays storing the centers and the radii of the balls may, on the one hand, be grouped into a single one ($CenRad(1:4,1:ne)$) or, on the other hand, not exist at all, the centers and radii being evaluated on the spot. This is the debate of storage versus computation (memory and access versus computational time).

It is clear that these arrays (apart from the first two) either have no interest to the exterior or are easy[26] to recreate, and therefore are not intended to be found in the external data structure.

• Visualization algorithms

For this type of algorithm, the internal structures concern the organization chosen to describe a mesh (list of elements, list of vertices per element and array of coordinates) and a solution driven by this mesh, but, also, structures more directly related to the underlying graphic aspect. Given this specificity, these issues will be addressed in Chapters 4 and 5.

1.3. External data structures

An external structure is the means of communication with the exterior. Initially, we give in full, an example in `ASCII` of a mesh file (a small mesh of two tetrahedrons and six pyramids).

```
 MeshVersionFormatted  2
 Dimension
3

 Vertices
          9
  0.  0. 0. 0
  1.8 0. 0. 0
  1.8 1.8 0. 0
  0. 1.8 0. 0
  0. 0. 1. 0
  1.8 0. 1. 0
  1.8 1.8 1. 0
  0. 1.8 1. 0
  .9 .9 .5  0
 Tetrahedra
          2
  1 2 6 9 2
```

26. Think of the arrays of neighbors, its storage in the external structure unnecessarily puts an additional burden on the latter, its re-creation, via a hash, remains a reasonable operation.

```
 1 6 5 9 2
Pyramids
         6
 1 2 3 4 9 1
 7 6 5 8 9 1
 3 4 8 7 9 1
 2 3 7 6 9 1
 1 5 8 4 9 1
End
```

We then indicate the underlying philosophy in other words, keywords, fields, etc. This philosophy of the external storage is to avoid any redundancy and to be as compact as possible, however containing enough information to help reconstruct all the useful information (which, for some, were in the internal structure, but were not saved, partly due to memory issues) for the algorithm under consideration.

The MESHB file format[27], see the example above, is the one we have defined to store the mesh. The main idea is to be based on a set of keywords, each used to describe a mesh entity. A keyword is a simple code (a string of characters) that identifies the type of data that is going to be provided (*Vertices*, *Tetrahedra*, *Pyramids*, *Edges*, etc.). After the keyword, we find the number of entities that will be described and then one "line" per entity, see above and below for the description of the edges.

```
Edges
5
1 2 0
1 5 100
6 7 0
6 2 0
3 4 100
```

example in which five edges are described by the indices of their extremities and a reference.

The file is made up of a succession of keywords whose order does not matter. Each keyword is associated with a field of values. Readers are invited to visit the website of the library `libMeshb` for a comprehensive description of the format and of the programs freely usable to handle this type of file. In addition to a mesh, the format proposes an organization to store solutions, this is a SOLB file, as the example hereafter. The format, which evolves according to technologies and novelties, offers backwards compatibility.

```
MeshVersionFormatted 2
Dimension 3
SolAtVertices
5
```

27. B file for binary.

```
 1 2                % 1 = A single field, 2 = Vector-type field
 1. 6. 8.
 2. 4. 8.
 3. 2. 6.
 4. 5. 6.
 5. 2. 6.
End
```

Files can be stored in ASCII format or in binary. The binary format is preferred in terms of performance since current flash memory (SSD) has speeds of several GB per second, while the ASCII format cannot exceed a few tens of MB per second because of the time taken for interpreting the data. The result is a factor of 100 in speed for the benefit of binary, and given that the acceleration of current SSDs and networks do not benefit the ASCII format, it is expected to further increase. In addition, reading and writing to the same file in ASCII are very complex operations to be parallelized due to the unpredictable length of the data, whereas access in parallel is trivial in binary. For example, the number 2.5 in binary will always take 8 bytes if stored in double precision, whereas in ASCII it could be perfectly written as 2.5000000 or 2.5 followed by a series of white characters for the purpose of aligning the data with the columns. The result is that if an ASCII file is split into two equal parts with the same number of bytes in order to be processed in parallel by two processors, the task receiving the second block will have no idea about the starting address of its data, which could start right in the middle of a line or a real number. This would require post-processing to partially re-assemble the data decrypted in parallel.

