UCLouvain

École polytechnique de Louvain

Effect of horizontal meshing on dissipation in a 3D ocean model

Author: **Arno Van de Vyver**
Supervisors: **Emmanuel Hanert, Jonathan Lambrechts, Vincent Legat**
Reader: **Eric Deleersnijder**
Academic year 2021–2022
Master [120] in Computer Science and Engineering
Master [120] in Mechanical Engineering

Abstract

The field of numerical simulation has been in constant progression for the last several decades. One important domain for which such simulations can be interesting to perform concerns the submarine processes, occurring in the seas and the oceans of the world. Those numerical analysis may be performed using finite element methods, which requires a discretization of the space. Several techniques to perform such a task are available, and different coordinate systems can be distinguished. This thesis will be interested in one system in particular, called the z coordinates, which may happen to be more dissipative. In this report, this decay of energy have been analyzed and quantified on a simple simulation setting. As this decay has been considered to be significant, a new remeshing strategy have been developed . Although a solution could be to refine the mesh on strongly dissipative zone, the technique implemented in the context of this thesis tries to lower this energy dissipation while keeping the size of the elements similar. This lead to the creation what has been referred to as the contour-fitted meshes. Although we did not have the time to test this new kind of meshes on real-life test cases, the results obtained on the simple setting are a good reason to expect a notable and positive impact of their usage. It is in consequence to be hoped that further work will be performed in order to test this concept in a broader perspective.

Remerciements

Après près d'une année passée à la réalisation de ce mémoire, je tiens tout d'abord à remercier mes promotteurs, pour le temps et la patience qu'ils m'ont accordé tout au long de ce périple.
Je remercie tout d'abord Jonathan Lambrechts, qui m'a initié à la comprehension de ce sujet et m'a fourni les outils nécessaire afin d'atteindre mes objectifs et d'obtenir des résultats. Je remercie aussi Emanuelle Hanert, pour le soutien constant qu'il m'a apporté ainsi que les conseils qu'il m'a prodigué tout au long de l'année. Enfin, je remercie bien évidemment Vincent Legat, pour ses critiques toujours constructives et bienveillantes qui m'ont permis d'avancer sereinement et de me poser les bonnes questions afin de progresser.
Je tiens aussi à remercier Ange Ishimwe ainsi que l'équipe de SLIM, pour toute l'aide qu'il m'auront apporté durant cette année et sans laquelle je n'aurais pas pu en arriver là.
Finalement, je tiens à accorder un remerciement tout particulier à Mareva. De l'implémentation à la rédaction, elle aura su être toujours présente pour me soutenir et me pousser à continuer, même dans les moments les plus durs.

Contents

Introduction

Contextualisation

During the last few decades, computers are become more and more efficient [1], opening to a whole world of new possibilities. A really interesting one is the ability to perform complicate numerical simulations, allowing to solve countless analytical problems, but also to provide real world predictions based on actual measurement. We could, for example, think about atmospheric data allowing to predict the weather [2], or even submarine processes based on tides and temperature data [3,4]. This second kind of simulations is the one concerned in this master thesis.

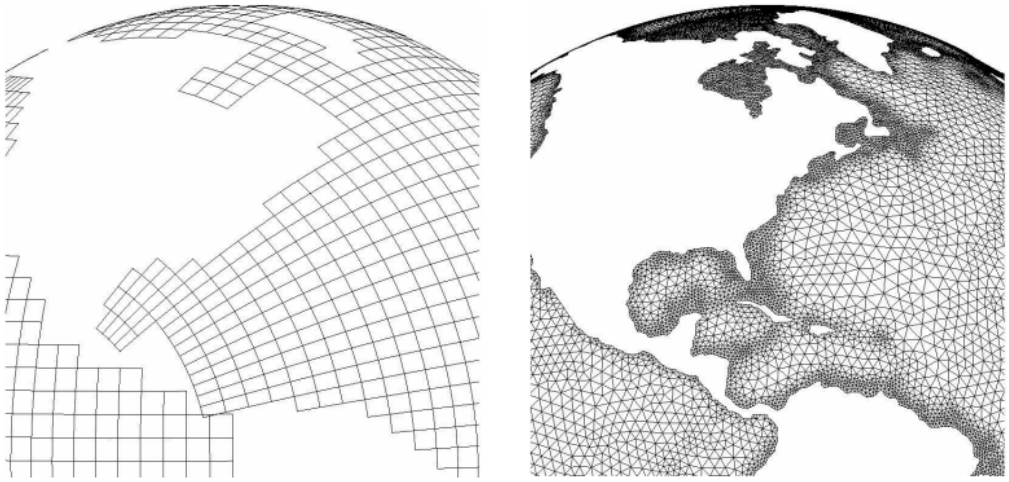
Indeed, many interesting processes may happen in the oceans and seas of the world. It is thus of a great importance to be able to study those, and even predict or prevent them. We could cite, as examples out of many, the analysis of the changes due to the climate changes [5], or even more localized and punctual issues such as the extent of an oil spill occurring somewhere [6], allowing to optimize our way of reacting to these.

Furthermore, oceans amount for a large part of the earth [7]. As one of the quest of humanity has always been to be able to understand the mechanism of the world and its subtleties, improving our comprehension of the marine phenomenons could perhaps help us to achieve this goal.

For this purpose, the current state of the art uses finite element methods [8], consisting in a discretization of the space to analyse and of the resolution of the conservation equation on each of the elements. Those discrete elements are all collected in a structure called a polygon mesh, simply referred as mesh in the following. Although other techniques are still in development and might be useful in later studies [9], usual finite element methods have already been proved very effective, providing accurate results in a reasonable amount of time [10].

The different types of meshes

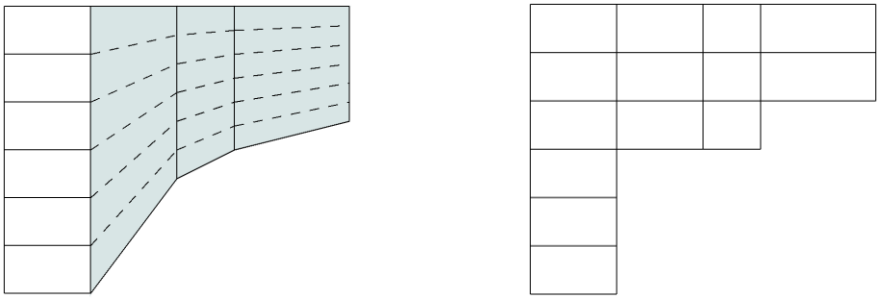
In the field of numerical analysis, several kinds of mesh are used in order to perform simulation. As this thesis concerns the modelization of aquatic processes, we can distinguish two categories of meshing strategies [11], depending on their direction: the horizontal meshing and the vertical meshing. The first one can be seen as the discretization of the surface of the water, while the second takes its depth as well as the seabed into account.
In terms of horizontal coordinates, two main types of mesh exists: the structured and the unstructured grid. In structured, or regular, grids, quadrilateral elements are uniformly spaced across the surface to mesh. Of course, since the surface of the earth is spherical, the use of curvilinear lines is mandatory, and the space between elements obviously vary. An example of such mesh is given on figure 1a. The other kind of the mesh uses an unstructured grid. The elements are usually triangles, and their size may vary strongly along the mesh. An example is visible on figure 1b. Concerning the vertical coordinates, several strategies exists, although two can be considered as the main ones: the σ coordinates and the z coordinates [11]. Other systems may be the isopycnal or the hybrid coordinates, but they can mainly be seen as extensions of the first two. The σ coordinates uses the same number of elements



(a) (b)

Figure 1: Examples of a structured mesh [12] on the left, and a structure mesh on the right [13, 14]

per water column. Furthermore, their bottom elements adapts to the seabed, allowing to fit the actual bathymetry closely. They are mainly used at low depth, and are convenient to represent stratified zones in the oceans and seas. The other kind of vertical meshing strategy uses what are called z coordinates. In contrary to the previous one, the number of elements per column of water changes with respect to their depth. There are thus more elements per water column as the sea gets deeper. Furthermore, the surface of their upper and lower surfaces are always kept flat, and are at the same depths for all the elements for all the i-th elements of each column. A schematic representation of such coordinate system is shown on figure 2. As one could expect, the choice of the coordinate systems strongly depends on the application. In the following of this thesis, the coordinates system that will be used are unstructured triangular meshes using z-coordinates.



(a) (b)

Figure 2: Representation of the σ and the z coordinate systems [15]

The numerical errors when using z-coordinate and the development of a new remeshing strategy

The aim of this master thesis will be dedicated to the analysis of the numerical errors that occurs when using z coordinates. Although easier to set up and more computationally efficient, this types of elements suffer from a lack of accuracy when the resolution gets too small. Indeed, figure 2b shows that the bathymetry resulting from the usage of such system is formed by steps. If, furthermore, these resulting lines of isobathymetry are sharp and strongly non-smooth, we can expect strong turbulent flows to arise around these, where the flow with the actual bathymetry could have been laminar. Such cases could potentially lead to a large amount of numerical errors. One way to avoid these would be to refine the horizontal mesh near those contour lines situated around the plateaus formed by the z coordinates. This strategy could probably work in practice, although it would unfortunately increase the computational cost of the simulations, reducing the efficiency such software using finite element methods.

In this thesis, we develop another solution to avoid this apparition of turbulence and reduce the amount of errors. Instead of increasing the resolution, the idea is to implement a remeshing strategy that would adapt existing horizontal meshes to the initial lines of isobathymetry. By doing so, it is expected to obtain meshes using z coordinates which would still closely fit to the actual bathymetry, but presenting much smoother isobaths. This would consequently avoid the creation of unexpected turbulent zones, thus reducing the amount of numerical errors. It will also be possible to obtain accurate results while using less elements, impacting positively the time required to perform the simulations. By analogy to the term *body-fitted* meshes [16, 17], the mesh resulting from such a remeshing strategy will be frequently referred as *contour-fitted* meshes.

Structure of the thesis

This thesis would be divided in three chapters.
The first chapter present a type of data structure that are used to represent and adapt meshes, namely the *half-edges* [18, 19]. The different parts composing this data structure will be detailed, along with the different local transformations that they allow to easily perform. Other implemented features will be mentioned, such as the addition of points into the mesh, as well as the definitions of size maps. One of the interesting way to define such a map based on a line of isobathymetry will also be explained, as well as a simple smoothing strategy going along with it. We will also mention how it is possible to read an existing mesh or create a new one based on the user's needs, and how they can be written using σ or z coordinates. After that, the algorithm used to update the mesh will be detailed, along with the quality indices that are used to define which local transformations should be applied and where. The chapter will end by showing the kind of meshes that can

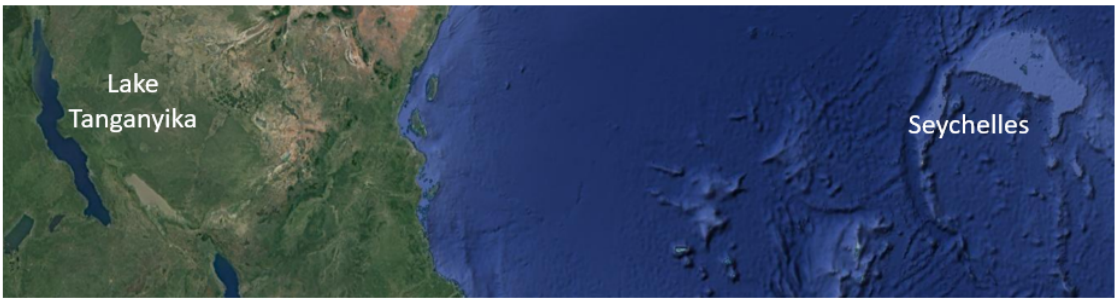


Figure 3: Satellite visualisation of the two geographical zones used in the following chapters

be obtained, mostly focusing on the reading of existing meshes and on the adaptation to lines of isobathymetry. Three different bathymetries will be used to define those contour lines: an analytical, smooth and axisymmetric bathymetry, a bathymetry of the Lake Tanganyika, and a bathymetry of the seabed of the Seychelles. Those last two zones are situated close of each other in the Eastern part of Africa, and are visible at figure 3.

The second chapter will be dedicated to test the concept of contour-fitted meshes, in order to justify the interest of such mesh adaptation. Indeed, although the intuition tends to make us feel that this idea should help to decrease the numerical degradation of the energy, this must imperatively be verified before computing the actual algorithm to automate the process. For this purpose, a basic test case will be defined where the initial solution can be found analytically. The initial total energy of the setting will be computed, and its decay will be measured over time and compared for two types of mesh. The first one is adapted to the chosen isobaths and the another one does not takes them into account. Since the concept of contour-fitted meshes still needed to be verified, the contours were chosen in such a way that adapting the mesh to them would remain simple. For this reason, the contour-fitted mesh used in this chapter was simply created with the

Gmsh software [20]. The full description of the test case will be given, along with the computation of the initial fields required to start the simulations. Those will be performed using SLIM [21]. The name of SLIM stand for Second-generation Louvain-la-Neuve Ice-ocean Model. It uses Discontinuous Galerkin finite element method [22, 23] and unstructured mesh in order to perform hydrodynamic simulation with a high precision. Furthermore, as this thesis was performed in association with its development team, it was of course an obvious choice. The results of the simulations will then be displayed and commented, and we will end this chapter by discussing about the potential interest of automating this remeshing strategy which adapts a mesh to the lines of isobathymetry.

Chapter 3 will be dedicated to the development of the algorithms that were used in order to automate the process of adapting existing meshes to the contours of bathymetry. This task will be decomposed into four principal steps: the computation of the lines, their filtering when they get too close to each other, their handling during the process of the algorithm to improve the mesh qualities, and the attribution of each element of the mesh to their corresponding plateau. Throughout this chapter, two different strategies will be detailed and their implementation will be compared for each of the four steps. One of the strategy will consider the contour lines as a set of points, and will be referred to as the *pointwise* strategy. The other, in contrary, will represent them as a composition of segments and will try to maintain the continuity of the curves. It will thus be referred as the *segmentwise* strategy. The chapter will proceed by displaying various meshes using the same bathymetries as the one defined in Chapter 1. The results will be shown using three different methods. The first will be a naive method, simply consisting in setting each elements to their closest plateau, while the other two will use the aforementioned strategies. Finally, the different meshes obtained will be discussed, pointing out the advantages and the disadvantages of both strategies, as well as mentioning the further

improvement that could still be added.
In the end, we will conclude by summarizing the methods and algorithms that were defined in this report, as well as the results obtained at the end of each chapter. We will finish by making some suggestions of improvement that could be interesting to investigate in the future.

Chapter 1

The mesh generation and improvement

This chapter will be relatively general and will concern the development of a mesh generation algorithm. It will use local transformations to update existing or new meshes, and a data structure called the *half-edges* [19], also known as the doubly connected edge list [24]. Although there already exists some packages and programs implementing this kind of mesh generation, the interest of its development detailed in the following of this chapter is twofold: first, it was a good introduction in order to familiarize with the structures and the operation in question; and second, it allowed to merge the whole process of reading an existing mesh or creating a new one, translating it into a half-edges representation, computing the contours from the mesh, including them into the mesh, and update this mesh while taking the contours into account.

1.1 Introduction

This chapter is composed of the details of the algorithms used in order to obtain and update a mesh by using half-edges. The whole program can start either by using an existing mesh, or by creating a custom one, only requiring some information about the needs of the user. Figure 1.1 shows a general flow-chart of the program, with the possible input and outputs as well as the optional steps. Some of these steps contains parameters that can be tuned by the user, up to some extent, but the specific details about their tuning are out of the scope of this document. In contrary to most of the flow-charts in the following of this report, it does not only shows the initial inputs, but also details the intermediate outputs of each boxes. All along this chapter, almost all of the boxes will be detailed and sometimes split into several intermediates steps, while avoiding to dive too deep into the actual lines of the code. The only part which will not be detailed in this chapter concerns the computation of the contours, as this will be the subject of the Chapter 3.

The first important part of the program that will be explained are the data structures that are used throughout the whole algorithm [25]. After that, the different mesh operations that have been used will be detailed, with the help of schematic drawings. Those operations are of five kind: the edge swap, the edge collapse, the edge split, the face split and the vertex relocation [26, 27]. Of course, there exists other operations, such as the vertex split [28], which will not be used nor detailed here. An interesting suggestion in order to improve the algorithm could be to include those operation in the update algorithm.

After this section, several features will be briefly explained. Those features concerns the initial addition of points into the mesh, the definitions of the size maps, the smoothing of the bathymetry, the creation or the reading of meshes, and the two types of methods to write those in `.msh` files. Those two last methods differs in the type of coordinates

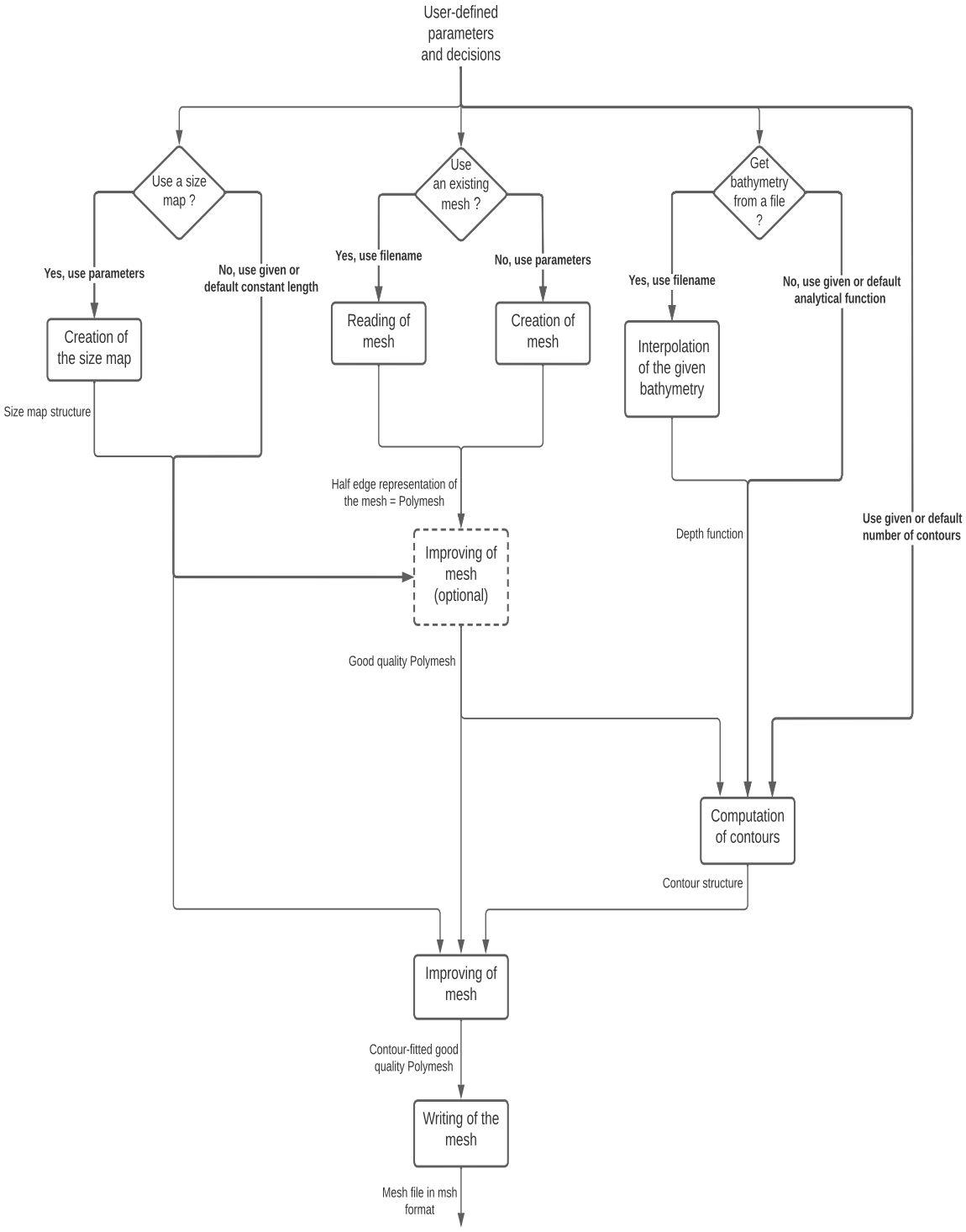


Figure 1.1: Flow-chart of the full program

used to display the mesh, the one using σ and the other z coordinates [11].

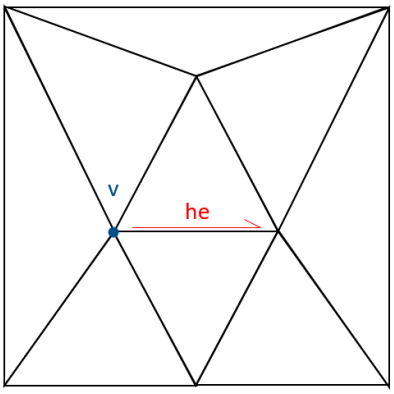
Then, the update algorithm, also referred in the following as the main loop, will be explained. This is the one in charge of deciding which operations should be applied and where. To achieve this, we need to be able to measure if an operation will improve the mesh or not. This lead to the choice and the definition of quality measures that will describe the positive or the negative effect of applying a local transformation to a given edge. Finally, most of the local operations will be put together, in order to obtain the main loop updating a mesh.
This chapter will then conclude by showing the meshes that we can obtain thanks to this update algorithm, using various parameters for the origin of the mesh and for the size map.

1.2 Materials and methods

In this section, the details on how mesh are read, stored, written and modified will be detailed. A first step had been to connect the provided codes for which the two main parts were in Python and in C99. This implies that the code have been made usable with both languages. Indeed, even though the program for the whole process, starting from an existing mesh and finishing with a mesh of a reasonable quality, has been fully automated, the interested user can also implement its own algorithms using half-edges structures to store a mesh and improving it with local transformations. In any case, it is recommended to tune at least the values used for the size map and for the number of contours to compute. A GitHub repository is currently in preparation order to make the aforementioned code available.

```
struct Vertex {
    double p[2];
    double depth;
    double size;
    int data;
    int is_model;
    HalfEdge *he;
};
```

(a)



(b)

Figure 1.2: Representation of a vertex; on the left in the code, on the right in the mesh

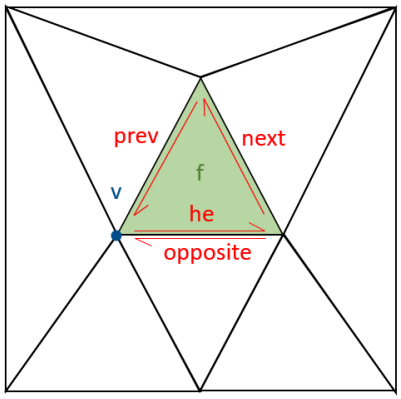
1.2.1 The structures

The first important point concerns the detailing of the structures that will be used by the various algorithm developed in this chapter, as well as in Chapter 3 [25, 29]. A description of a structure content will be accompanied by a scheme showing its actual representation in a mesh.

The most basic data structure that are used by the half-edges are the `Vertex`. It is visible on figure 1.2. They represent the nodes of the mesh and contains consequently its x and y coordinates in a vector `p`, along with other fields providing information about the vertex. These fields include the value provided by the size map at its position, as well as its depth, a data value which may be used to differentiate each vertices, and an integer called `is_model`. This last field will be detailed later in this chapter and its usage will be extended in Chapter 3. It serves to define if the node is part of the border, a contour lines or none of these. A last field, necessary for the half-edges, is the pointer to one of its incident half-edges `he`. It may be seen as the link of a vertex to its neighbours in the mesh. This is of an uttermost importance when

```
struct HalfEdge {
    Vertex *v;
    Face *f;
    HalfEdge  *prev;
    HalfEdge  *next;
    HalfEdge  *opposite;
    int data;
    int is_model;
};
```

(a)



(b)

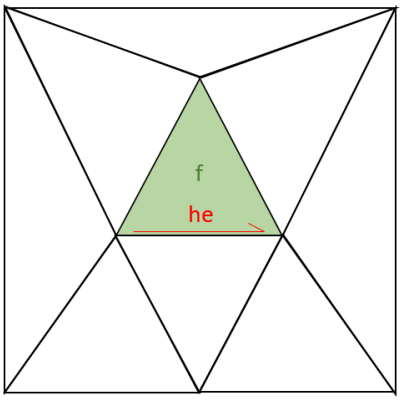
Figure 1.3: Representation of a half-edge; on the left in the code, on the right in the mesh

dealing with local operations, or when one wants to find all the incident faces of a node. The algorithm to do such tasks will be detailed at the end of this subsection, showing why this field is so meaningful.

The second structure, shown at figure 1.3, contains the information about the `HalfEdge`, which is obviously the one giving its name to the whole data structure. Similarly to the `Vertex`, it has a field to hold a data or to define if it is a model edge or not. This field will not be used before the Chapter 3. Besides containing a pointer to its origin vertex, and to the face on which it belongs, three pointers contains the address of three other `HalfEdge` structures. The pointer `prev` and `next`, as shown on figure 1.3b, contains simply the addresses of the previous and the next half-edge on the same face as the current half-edge. It is important to note that the arrows are arranged in the counter-clockwise order. This is the case for each faces, and during the complete simulations. If the half-edges of a face would happen to appear in the clockwise order, this would show an error which will

```
struct Face {
    HalfEdge  *he;
    int data;
};
```

(a)



(b)

Figure 1.4: Representation of a face; on the left in the code, on the right in the mesh

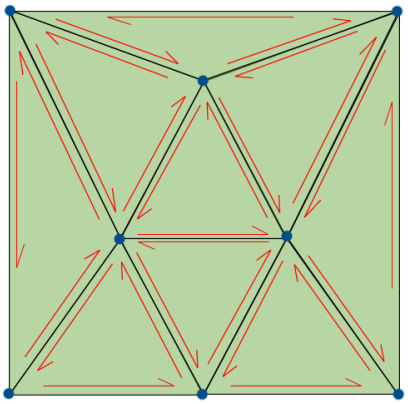
have to be treated. The next section detail how the local operations checks the resulting mesh before being applied, in order to avoid such issues. The final pointer contains the address of the opposite half-edge, which can be seen as the other half of the edge situated between the two half-edges. We can see that it is directed in the other direction, mainly to keep the counter-clockwise order on all the faces. It is useful to mention that the opposite of the opposite of an half-edge equals the half-edge itself. This is also true for the next half-edge of the previous one, the previous of the next one, or even for the one obtain after following three `next` or `prev` pointers.

Even though this might seem slightly surprising, the structure of a face, on figure 1.4, is the simplest. It only stores a data, which is rarely used, and a pointer to one half-edge situated on this face. Indeed, thanks to this pointer, all the required information we would need for a face are already available.

Finally, a list of all the faces, the half-edges and the vertices created during the running of the algorithm are stored in arrays, themselves stored into one structure called `PolyMesh`. This is why in the following,

```
struct PolyMesh {
    Vertex **vertices;
    HalfEdge **hedges;
    Face **faces;
    Vertex ***contours;
    int ****cells;
};
```

(a)



(b)

Figure 1.5: Representation of the whole half-edges data structures; on the left in the code, on the right in the mesh

the complete structures representing the mesh will be sometimes called polymesh. The other arrays that it contains are a matrix of cells and an array of arrays containing pointers to vertices. The first one is a data structure used for the size map, while the second is used to represent the contours [30]. Both will be detailed in Chapter 3. Other fields are actually present in the code, but since they are not required for the understanding of this report, they have not been included on this figure.
These structures, put together, allow to easily obtain the neighbouring faces, edges and vertices of a node. This is the main reason why it is so useful when one wants to work with local operations, as it is the case here. For example, the algorithm shown on figure 1.6 shows a simple algorithm to visit all the neighbouring vertices of a given `Vertex` v.

1.2.2 Description of the local transformations

This section will details the five local transformations that are used in the following algorithms. Most of the explanations will rely on draw-

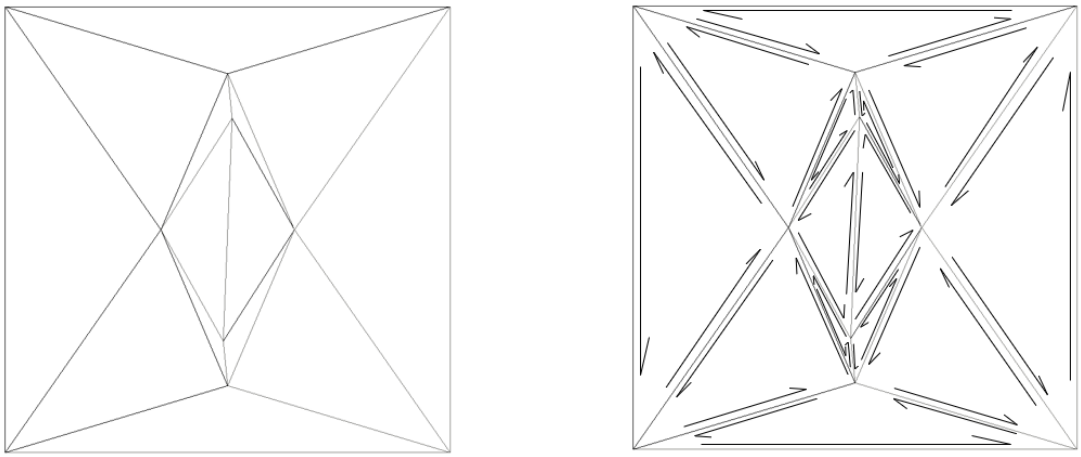
```
void *visitFaces(Vertex *v)
{
  HalfEdge *he = v->he;
  do {
    vistit(he->f);
    he = he->opposite;
    he = he->next;
  } while(he != v->he);
}
```

Figure 1.6: Small and efficient algorithm to visit all the incident faces of a given vertex using the half-edges

ings where the structures that are added or modified can be visualised at each step. The common thread of this subsection will start with the simple mesh visible on figure 1.7, and will transform it subsequently by applying each of the operations on the identified edges.

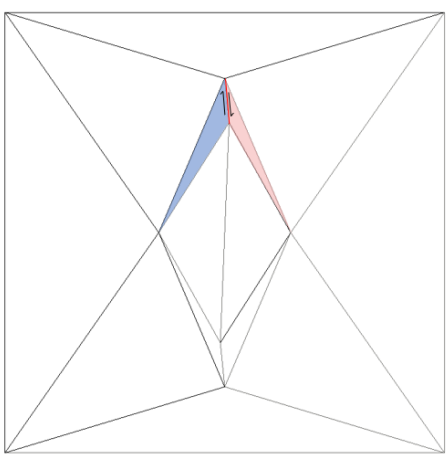
Edge swap

The swapping, also called the flipping, of an edge is an operation consisting of disconnecting an edge from its two end-vertices, and reconnecting them to the opposite vertices of its neighbouring face [31]. The operation may seem quite simple, but some additional considerations need to be mentioned. Indeed, this operation will not be valid if the quadrangle formed by removing the edge is not convex. For example, figure 1.8 shows such a case, where the red and the blue faces form a concave shape. The resulting mesh shows the blue face overlapping the red face, thus not respecting the manifoldness property of the mesh [32,33]. In contrary, figure 1.9 shows the complete process of the swapping procedure, starting from an edge which respect the convex shape condition.

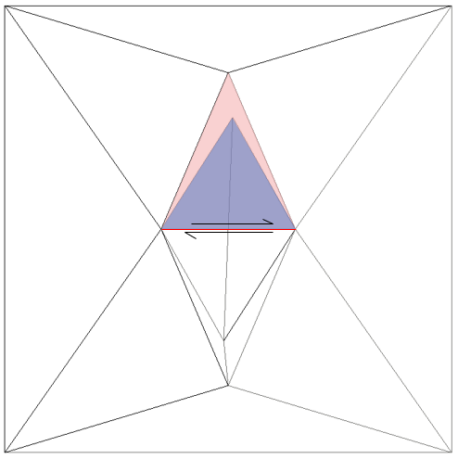


(a) (b)

Figure 1.7: Initial mesh used in this section; without its half-edges on the left; and with its half-edges on the right

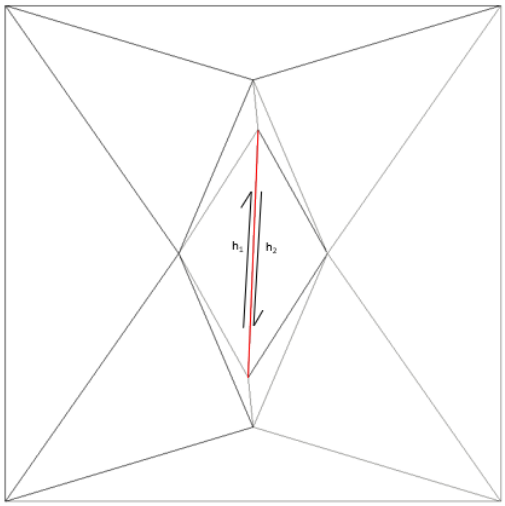


(a) The red edge is the edge to swap, with its two initial neighbouring faces colored in red and blue.

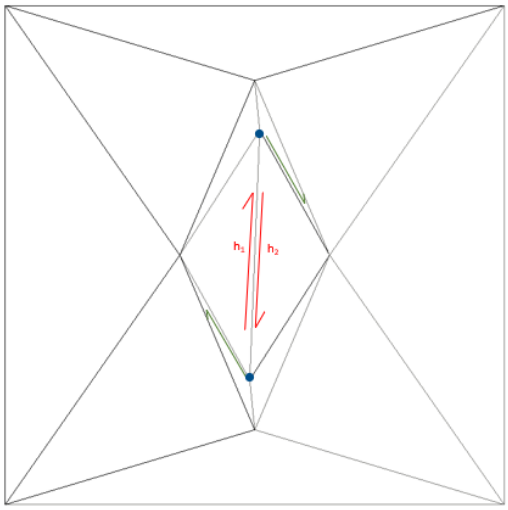


(b) Mesh obtained after the swapping operation. We can see that the two initially neighbouring faces are now collapsing.