1.4. Data structures and memory access

In general, the organization of data in memory should not solely be a function of the requirements of a particular algorithm, but must take into account the physical constraints of the hardware. It is therefore necessary to dive a bit into the technical aspects because hardware and its architecture (multi-core, GPU, but also simply sequential) have a preponderant influence on performance.

The performance of the computer main memory (DRAM) depends on two main factors: clock speed in Hz, that is, the number of words that can be read or written in a second and the width of these memory words in bytes. It should be noted that these sizes are not related to the frequency and the bandwidth of the memory bus, which is used to connect the DRAM to the CPU. These values are often put forward by manufacturers, but hide a much more complex reality. On the other hand, the memory frequency has not changed for several decades and stagnates around 100 MHz. To improve performance, memory manufacturers have no other solution than to enlarge the memory word. This has a non-negligible impact on how memory is managed because a system based on DDR4-2133 modules (as on a simple laptop) is capable of transferring 133 million words of 128 bytes per second. As a result, when the system accesses the memory to read a byte or 128 bytes, the access time is the same. This indicates the importance of grouping memory accesses together and consecutively, and therefore, to store information about an entity in an appropriate way.

As it has just been said, the cost of access is constant up to 128 bytes. As such, it may be useful to store the data related to an entity in a consecutive fashion. As seen above, simply for a coordinate array, one can have a single array containing all the coordinates of a vertex or an array for each of the coordinates. In the first case, access to the x-component of a vertex triggers the reading the neighboring cubes that contain the y-components (as well as z-components).

The reasoning can be taken a step further by storing the vertices in the form of an *array of structures* (AOS), each structure containing all the data related to a vertex – coordinates, metrics, material reference, germ, or degree (number of elements of its topological ball), etc. The underlying idea is that, again, access to all this data is done simultaneously, therefore at the same cost as access to a single data item. Such a structure is optimal when the algorithm needs all of the data. On the other hand, if the algorithm is required to travel the structure to retain only one integer value, it triggers the reading of 128 bytes to keep thereof only 4 (the integer value). It is then advantageous to store data in the form of a single structure containing multiple arrays – an array for each coordinate, a reference array, etc. This will then be referred to as *structure of arrays* [SOA].

What structure (AOS or SOA) should be chosen? None are optimal in all situations and the designer is left with the choice of organizing the data according to the modes of access of its algorithms. In some simple cases, one of the methods is clearly more efficient and also for most algorithms. In other cases, access patterns change and it is impossible for a single storage method to be optimal. It is then necessary to change the type of layout throughout execution using AOS $\leftrightarrow$ SOA conversion routines, when the program will enter a phase where the nature of data accessed is recognized as being of one type rather than another. It should be noted that the change of data organization, called *transposition*, is not transparent from the programmer's point of view. For example, in `C` language, `mesh->vertex[i]->ref` in SOA mode becomes `mesh->vertex->ref[i]` in SOA mode.

This discussion, if we now take the case of a square matrix, repeats itself to know how to store it, by line, by column or even by metaline, etc. This non-trivial question falls outside the scope of this book.

*

* *

In this chapter, we tried to show how to use basic data structures, therefore conventional (arrays, list, linked list, tree, grid, etc.), and also the basic techniques (sorting, hashing, coloring, etc.) for developing performing algorithms not only for the purpose of improving speed, but also from the point of view of optimizing the necessary memory space. Performance is achieved when, for the most part, obtaining linear complexities or close whereas, in some cases, a naive implementation (perhaps simpler) would rather be quadratic. However, it should be stressed that performance, in terms of complexity, is not really obtained if memory management is not finely achieved in order to avoid the cost of memory access. Naturally, this potential negative impact is not visible if we are just addressing small-sized cases or, at the very least, with a size not

hampering operations related to accesses. To properly take these kinds of issues into account, we saw that a minimum of knowledge of how readings and writings occurred was necessary and therefore, sometimes, go so far as to look at how many bytes were affected by any of the operations and whether it was possible to influence this factor by organizing the data structures used within an algorithm differently (internal structures). Now, regarding the external data structures, as interfaces between a given algorithm and the outside world, we saw the limitations of writings (readings) in *ASCII* mode and all the benefits brought by writings and readings in binary mode. In addition to speed, the necessary memory resource is significantly reduced. This mode is therefore the only one really suitable for processing large-sized problems.

We took the liberty to promote mesh formats and solution fields that we developed by indicating the existence of a library of software programs for their processing (writing, reading, converting, etc.).