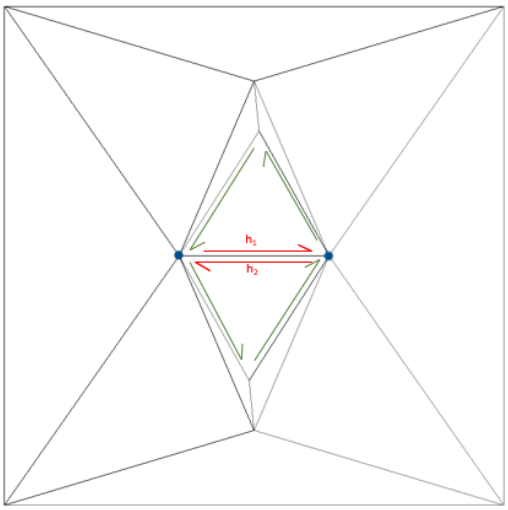
Figure 1.8: Swap operation failing the validation check



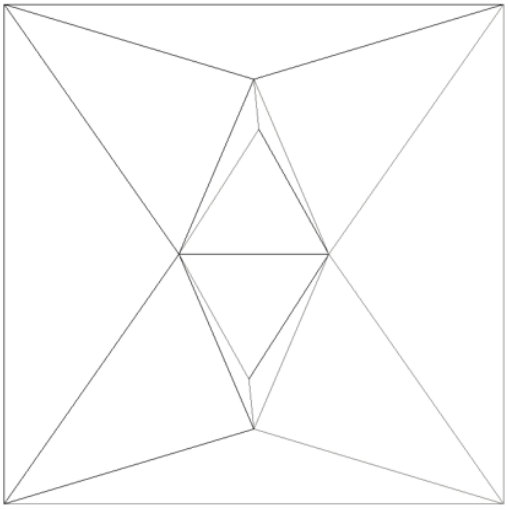
(a) The red edge identifies the edge to swap, while h_1 and h_2 are its two half-edges.



(b) First, we have to ensure that the half-edges pointed by the two end-vertices are different from h_1 and h_2.



(c) Then, the actual swap is performed by setting the green half-edges and the red half-edges to their correct face, and setting correctly the `prev` and `next` pointers. Then, the origin of the red half-edges is changed, and we ensure that each created face point to a half-edges it contains.



(d) Final mesh obtained after the performing of the swap.

Figure 1.9: Step-by-step visualisation of a successful swap transformation using half-edges

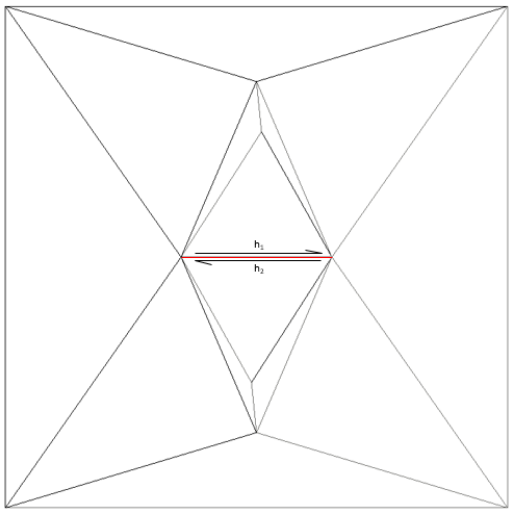
Edge collapse

The edge collapse is an operation consisting of merging two vertices into a unique vertex [34]. It is possible to find several version in the literature, taking different point to place the final vertex. Three versions are used in the current algorithm: one choosing the position of the origin of the half-edge, one taking the origin of the opposite half-edges, and a last one which takes the position at the center of the edge.
This operation is one of the most error prone as there are two conditions which needs to be satisfied if we want to ensure the resulting mesh to be manifold [35]:

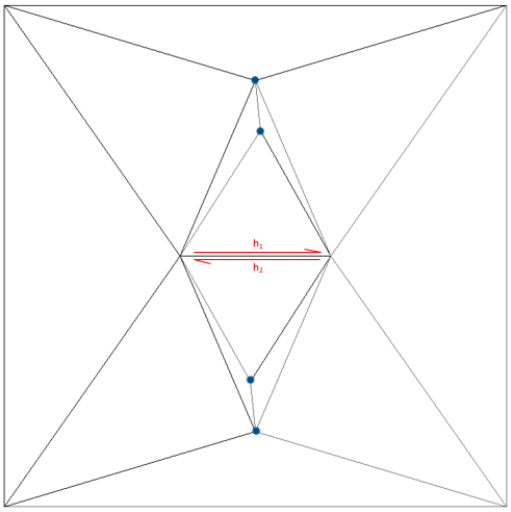
1. First, the two end-vertices of the edge to swap should only have to common neighbours. If this condition is not satisfied, the connectivity property of the resulting mesh is not guaranteed. This case is shown on figure 1.10.

2. Second, we have to ensure that no triangle gets inverted after the operation. In order to be verified, this condition requires to compute the orientation for each of the resulting faces. As detailed previously, when using the double connected linked list, the half-edges of each faces needs to appear in the counter-clockwise direction. This condition is thus not validated if one of the faces happens to have the direction of its half-edges inverted. In this case, the results does not respect the geometry of the initial mesh. Such a situation is shown at figure 1.11. It uses the mesh before the swap operation, in order for the resulting errors to be more easily visible.

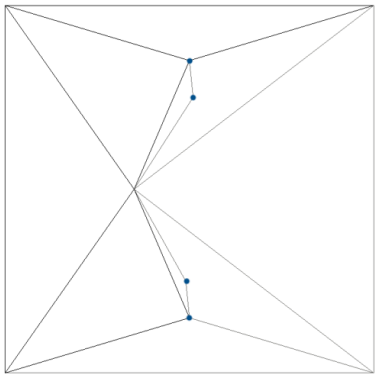
A situation of an edge collapse where those two conditions are satisfied is shown at figure 1.12, for each of the three versions.



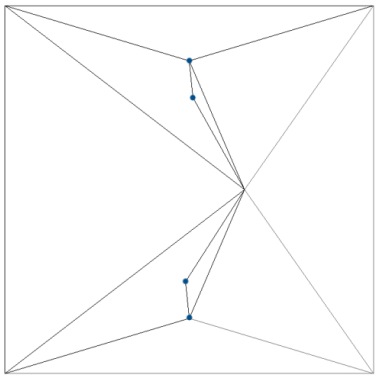
(a) The red edge is the edge to collapse, while h_1 and h_2 are its two half-edges.



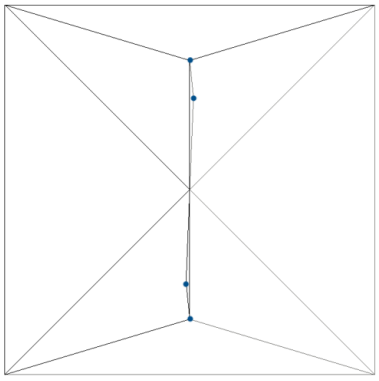
(b) When performing the first validation check, we detect that the two origins of the identified half-edges share more than two neighbours, highlighted with the blue dots.



(c) Resulting mesh when collapsing the origin of h_2 on the one of h_1

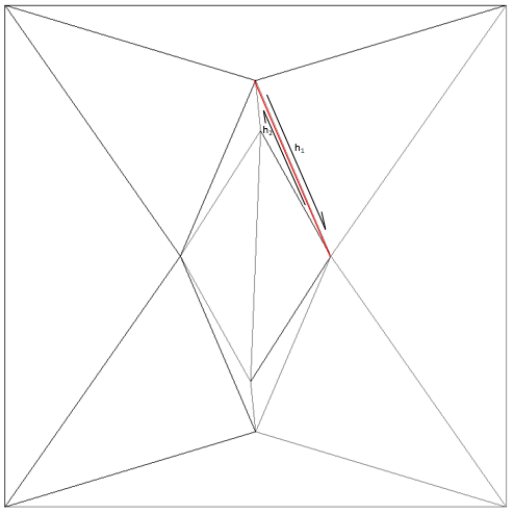


(d) Resulting mesh when collapsing the origin of h_1 on the one of h_2

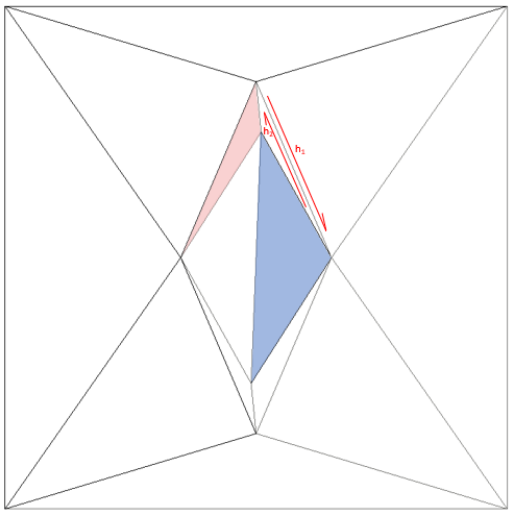


(e) Resulting mesh when collapsing both origins into their mid-point

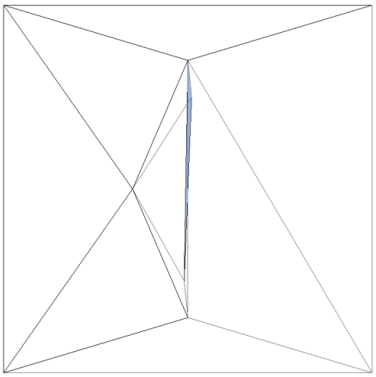
Figure 1.10: Collapse operation failing the first validation check. We can see on the second line that every version of the operation fails since they create some kind of concave quadrangles, not respecting the connectivity of the mesh



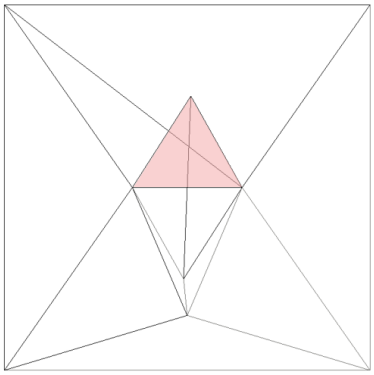
(a) The red edge is the edge to collapse, while h_1 and h_2 are its two half-edges.



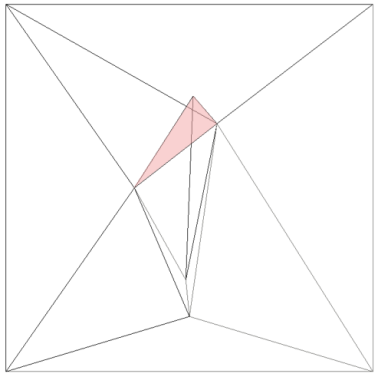
(b) The initial shape of two faces incident to the end-vertices are highlighted in blue and in red.



(c) Resulting mesh when collapsing the origin of h_2 on the one of h_1

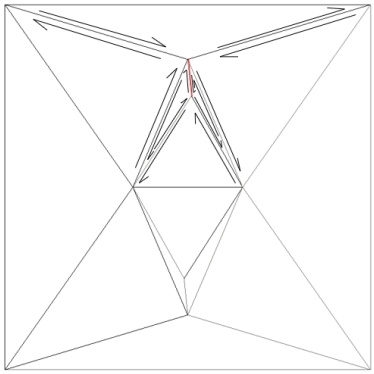


(d) Resulting mesh when collapsing the origin of h_1 on the one of h_2

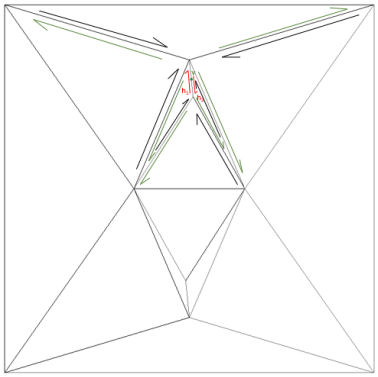


(e) Resulting mesh when collapsing both origins into their mid-point

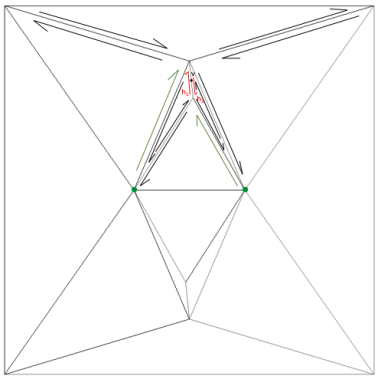
Figure 1.11: Collapse operation failing the second validation check. We can see on the second line that every versions of the operation fail as faces get traversed by edges. This does not respect the geometry property of the mesh.



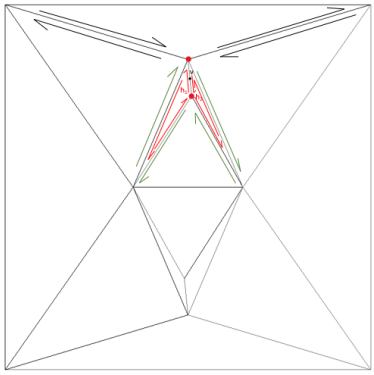
(a) The red edge is the edge to collapse using its mid-point. Its neighbouring half-edges have been made visible.



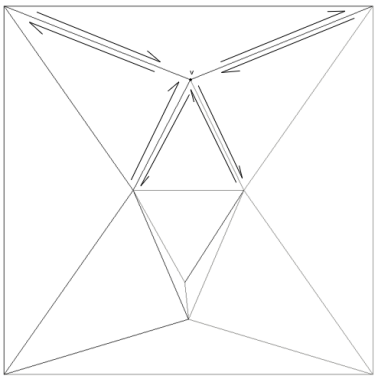
(b) First, a new vertex is created in the middle of the edge, and the half-edges starting from the two end-vertices are given this vertex as their new origin.



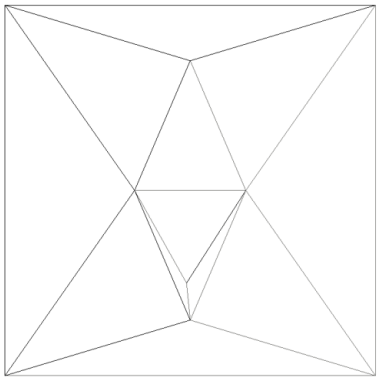
(c) Second, it is ensures that the third vertex of each of the neighbouring faces points toward an incident half-edges that would still exist after the operation.



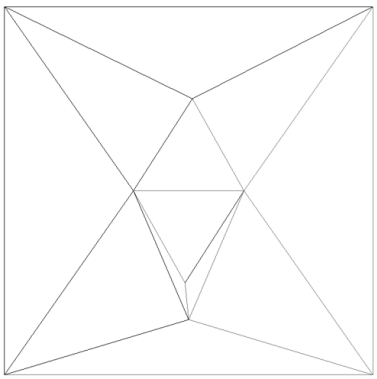
(d) The red half-edges and dots are then removed, and the opposites of the green half-edges are correctly set.



(e) Final mesh obtained after the operation is performed.



(f) Resulting mesh when collapsing the origin of h_1 on the one of h_2



(g) Resulting mesh when collapsing the origin of h_2 on the one of h_1

Figure 1.12: Step-by-step visualisation of a successful collapse transformation using half-edges

Edge split

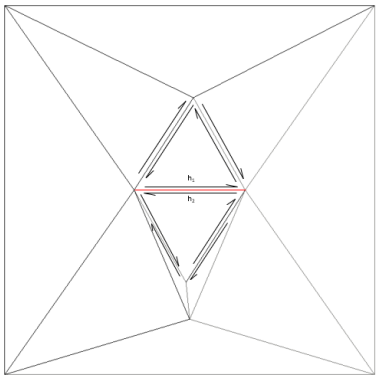
The edge split operation divides an edge as well as its two neighbouring faces into two parts. In the way it has been implemented, it does not require any additional check to be performed. Indeed, if the new point linking the two created edges is situated on the original edge, there is no case where the resulting mesh would not be manifold. As the implemented version always cuts an edge in its middle, the operation is always performed when it has to be applied. The detailed steps of the transformations, using the half-edges, are visible on figure 1.13.

Face split

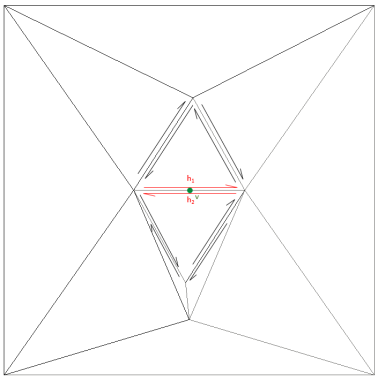
As its name suggests, the face split is similar to the edge split, as it consist in dividing one element, namely a face, into several ones. More precisely, a face split takes a points and divides the face containing it into three faces. A vertex is created at the position of the point, and three edges, so six half-edges, are created to connect the new vertex to the ones of the face which has been split. Similarly as the split of an edge, the only condition for a face split to be valid is that the point lies inside the face. It is indeed fairly easy to imagine the resulting mesh if one were to split a face using a point lying outside of it. The steps performed for a successful face split are shown on figure 1.14.

Vertex relocation

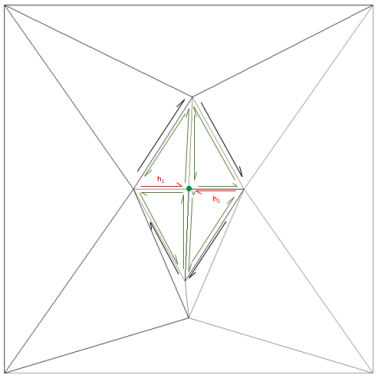
The last important operation which will be used in the following is the vertex relocation. This operation is also known in the literature as the mesh smoothing, vertex averaging, or vertex re-positioning [36,37].



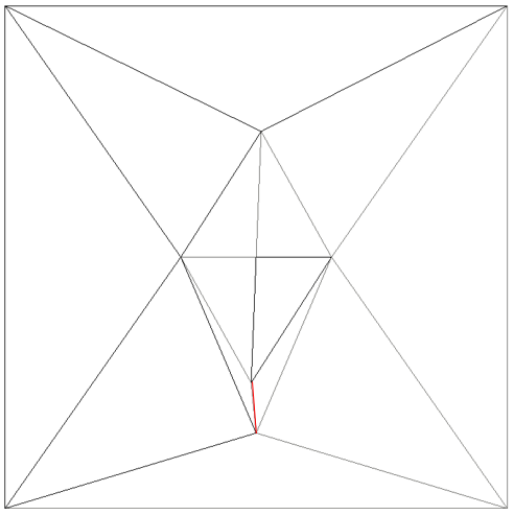
(a) The red edge is the edge to split using its mid-point. Its neighbouring half-edges have been made visible.



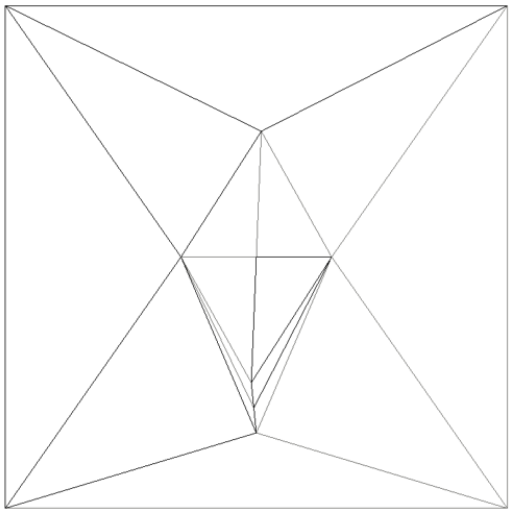
(b) The mid-point is shown by the green dots, and the two initial neighbouring half-edges are depicted in red.



(c) Then, six additional half-edges are created, and their origin are correctly set. Furthermore, the `next`, `prev` and `opposite` pointer are correctly set. Finally two additional faces are created, with their contained half-edge correctly identified.

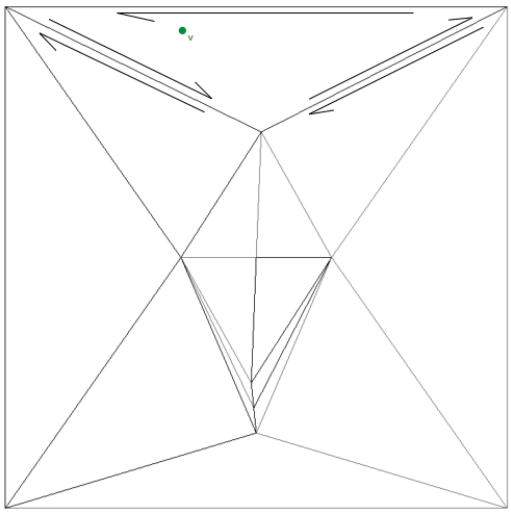


(d) After the first swap has been performed, another can be made on the edge marked in red.

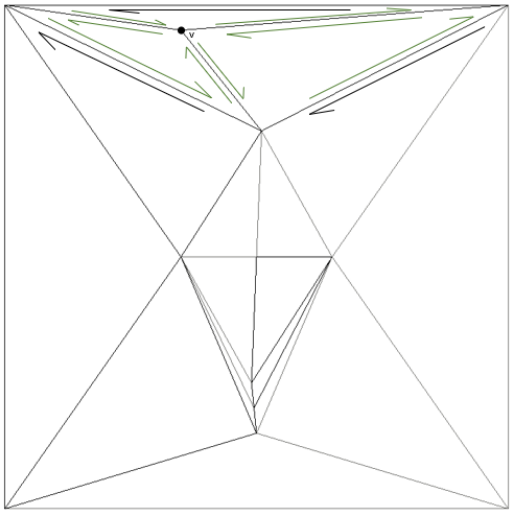


(e) Final mesh, resulting of the application of the two split operations

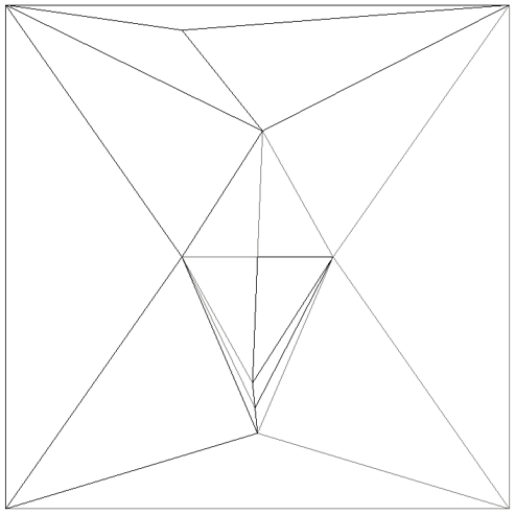
Figure 1.13: Step-by-step visualisation of an edge split transformation using half-edges, followed by a second split operation



(a) The green point is the point to include in its surrounding triangle, whose half-edges have been made visible.

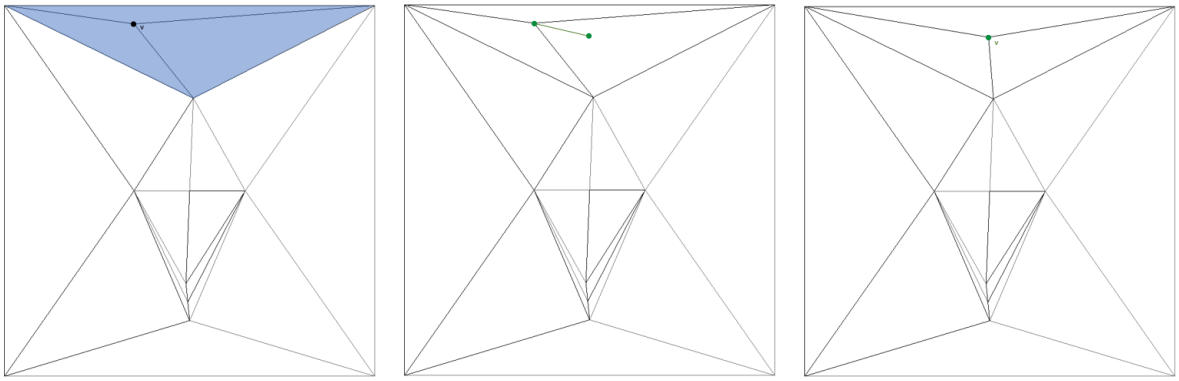


(b) A new vertex is created at the location of the point. Six half-edges are created and set correctly, while 2 additional faces are created. Those faces set to point to one of the half-edges they contain.



(c) Final mesh obtained after the insertion of the new vertex by means of a face split

Figure 1.14: Step-by-step visualisation of a successful face split transformation using half-edges



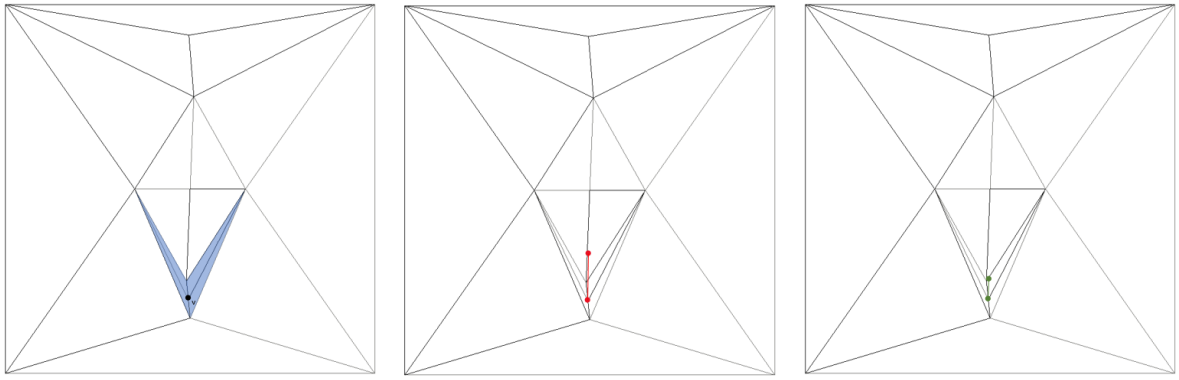
(a) The black vertex is the one to relocate. The cavity that is created when it is removed is highlighted in blue.

(b) The green line links the initial position of v with the center of mass of the vertices of the previously highlighted zone.

(c) The line is then bisected in order to find the location of the new vertex maximizing the minimal quality of its neighbouring elements. The vertex is then relocated at this location.

Figure 1.15: Step-by-step visualisation of a successful vertex relocation transformation using half-edges

The version implemented here is based on the one developed by Shephard and Georges [38]. It consists in tracing a line connecting the initial position of the vertex to the mean position of its neighbouring vertices. Then, it tries to find the position on this line which would maximize the minimum quality of the surrounding elements, by using a bisection method [39]. A successful relocation is shown at figure 1.15. A condition that needs to be verified is that the new position needs to be inside the cavity made by removing the vertex. Indeed, as visible on figure 1.16, the center of mass of the surrounding vertices can be situated outside of the cavity [40]. The procedure to take such cases into account mainly relies on the bisection method used and on the check to ensure that no triangle gets inverted, as for the collapse operation.



(a) The black vertex is the one to relocate. The cavity that is created when it is removed is highlighted in blue.

(b) The red line links the initial position of v with the center of mass of the vertices of the previously highlighted zone. It is visible that it is not contained in the aforementioned zone.

(c) The line is thus cut in order to be fully contained inside the previously highlighted zone. The bisection method then happens as usually.

Figure 1.16: Step-by-step correction of the line to be bisected when using the relocate vertex transformation

1.2.3 The function `add_points`

The `add_points` function is one of the most important function which is provided. Even though it is composed of only a few big steps, as visible on figure 1.17, most of them uses quite interesting though complex existing algorithm. The function takes as input a mesh, a list of points and an initial face. Consequently, a mesh has to be created first in order to use it. In the algorithm implemented, an initial mesh composed of two triangles forming a square is initially created. It is ensured that this initial square will contain all the points subsequently added. The first step of the actual algorithm implies a Hilbert sort of the points [41]. More precisely, it first finds the coordinate of each points on a Hilbert curve, and sort the points according to their corresponding coordinate value. Figure 1.18 shows such Hilbert curves. More details about them can be found in the specialized literature [41–46]. Without entering into all the details on how their are constructed, we can see that they are all composed of the same patterns, which is repeated in such a way that a whole square region gets filled by the lines. The reason why this sorting is performed is simple: it allows to include the points in an order such that each one is close to the next one. Doing so definitely helps the face finding to happen faster, and might also be helpful for the edge flip algorithm for Delaunay triangulation detailed later, since the flips take care of one localized region at a time. The second important step involves this finding of the face surrounding the current point `p`. Starting from the initial face `F`, one can easily traverse the whole mesh, as detailed in section 1.2.1. A check on each of the edges of a face is performed to see if the triangles formed by the two points of an edge and the current point to insert `p` form a triangle in the clockwise or in the counter-clockwise direction [47]. Based on this information for the three edges, it is possible to select the next face on which to iterate in order to arrive to the desired face. Once this is done, we set `F` to be equal to this face, such that the search for the next, supposedly close point will start from there.

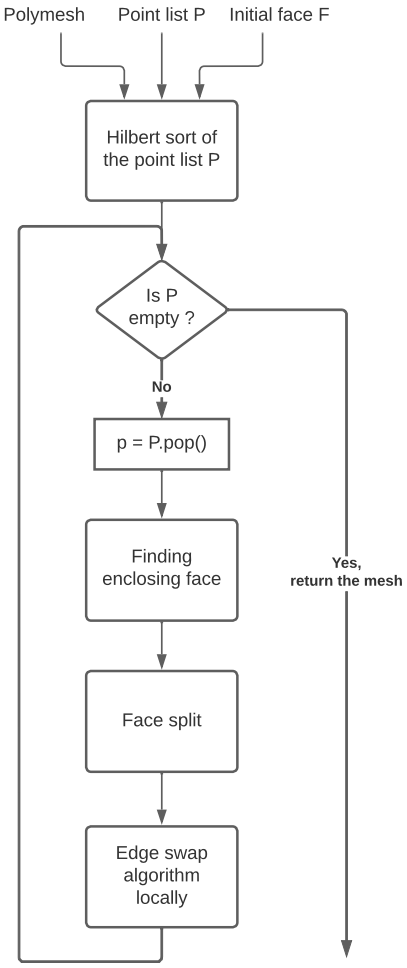


Figure 1.17: Flow-chart for the function `add_points`

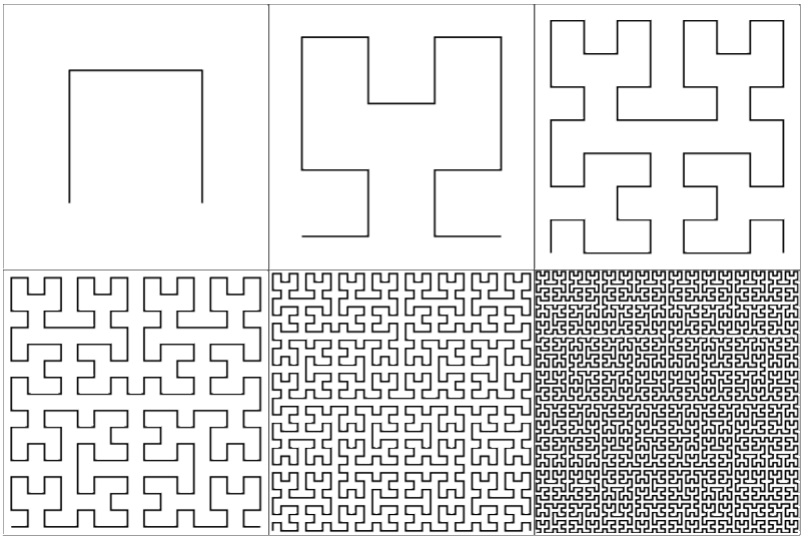


Figure 1.18: Example of the Hilbert curves obtained with different scale values

The final step performed for each point to insert is to split the face where it belongs, and to preform what is known as the edge swap algorithm for Delaunay triangulation . A Delaunay triangulation [48–50] is simply a mesh containing only triangles, and for which each 2D element respects the following condition: its circumcircle formed by its three vertices must not contain any vertex of another element [12]. Figure 1.19 depicts such a case and its counter case. Although different algorithm to obtain such triangulations [51], the algorithm used here to ensure this property for all the edges is the one developed by Charles Lawson [52–54]. It can be expressed, in our case, as follow:

1. Push the three half-edges created by the face split and starting from the new vertex in a stack, and mark them as inside the stack.

2. Pop the first edge in the stack.

3. Erase its mark stating that it is inside the stack.

4. If the Delaunay condition is ensured, we do nothing; otherwise, the edge is swapped, and the four neighbouring half-edges belonging to its two incident faces are included in the stack, but only if they or their opposite half-edge are not already marked as being in the stack.

5. Mark the included half-edges as being inside the stack.

6. Repeat step 2 until the stack is empty.

At the end, since there are no more half-edges in the stack, it implies that there are no more edge which does not respect the Delaunay condition. This directly lead to that fact that the whole mesh is a Delaunay triangulation. More information about the Delaunay triangulation and its interests are available in the literature [51, 55].
Finally, we can note that this function is not directly present on the figure 1.1. This is simply because, as for the local transformations and the half-edges structures, it is simply used by other some of these steps, but does not represent an really important step by itself. Currently, it is only used when reading a mesh from an existing file.

(a) (b)

Figure 1.19: Schematic representation of a mesh composed of two faces, with one of the circumcircle visible; the left mesh respect the Delaunay criterion, the right mesh does not

1.2.4 The definition of the size map

The size map can be defined in three ways divided in two categories, as shown on figure 1.1.
One easy way is to define a constant value for the element size, which will be the same for all edges during the whole simulation.
Apart from this trivial way, two other types of size map are included in the name "Size map structure" on figure 1.1. The first type of size maps are the ones defined in the Gmsh package [20]. Simply put, it is a function which basically takes the coordinates of a point and returns the desired size for the mesh at this position. Another way, which has been implemented specifically for the task at hand, uses the distance from a contour line to find the desired size. It is based on four parameters that should be defined by the user, depending of his needs. The first two are the minimum and the maximum size of the elements in the mesh. They go in pair with the other two, which represent two distances, named d_{min} and d_{max}. The required size at a given position is then defined as follow:

- If the minimal distance between the given coordinates and the contour line is lower than d_{min}, the desired size equals the minimum one.
- If the minimal distance between the given coordinates and the contour line is higher than d_{max}, the desired size equals the maximum size given by the user.
- Otherwise, the mesh size is linearly interpolated between d_{min} and d_{max}, using the value of the minimal distance as in the other cases.

In order to compute this minimal distance efficiently, a matrices of cells are used to store the points of the contour lines defined [30, 56].

This structure will be detailed later in section 3.2.2, when mentioning the different filtering algorithm implemented.

1.2.5 The smoothing of the bathymetry

As mentioned in the previously in the introduction, it is possible to apply a particular type of smoothing on the stored mesh, in order to flatten some zone where the bathymetry is considered as more noisy or simply less important. In order to perform such smoothing, two simple algorithm have been implemented.
The first one is the algorithm known as the Laplacian smoothing [57], for which a hybrid extension is already used to perform the vertices relocation. Its particularity is that is only applied on the depth of each points, modifying thus only the vertical coordinate and without touching to the horizontal one.
The working of this algorithm is quite simple. for each vertex, it takes the average depth of all its surrounding vertices, and perform the following computation:

$$v_{depth} = v_{depth} + \alpha(vs_{depht} - v_{depth})$$

where v_{depth} is the depth of the vertex considered, vs_{depht} is the average depth of the surrounding vertices, and α is a weighting factor, which has been here set to $\frac{1}{2}$. The average depths are thus first computed for each vertex in a first loop, and then the mean between the current depth of a vertex and the corresponding average of its neighbours are set to be the new depth.
The second algorithm simply generalize the first one. Although the usual Laplacian smoothing could be simply run multiple times in order to perform a stronger smoothing and extending the range of the smoothing, another strategy have been used in order to do so. Instead of only looking to the neighbours, other layers of neighbours are considered. More precisely, if the number of layers equals 2, the average

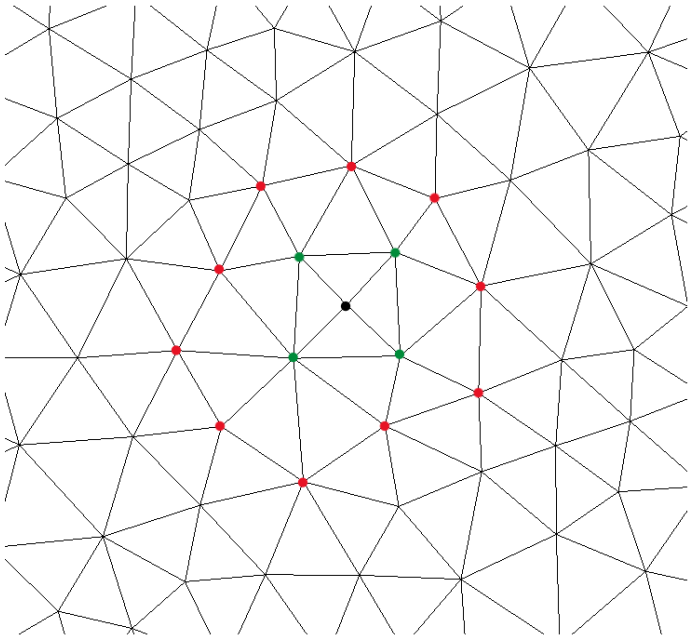


Figure 1.20: Schematic representation of the smoothing function based on the moving average algorithm; the black point is the current vertex, the green points are its first layer of neighbours, and the red points are its second layer of neighbours

of the neighbours' depths would be added to the one of the neighbours of the neighbours which has not been considered yet, and added to the current depth of the vertex. This value would then be divided by 3, and the resulting depth would be the one of the vertex considered. This could be seen as performing a moving average smoothing on the vertices of the mesh [58]. A depiction of this case is given on figure 1.20.

Those particular smoothing algorithm are especially useful in for the current task at hand. Indeed, as this smoothing is performed on the neighbouring vertices of a node, it is stronger in the zones where the mesh is coarser. By using the size map based on lines of isobathymetry defined in the previous subsection, it is possible to refine the mesh

near the contours that we think are the most interesting. If the Laplacian smoothing is performed then, the bathymetry will then get more smoothed on the zone that we find less interesting to fit precisely.
As this smoothing step can be applied in various places of the flowchart visible on figure 1.1, it has not been placed on it for a better readability.

1.2.6 The creation of meshes based on the user's needs

In order to create a mesh, only a reduced amount parameters need to be defined by the user. More precisely, only 4 parameters are required: the minimum values of the x and the y coordinates, as well as their maximum.
Initially, the mesh returned will only contain two big right-angle triangles. The user can of course define a size map and the main loop until the given size has been reached, while ensuring the good quality of the mesh. A last feature that can already be defined by the user is the bathymetry that has to be taken into account. This can be done in two ways: either by defining a function giving for each position x and y its corresponding depth, or by using a `.tiff` that will be read and interpolated if needed to obtain these depths. It is advised, although not required, to use a `.tiff` file such that its domain contains the coordinates given as initial parameters. Even though the required values would be extrapolated anyway via the nearest-neighbor method, it may avoid unpredictable behaviour.
Besides the definition of the size map detailed previously and the choice of the desired number of contours, nothing more is required in order to obtain a mesh adapted to the user's preferences. Nevertheless, another option than using four corners to get a rectangular mesh is also available. Instead, the user can also use an existing `.msh` file, in order to

only define the boundaries of the mesh. The inner parts of this initial mesh will simply be considered as empty, and the previous parameters such as the bathymetry and the size map can be added in order to perform the remeshing of the inner zone of those boundaries.

1.2.7 The reading of existing mesh files in `.msh`

Of course, the natural continuation to the reading of the border of a mesh is the addition of its inner nodes and edges. With the use of the `add_points` method referred previously, this task is quite straightforward in most of the cases. Indeed, if the hypothesis is made that the initial mesh is of good quality, the edge flip algorithm for Delaunay triangulation used when including new points should lead to a good mesh, thus resembling to the one given as input.
Nevertheless, there may be reasons motivating a user to retrieve a half-edges representation of the exact same mesh as the one passed as input. For example, a mesh may contain triangular zones which are actually not part of the studied area, such as islands. If such cases are not taken into account, an edge could unfortunately be swapped at the location of the islands. This would in consequence include these zones into the studied area, which may lead to wrong results and errors during the simulations. Although such occurrences are already fully treated with the current algorithm, this can still be enhanced. Indeed, the strategy followed currently mainly boils down to a comparison of all the faces in the mesh to be read with the mesh stored in half-edges. When an inconsistency is detected, swaps are performed in order for this first to be correctly represented. This tedious work, from which the full details will not be given, could be avoided in a quite simple way. A simpler strategy would be to consider the fact that the boundary edges are already marked so in the `.gmsh` file that have been currently used. As ensuring the consistency of the boundary would already suffice to keep the studied zone fixed, this could represent an interesting aspect

to improve in the near future, in order to make the reading of existing meshes faster.

1.2.8 The two types of methods to write a mesh in a file

Although only shown at the end of the algorithm on figure 1.1, a mesh can be written between each of the steps. This can be achieved thanks to two distinct function, each using a different coordinate system for the **.msh** files they return. Indeed, one uses the σ coordinate system and the other uses the z coordinate system. Both were already described in the introduction [15]. Another parameter can be used to modify the scale of the depth of the mesh. Indeed, it is not uncommon that the zone which is meant to be meshed has larger dimension in the x and y direction than in the z coordinate. In order to make the visualization easier, the scale can be tuned by the user. It is also useful to mention that this parameter is just present for convenience, as it is also possible to change the scale by using software such as Gmsh.
Additionally, the function to write a mesh using the z coordinate system has also be expanded with two features. The first one allows to fit the faces to their closest plateau. It takes as parameter a number of contours, and divides equally the total range of depth in order to compute as many levels. The minimum depth of each faces it then compared to the list of depth and the closest value in the list is the one given for the height of the current face. The second features takes care of a similar concern, but uses the structures of the contours in order to set each face to its corresponding plateau. This matter is detailed in section 3.2.4. Examples of meshes resulting from these two features will be shown and commented in section 3.3.

1.2.9 Description of the algorithm to improve or maintain the quality of a mesh

In this last subsection, the main algorithm used to update the mesh and improve its quality will e detailed. This algorithm as well as the quality measures used are strongly inspired from the work of Christophe Geuzaine and Jean-François Remacle [59].

The quality measures and their competing goals

In order to enhance the quality of the mesh, an accurate definition a what is expected needs to be given. More precisely, two features are usually desired for a mesh: it has to be consistent with the given size map, and most of its elements should be regular, i.e. should be close to equilateral triangles in our case. The second condition can also be defined as the aim of having a number of neighbours close to six for each vertex [36]. It is easy to understand that those two condition may contradict themselves. Indeed, when using a size map which is not constant, the fact of having elements with edges of different sizes is unavoidable. The update algorithm must in consequence find the best compromise in order to maximize both of the quality measures, or to favour one upon the other.
Mathematically, the first condition can be computed by performing the integral of the function given by the size map on a given edge. The value obtained would represent the edge length prescribed at the position of the edge. Nevertheless, in the program, the actual integral is not performed. Instead, the values prescribed by the size map are only taken at the two end-vertices of the edges and averaged. This strategy was only used in order to simplify the algorithm and to avoid taking too much time, mainly when the size map is based on the distance from contours. The length obtained is then used to divide the actual length of the edge, resulting in the adimensional length l_e of the edge.

In order to take the adimensional lengths into account at the mesh level, the following formula can be used to defined the efficiency index τ [60]:

$$\tau = \exp(\frac{1}{n_e}\sum_{e=0}^{n_e-1}\tau_e)$$

where n_e is equal to the number of edges. For a given edge index e, its corresponding value τ_e is defined as $\tau_e = l_e - 1$ if $l_e < 1$ and $\tau_e = \frac{1}{l_e} - 1$ otherwise. It ranges from zero to one, and should be as high as possible.
The second measure, to evaluate how close the elements are from regular ones, uses the ratio between the radius of the incircle and the radius of the circumcircle. This ratio is normalized so that it is equal to one when the triangle is equilateral, and to zero if its three vertices are aligned. By representing the three angles of a triangle by $\hat{a}$, $\hat{b}$ and $\hat{c}$, the normalized ratio can be defined as:

$$\gamma = 4\frac{\sin(\hat{a})\sin(\hat{b})\sin(\hat{c})}{\sin(\hat{a}) + \sin(\hat{b}) + \sin(\hat{c})}$$

Detailing of the main algorithm

Now that every ingredients have been detailed, the update algorithm can be described. A flow-chart representation is available at figure 1.21. It mainly consist of three loops: two loops on the half-edges, and one on the vertices.
The first loop on the half-edges tries to apply three local transformations. The first operation, the swap, needs to validate two checks in order to be performed. The first one is based on the quality measure γ defined previously. It is valid only if the minimum quality of the neighbouring faces of the half-edge after the swap is higher than this minimum quality before the swap. An additional verification also

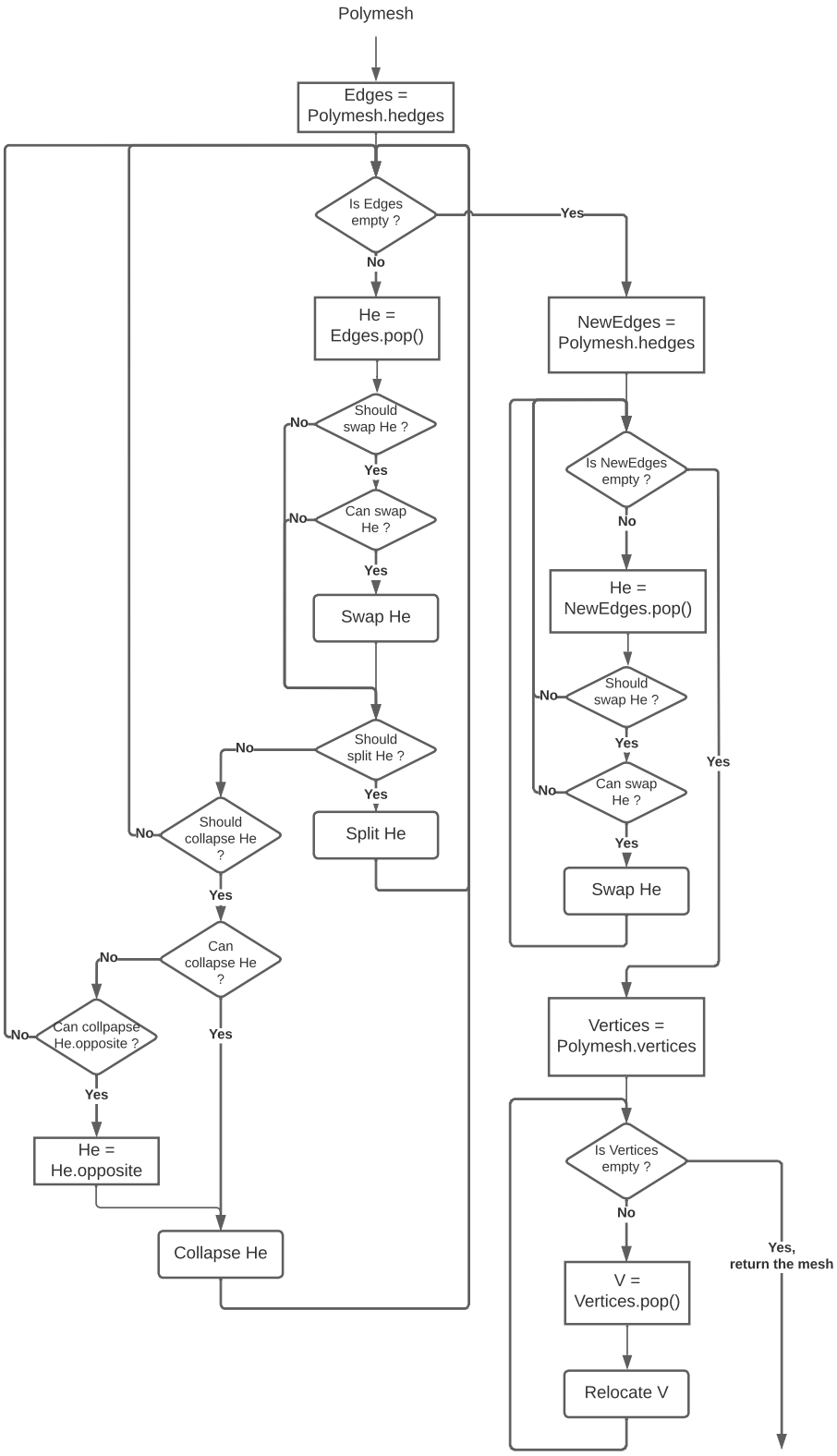
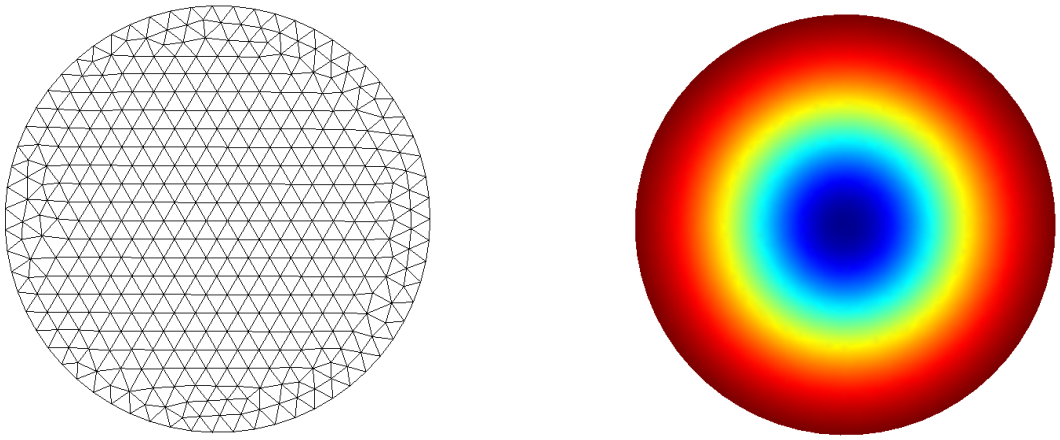


Figure 1.21: Flow-chart of the main loop used to update a mesh

checks that the edge is not part of the border, nor part of a contour line. After this first validation step, a second one is performed to ensure that the swap would keep the mesh manifold. This verification has already been explained when defining the swap operation. The second operation is the split. The check to validate its application requires to compute the adimensional length l_e of the edge [59]. Then, if its value is higher than the value of $2\sqrt{2}$, the edge is split, and the loop continues. Otherwise, the adimensional length is used to define if the edge should be collapsed. Indeed, the first validation step of the collapse operation is passed only if l_e is lower or equal to $\sqrt{2}$. This value is of course not random. It is the value of x which results in τ_e to be equal for both x and $2x$. Those thresholds are thus used in order to maximize the efficiency index τ. Of course, other checks need to be performed before collapse the edge, as detail when the local transformation has been defined.

After this first loop through all the half-edges, another loop through all the half-edges has been added. It passes through the previous half-edges, as well as through the ones created during the first loop. As only swaps are considered during this traversal, it mainly aims at maximizing the quality of the elements in the mesh than maximizing the consistence with the size map, which is here simply a design choice. Afterwards, a last loop traverse the list of vertices, and applies the vertex relocation operation if the vertices are not part of the border or of a contour.

Finally, the resulting mesh is returned, along with the quality indices τ, the average of the values of γ for all the elements, as well as the minimum value of γ encountered. In this way, the function can be easily placed inside a while or a for loop, and stopped when convergence is reached for one, two or all the indices, or after a certain number of iterations if such convergence can not be reached.



(a) (b)

Figure 1.22: On the left, initial mesh used in the section 1.3.1; on the right, the analytical bathymetry used

1.3 Resulting meshes

This section will show the types of meshes that can be obtained with all the algorithm detailed in this chapter. The main feature that are going to be shown concerns the creation of a mesh with user-defined parameters and the reading of existing meshes, as well as their adaption to a line of iso-bathymetry. The program can be used with a wide variety of meshes and bathymetries. In this section, three bathymetries will be used: an axisymmetric and smooth bathymetry on a circular mesh, the bathymetry of the Lake Tanganyika, and the bathymetry of a zone in the Seychelles.

1.3.1 The circular mesh

The initial mesh and its bathymetry used in this subsection are shown at figure 1.22.

On figure 1.23, we can visualize the some intermediate steps as well as the final mesh that can be obtained after setting the size map to a constant, smaller value than the initial one. The mesh algorithm converged after 8 steps, and we can see that most of the elements looks close to equilateral.

Another adaptation of the mesh uses the distance from a contour to define a size map. Figure 1.24 shows the computed contour on the final mesh obtained at figure 1.23. After only 6 steps, the algorithm converges to a mesh where the elements close to the contour are smaller, and gets bigger as they get further to it.

1.3.2 The meshes based on the bathymetry of the Lake Tanganyika

The initial meshes used in this subsection are the ones of figure 1.25, with the bathymetries of the zone they represent are shown at figure 1.26. It is useful to note that the first mesh has been obtained by using a constant size map and user-defined coordinates for the creation of the mesh, while the second has been obtained with Gmsh, using a size map where the elements gets bigger as they are far from the coast.

Figure 1.27 shows the adaptation of the first mesh based on a constant size map. The process required 10 iterations before converging. After this first adaptation step, the mesh has been reused in order this time to adapt to a line of iso-contour. The successive iterations and final mesh can be seen on figure 1.28. It is visible from both figure that meshes are efficiently refined, and that the last adaptation effectively produces smaller elements around the displayed isobaths.

As a last result for this geographical zone, figure 1.29 shows the adaptation process of the second mesh of figure 1.25. This adaptation is performed based on a deeper contour lines than for the previous one. Also, it is important to note that the boundary of the zone meshed has been effectively conserved until to end of the algorithm, which is a requirement of the implemented algorithm.

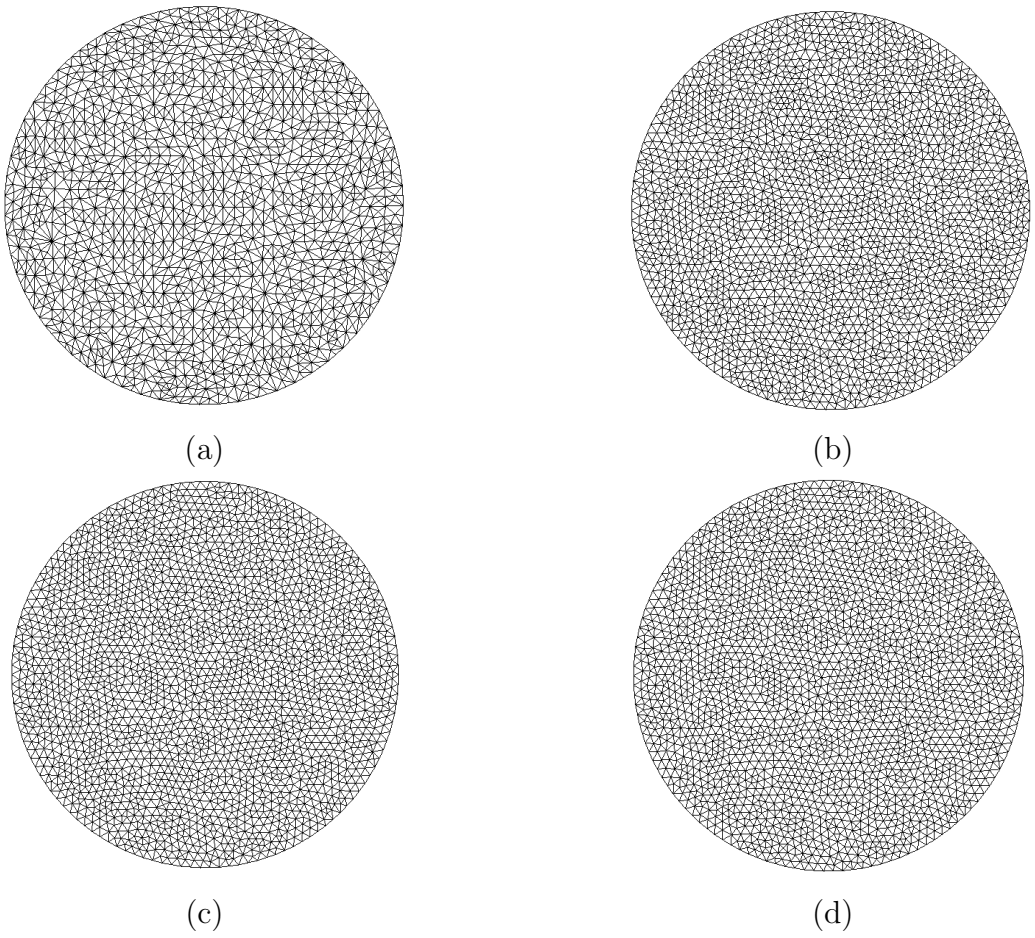


Figure 1.23: Refinement of the initial mesh on figure 1.22 with a constant size map; after (a) 1 iteration, (b) 3 iterations, (c) 6 iterations and (d) 8 iterations

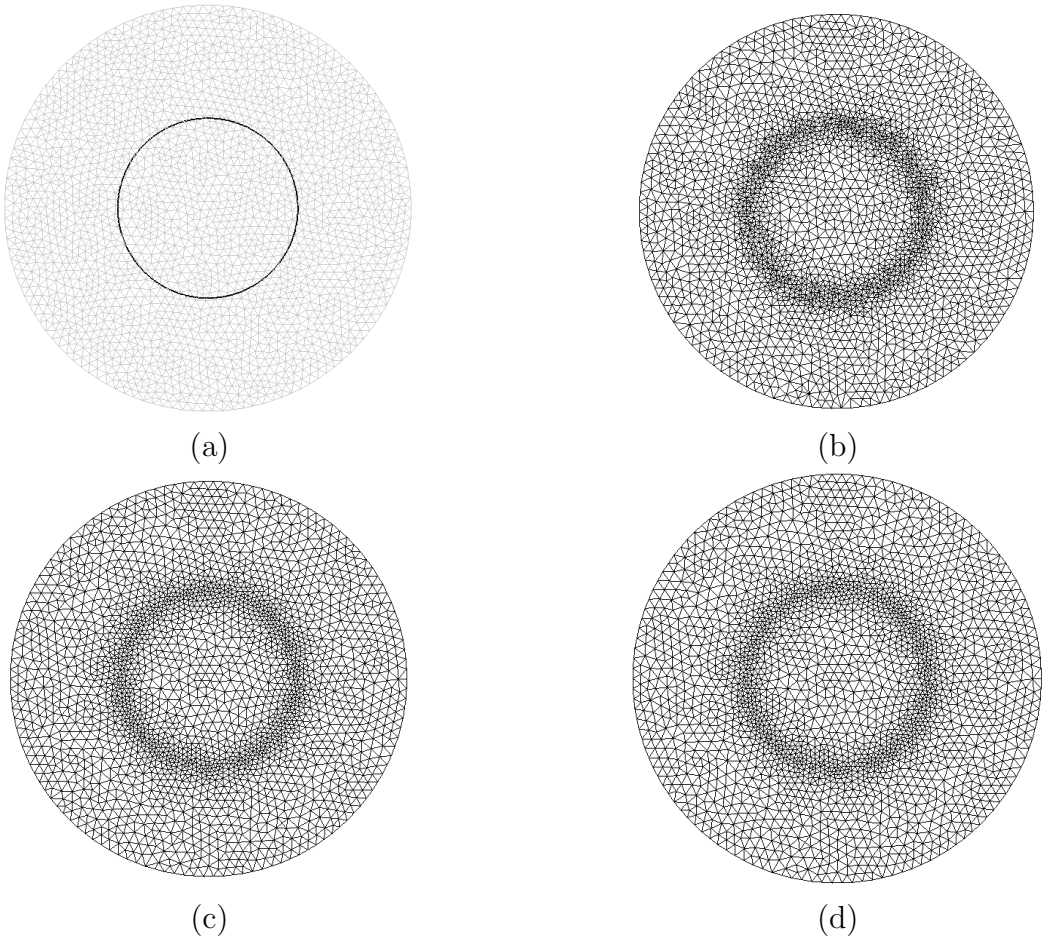
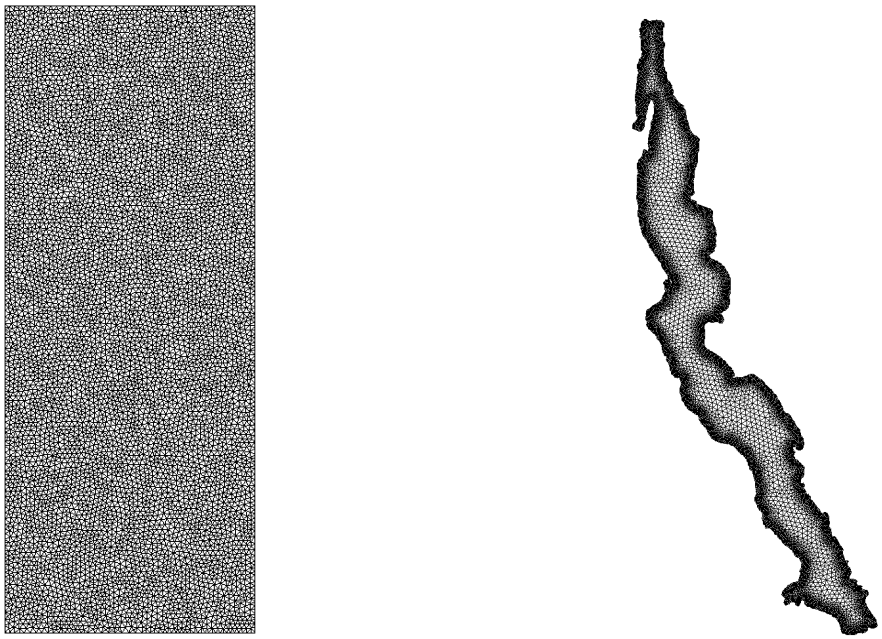
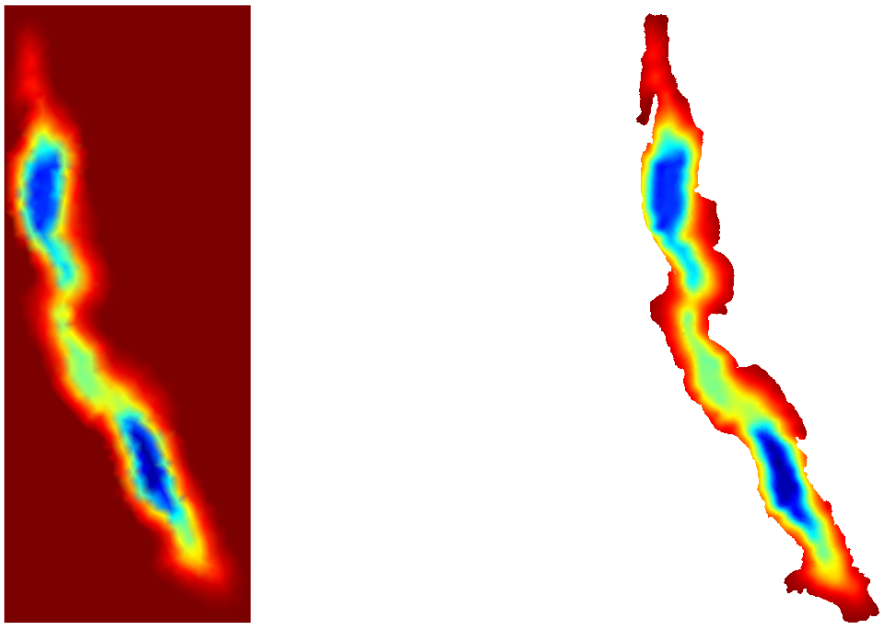


Figure 1.24: Refinement of the resulting mesh of figure 1.23 based on a contour line; (a) initial contour line, and after (b) 1 iteration, (c) 4 iterations and (d) 7 iterations



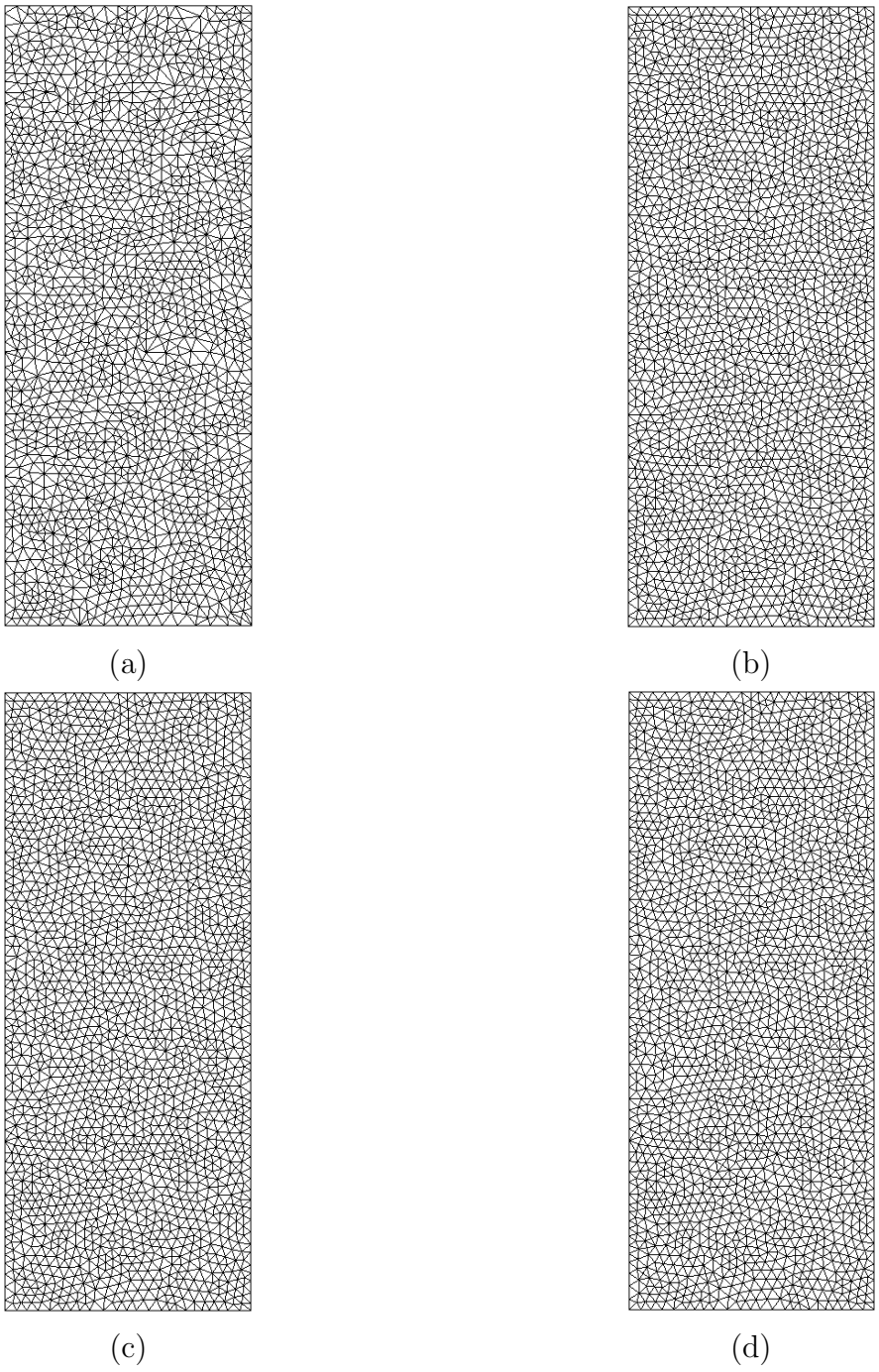
(a) (b)

Figure 1.25: Initial meshes of the Lake Tanganyika zone used in section 1.3.2



(a) (b)

Figure 1.26: Corresponding bathymetries of the meshes in figure 1.25



(a) (b) (c) (d)

Figure 1.27: Refinement of the initial mesh on figure 1.25a based on a constant size map; after (a) 1 iteration, (b) 4 iteration, (c) 6 iterations and (d) 10 iterations

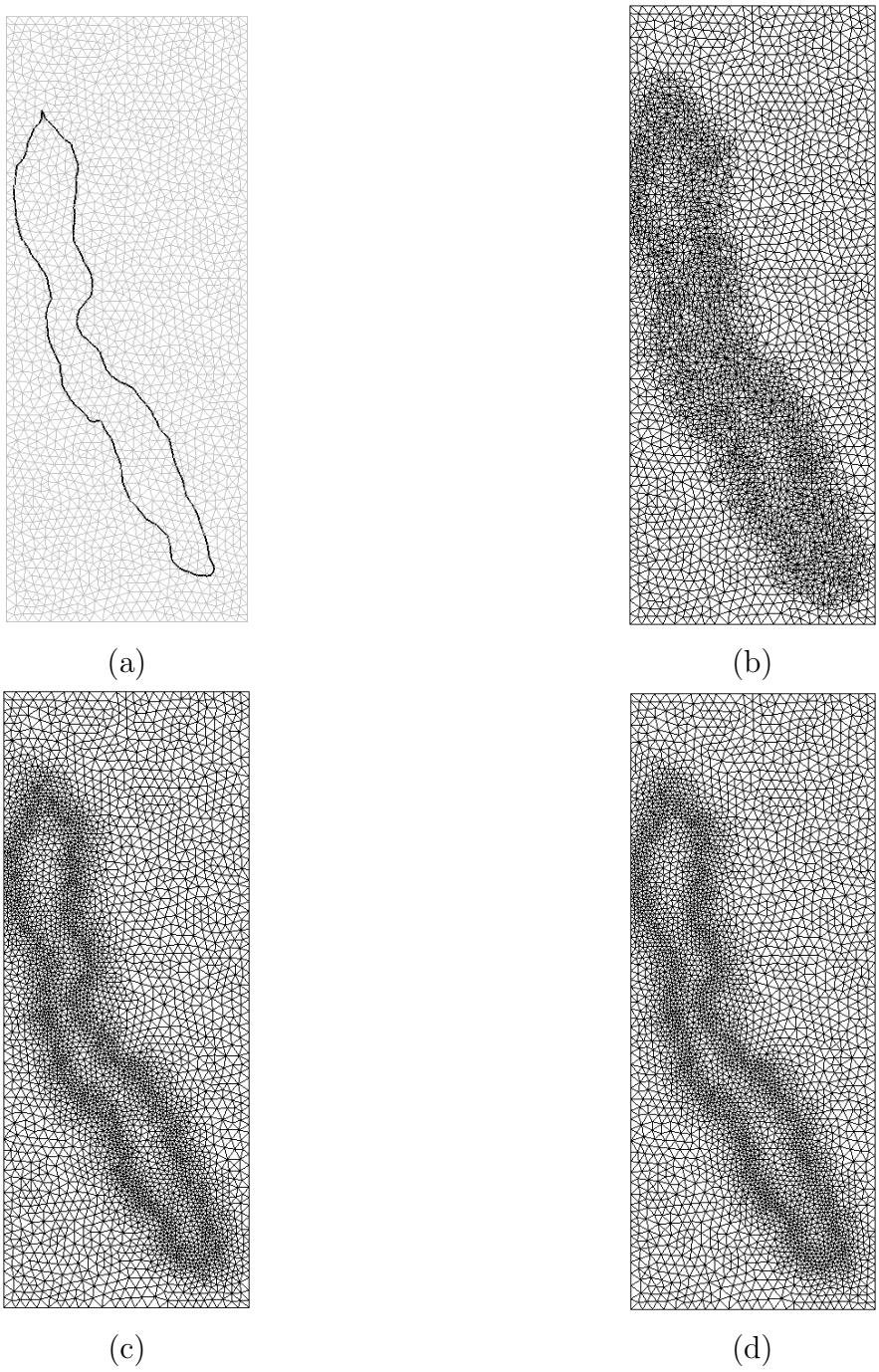
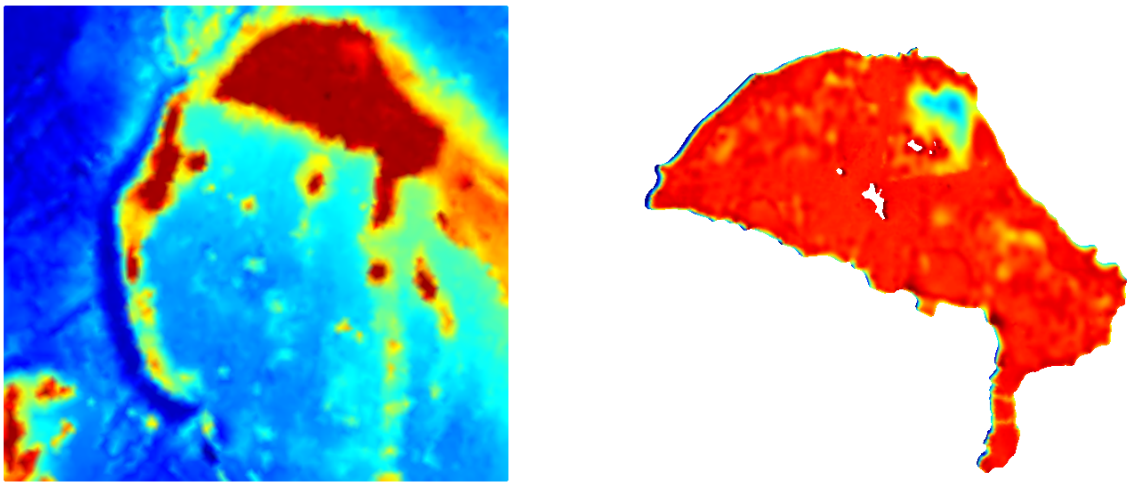


Figure 1.28: Refinement of the resulting mesh on figure 1.27 based on a contour line; (a) initial contour line, and after (b) 1 iteration, (c) 3 iterations and (d) 8 iterations

(a) (b)

Figure 1.29: Refinement of the initial mesh on figure 1.25b based on a contour line; (a) initial contour line, and (b) after 8 iterations



(a) (b)

Figure 1.30: Bathymetries used in section 1.3.3

1.3.3 The meshes based on the Seychelles bathymetry

Finally, a last zone on which the program has been tested is situated in the Seychelles. The bathymetries are available at figure 1.30.

The first results were obtained by defining the coordinates of the corners of the zone in order to create an initial mesh. The whole process, starting from a simple, big square zone and ending with a mesh of a good quality is shown at figure 1.31, and converged after 11 iterations. This newly created mesh can be considered of good quality since most of its elements are indeed close-to-equilateral. It can be furthermore refined, until obtaining the mesh of figure 1.32, which still keeps a good quality and proves that the algorithm are able to deal with a large amount of elements. As a last refinement, we could also fit it to computed contour lines. This process, as well as the lines on which the size map is based, are shown at figure 1.33, and shows again an accurate and correct working of the implemented functions.

As a next interesting example, the steps obtained when starting from

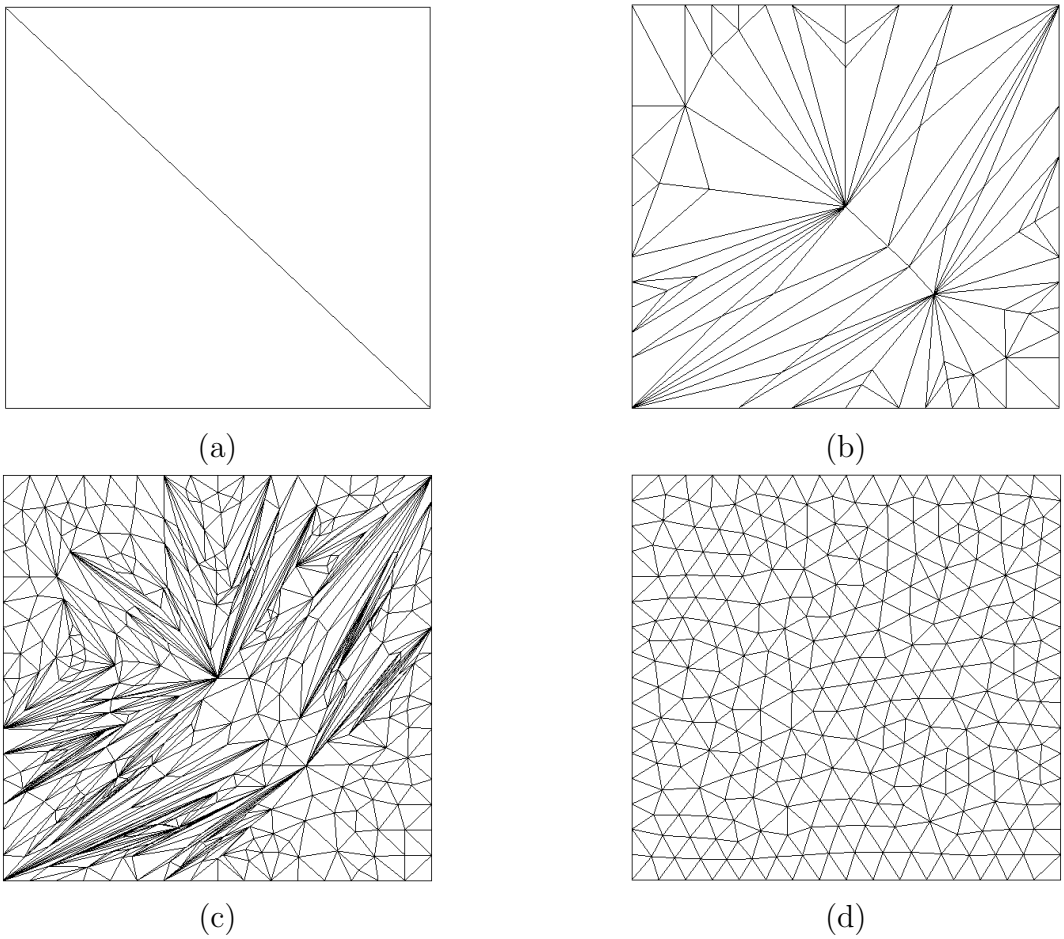
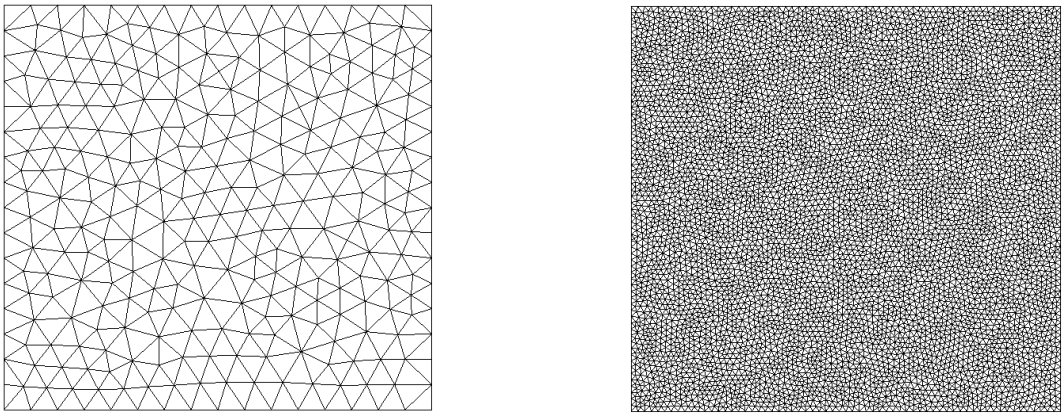
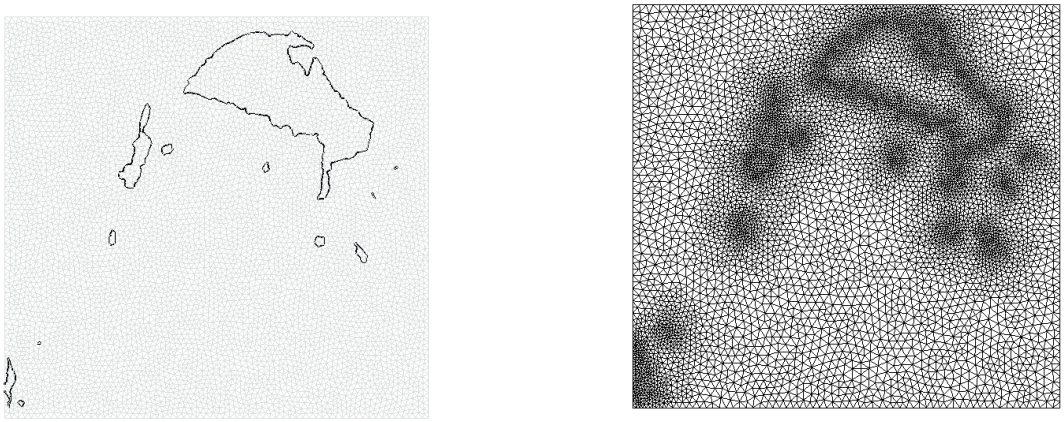


Figure 1.31: Creation of an initial mesh based on the coordinates of the four corners of the bathymetry visible on figure 1.30a and based on a constant size map; (a) resulting mesh at creation, after (b) 2 iteration, (c) 4 iterations and (d) 11 iterations



(a) (b)

Figure 1.32: Refinement of the resulting mesh on figure 1.31 based on a constant size map; (a) initial mesh used, and (b) after 9 iterations



(a) (b)

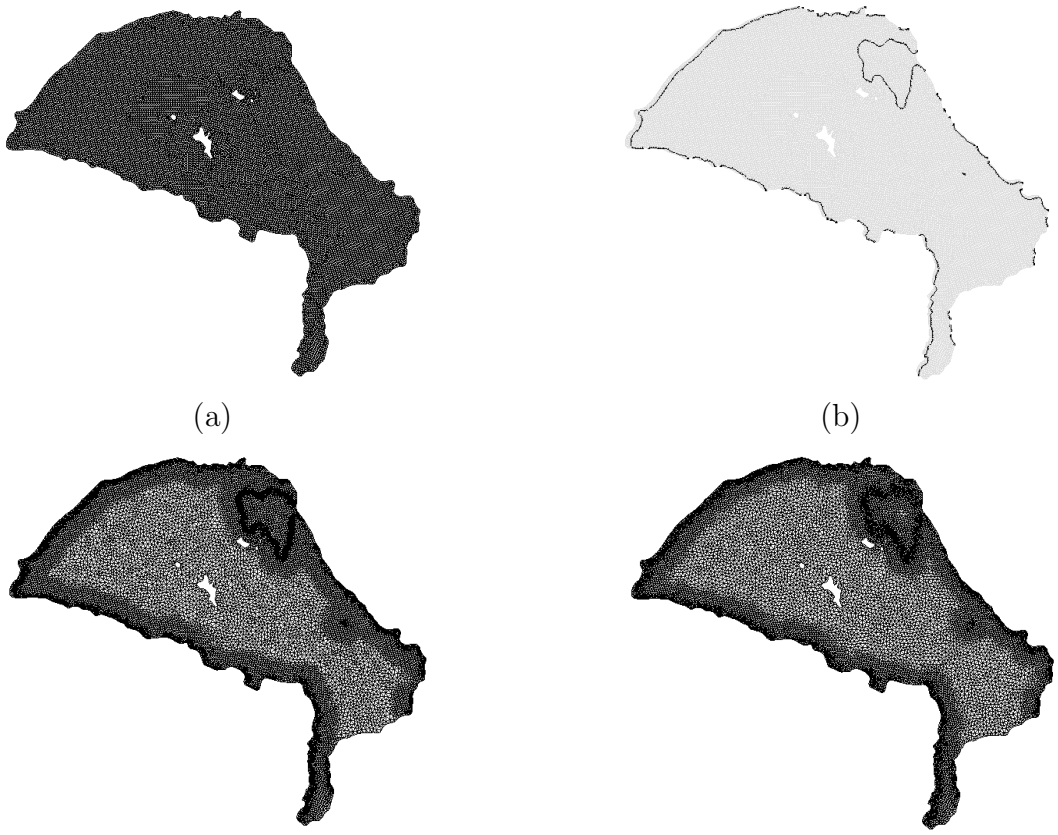
Figure 1.33: Refinement of the resulting mesh on figure 1.32 based on a contour line; (a) initial contour line, and (b) after 11 iterations

a fine mesh computed with Gmsh to a coarser mesh with a size map based on contour lines is shown at figure 1.34. Again, it is interesting to see that the initial border of the provided mesh are correctly conserved during the whole simulation.

Finally, a last interesting results that can be visualized is the effect of the smoothing function on an irregular bathymetry such as the one of the Seychelles. We used the final mesh shown at figure 1.33b, and applied the generalized smoothing algorithm up to a number of 2 layers. The resulting bathymetry is available at figure 1.35. We can see that, as the mesh was refined around a given level of contour, the zone where the mesh was coarser were more strongly affected by the smoothing operation, as desired. Furthermore, there is less peaks than previously, and the one still presents are lower. Such a smoother bathymetry has been used in Chapter 3 in order to obtain the desired, clean contour lines.

1.4 Discussion on the results

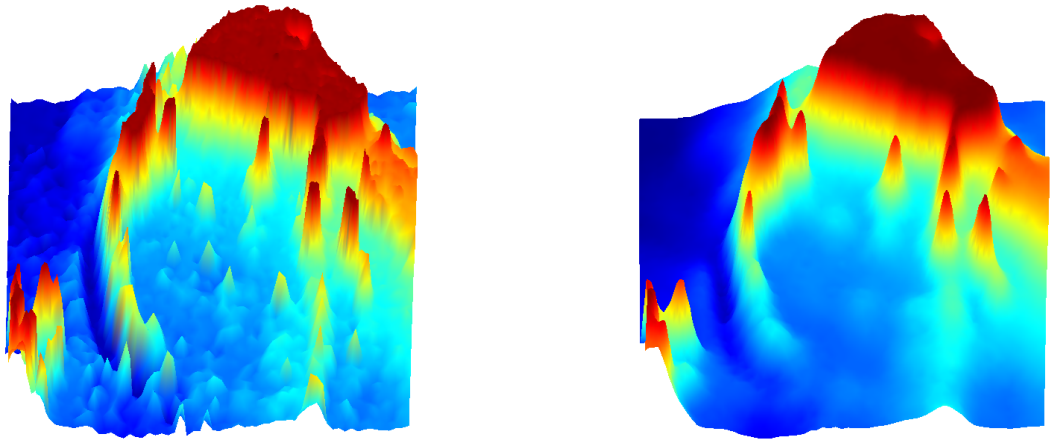
Although this chapter concerned general and well known algorithms, the obtained results shows their correct working. In consequence, they will be perfectly usable when additional algorithm will be implemented in Chapter 3. It is also interesting to mention that, in addition with the usual functions to read, write and update meshes, some convenient function for the task at had have also been specified and included. We could cite the size map based on contour lines, which goes perfectly in pair with the smoothing algorithm used and the task at hand in this master thesis. It is furthermore visible through the results obtained in this chapter that those work indeed as expected. Some results, though, have not been showed in the previous section. Indeed, the difference between the two types of writing a mesh have not been clearly identified. Nevertheless, the writing of meshes using z coordinates will be



(a) (b)

(c) (d)

Figure 1.34: Refinement of a mesh created with Gmsh based on a contour line; (a) the initial mesh, (b) the initial contour line, and after (c) 1 iteration and (d) 9 iterations



(a) (b)

Figure 1.35: Visualisation of the effect of smoothing when using the version inspired by the moving average strategy with a number of 2 layers; on the left the initial shape of the bathymetry of the Seychelles, on the right the resulting bathymetry after smoothing

extensively used in Chapter 3. Therefore, it would have been unnecessary to make this chapter heavier than it should be by showing quite trivial results as those.

Chapter 2

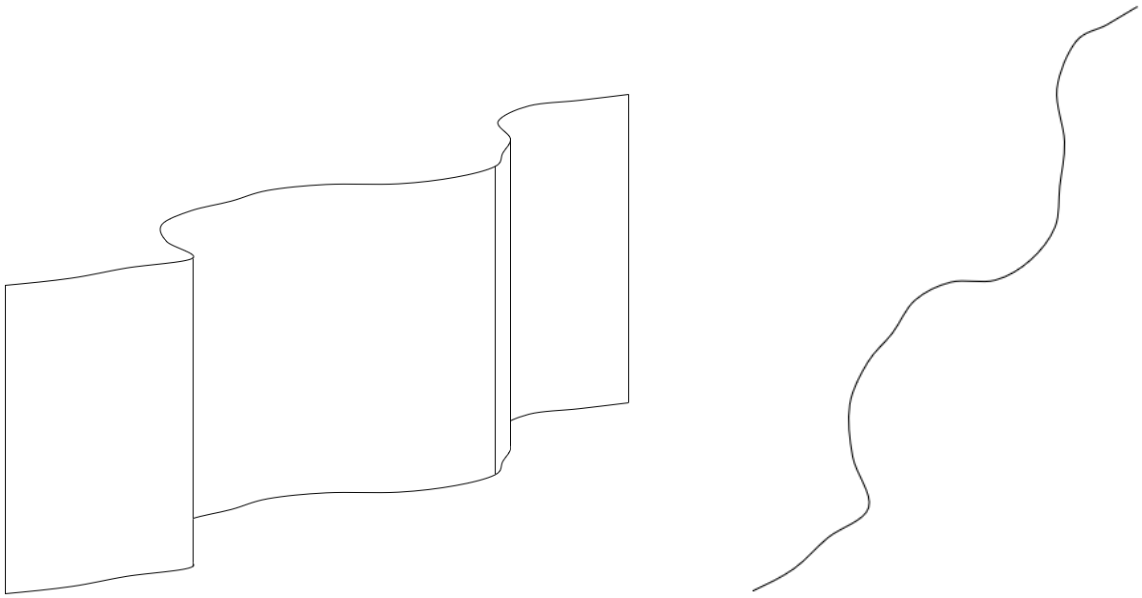
Validation of the concept on a simple test case

This chapter aims to test the idea on the base of this thesis in order to check if it might be an interesting add-on to the SLIM software [21] or to other finite element programs [61–63] for real-life cases. To perform such a validation, the contour-fitted meshing strategy will be applied on a simple, axisymmetric case from which it is possible to compute the analytical solution. After that, the three-dimensional finite element mesh generator Gmsh [64] will be used to obtain an initial, random mesh in a circular area, and another mesh where elements have been forced to fit along concentric circles, representing the chosen lines of isobathymetry. The axisymmetric loads will then be applied, and a simulation will be run for both meshes using the SLIM software. The goal is to verify that, using z coordinates, the numerical dissipation [65] of the total energy decreases when using the second, contour-fitted mesh instead of the random one.

2.1 Introduction

As the whole concept studied in this master thesis had not been proven to be effective yet, it was utterly needed to test it before going further. Indeed, even though the initial structures detailed in the first chapter have already been made available, the actual algorithm to automate the process of finding the isobaths still needed to be added. This may have implied a large amount of useless work if an adequate validation of the concept had not been developed first. The obvious solution to perform such a task was to test it on a really simple case, with a well known solution, allowing to start from an equilibrium state. Throughout long time simulations though, it is expected that this equilibrium will not be maintained by a numerical solver. Indeed, even if the initial solution is a steady-state solution of the equation, the software programs based on finite element suffer from a small lack of precision due to the fact that numbers have to be expressed with a fixed number of bits [66], while real number could in reality require an infinite among of digits.Nevertheless, these seemingly negligible errors in the beginning can grew up to really large values when nothing is done to prevent this kind of behaviour.
Furthermore, the meshes developed in this thesis are meant to be used with the z coordinate system [15]. As detailed in the introduction, it creates columns composed of a varying number of prism, but having all similar intermediate depths, their difference in number in each column of water depending on the total depth of the column. For the sake of illustration, we can imagine the case of a submarine cliff with a close-to-infinite gradient, as depicted on figure 2.1 in perspective and from the top. If this zone had to be extruded at constant z coordinates, the user could end up with a mesh as the one visible on figure 2.2. The smooth line that originally separated the two depth levels has now become completely irregular and contains several sharp turns. During the simulation, reaching convergence at those tight corners becomes a

really difficult task, while the line separating the two levels was much more smooth in the original bathymetry, and even though the mesh used was of a good quality. We can naturally imagine that the accumulation of errors at these zones is in consequence really large compared to the error accumulation on flat areas, or at the places where the transition between two levels happens more smoothly. These sharp corners would thus really need to be avoided.
Taking cares of these zones directly leads to the concept of contour-fitted meshes developed in this report. Indeed, when forcing the elements to lie on such lines where the depth slope is really strong, one could obtain a mesh such as the one depicted on figure 2.3. When looking now at the interface between the two levels, it easy to see that it is much more smooth compared to a mesh which does not take the same considerations. However, both meshes contain mainly elements of good quality, i.e. with roughly the same size and containing mostly close-to-equilateral triangles, and are consequently both appropriate to be used in simulations.
In the following of this section, this concept of adapting a mesh to the isobaths will be verified more formally, instead of only relying on some vague observations and conclusions. First, the actual test case will be fully detailed, along with its analytical solution for its elevation field and the prescribed velocity field. Then, the results obtained when running a simulation using SLIM will be shown and detailed, using both 2D and 3D simulations as the fourth version of the software already offered this possibility. Then, conclusions will be drawn on the potential effectiveness and interest of this original type of mesh in more realistic situations.



(a) View in perspective (b) Top view of the cliff

Figure 2.1: Depiction of a bathymetry with a steep slope

2.2 Materials and methods

This section will focus on the precise description of the test case. A field for the bathymetry and for the velocity will be chosen, and the corresponding elevation field will be expressed. Then, the setting of the simulation will be defined in order to obtain the results of the following section.

2.2.1 Description of the test case and its initial fields

The initial test case used to validate the solution has been chosen to be an axisymmetric problem with a well-known solution. The aim is to

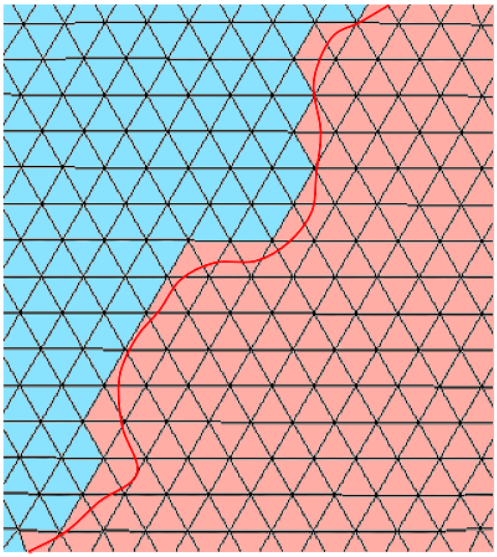


Figure 2.2: Non-fitted meshing of the bathymetry shown in figure 2.1; each color corresponds to a level of depth

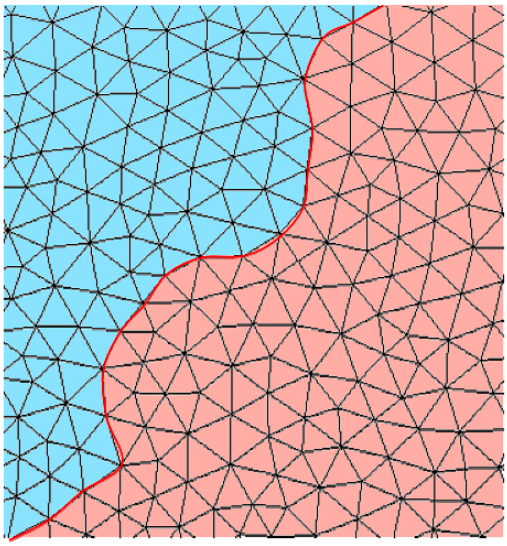


Figure 2.3: Contour-fitted meshing of the bathymetry shown in figure 2.1; each color corresponds to a level of depth

deploy the concept on SLIM, which simulates the behaviour of oceans and seas around the globe. One of the interesting possibility was to use the equation of the geostrophic equilibrium [67, 68], on a circular mesh and with an axisymmetric bathymetry.
The computation of the equations used as a base in this section are the shallow-water equations [69]. More precisely, we will start from the resulting momentum equation, which expresses as follow:

$$\frac{D\mathbf{u}}{Dt} + f\mathbf{k} \times \mathbf{u} = -g\nabla H \tag{2.1}$$

where $\frac{D\mathbf{u}}{Dt}$ is the material derivative [70] of $\mathbf{u}$ and which can also be expressed as $\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}$. As usual, g is the gravitational acceleration and f is the Coriolis coefficient, while $\mathbf{u}$ is the velocity vector.
The velocity vector that will be imposed, in order to keep the problem axisymmetric, will represent a rotational speed of the form:

$$\mathbf{u} = \begin{bmatrix} \omega y \\ -\omega x \\ 0 \end{bmatrix} \tag{2.2}$$

with ω the chosen rotational speed. Furthermore, as we aim for the solution in (dynamic) equilibrium, this velocity vector will be considered as constant with respect to the time, leading to the first term in the expanded form of the material derivative to be equal to zero, i.e $\frac{\partial \mathbf{u}}{\partial t} = 0$. Also, as the direction of the velocity vector $\mathbf{u}$ is totally angular, its gradient $\nabla \mathbf{u}$ is fully radial, and so both vector fields are tangent, leading to the second term of the material derivative equation $\mathbf{u} \cdot \nabla \mathbf{u}$ to be also null in this case.
In order to expand the right-hand term of equation 2.1, one can make use of the definition of $H = h + \eta$, where h corresponds to the height between the seabed and the surface of the water at rest and η corresponds to the elevation of this surface with respect to its resting level,

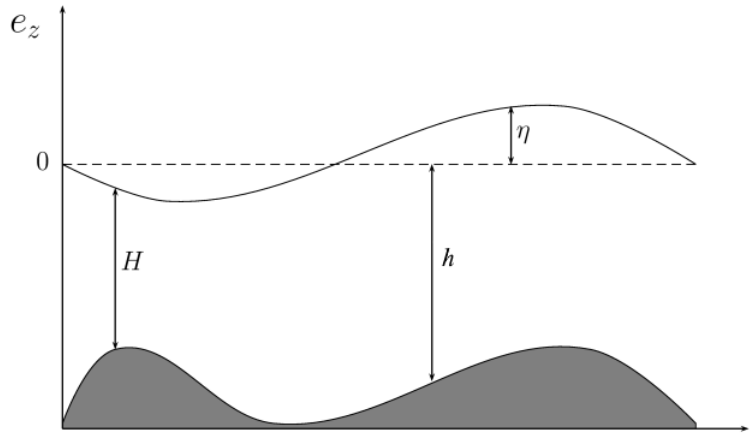


Figure 2.4: Example of a cut in the ocean where at y is equal to a constant cst. η represent the surface elevation, while h is the depth with respect to the reference level, and H the total depth [71]

as schematized on figure 2.4). In the current situation, h will be considered as constant for a given x and y coordinate, i.e. the bottom of sea does not change with respect to the time t. Moreover, we consider a bathymetry composed of flat zones, called plateaus, delimited by circular lines of infinite slope. This flat zones have obviously a gradient equal to zero. Furthermore, the gradient on a contour line is also null, as their derivatives are always equal to zero when the value of r tends to the radius of the contour in both direction. When applying the gradient operator to H, we thus obtain the following results:

$$\nabla H(x,y) = \nabla(h + \eta(x,y)) = \nabla h + \nabla\eta(x,y) = 0 + \begin{bmatrix} \frac{\partial \eta}{\partial x} \\ \frac{\partial \eta}{\partial y} \\ 0 \end{bmatrix} \tag{2.3}$$

When putting equations 2.2 and 2.3 into equation 2.1, and since in our case $\frac{D\mathbf{u}}{Dt}$ equals zero, we obtain:

$$\begin{bmatrix} 0 \\ 0 \\ f \end{bmatrix} \times \begin{bmatrix} \omega y \\ -\omega x \\ 0 \end{bmatrix} = -g \begin{bmatrix} \frac{\partial \eta}{\partial x} \\ \frac{\partial \eta}{\partial y} \\ 0 \end{bmatrix}$$

$$\Leftrightarrow \begin{bmatrix} f\omega x \\ f\omega y \\ 0 \end{bmatrix} = -g \begin{bmatrix} \frac{\partial \eta}{\partial x} \\ \frac{\partial \eta}{\partial y} \\ 0 \end{bmatrix} \tag{2.4}$$

This results in a system of two differential equations, which can be solved in order to obtain the elevation field in function of the position:

$$\begin{cases} f\omega x = -g\frac{\partial \eta}{\partial x} \\ f\omega y = -g\frac{\partial \eta}{\partial y} \\ 0 = 0 \end{cases} \Leftrightarrow \begin{cases} \int (f\omega x)\ dx = -g\eta(x,y) \\ \int (f\omega y)\ dx = -g\eta(x,y) \end{cases}$$

$$\Leftrightarrow \begin{cases} -\frac{f}{2g}\omega x^2 + C_x(y) = \eta(x,y) \\ -\frac{f}{2g}\omega y^2 + C_y(x) = \eta(x,y) \end{cases}$$

where C_x and C_y are constants with respect to their variable in subscript. By putting those two relations together, we obtain the expression of the η up to a constant value C:

$$\eta(x,y) = -\frac{f}{2g}\omega x^2 - \frac{f}{2g}\omega y^2 + C \Leftrightarrow \eta(x,y) = -\frac{f}{2g}\omega(x^2+y^2) + C \tag{2.5}$$

Before finding the expression of the constant, it is useful to remind that, as expected for an axisymmetric problem, the expression of the elevation field only depends on r, since the well-known relation $r = \sqrt{x^2+y^2}$ [72] can easily be replaced in equation 2.5. The elevation in

polar coordinates can then be expressed as:

$$\eta(r,\theta) = -\frac{f}{2g}\omega(r^2) + C \tag{2.6}$$

Finally, boundary conditions have to be imposed in order to define the constant C. In this case, we will simply consider that the domain exactly encloses the perturbations provoked by the chosen velocity field. In consequence, we have that $\eta(r,\theta)|_{r=R} = 0$ for all the values of θ. This leads to C being equal to $\frac{f}{2g}\omega(R^2)$. The final expression of η in polar coordinate can now be obtained as:

$$\eta(r,\theta) = \frac{f}{2g}\omega(R^2 - r^2) \tag{2.7}$$

We still have to take care of the continuity equation. As written previously, the field of bathymetry considered in this section is axisymmetric. This is actually required if we want this relation to be valid, given the chosen rotational velocity. Indeed, the equation of mass conservation under the shallow-water hypothesis can be expressed as:

$$\nabla \cdot (H\mathbf{u}) = 0 \tag{2.8}$$

This relation can also be written as:

$$\mathbf{u} \cdot \nabla H + H\nabla \cdot \mathbf{u} = 0 \tag{2.9}$$

The definition of the chosen velocity field $\mathbf{u}$ has been given in equation 2.2, and leads to the following for the second term of the left-hand side of equation 2.9:

$$H\nabla \cdot \mathbf{u} = H\ (\frac{u_x}{\partial x} + \frac{u_y}{\partial y}) = H\ (\frac{(\omega y)}{\partial x} + \frac{(-\omega x)}{\partial y}) = 0 \tag{2.10}$$

This means that, in order to satisfy equation 2.8, we need to have that $\mathbf{u} \cdot \nabla H$ equals to zero, meaning that the variation of the field H

should be tangent to the vector field $\mathbf{u}$ everywhere in the domain. As the velocity $\mathbf{u}$ only depends on r in cylindrical coordinates, and as we found previously that the elevation field η only depends on r, i.e. its gradient is strictly radial, this leads to the fact that h should also only depends on r, and thus should be axisymmetric.

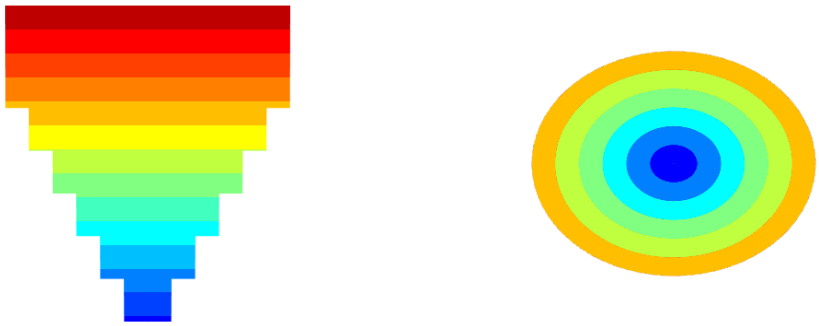
2.2.2 Setting of the simulations

After having found the expression of the elevation field η, the parameters of the simulation still need to be defined.
As mentioned previously, it is required to have an axisymmetric field for the bathymetry. Also, this field has been defined as being a composed of piecewise constant functions, representing plateaus, in order for its gradient to be also neglected. As the aim of the meshing strategy, fitted to the isobaths, is to decrease the error when facing high gradient of this field, this field containing a finite number of concentric contour lines of infinite gradient seemed to be a convenient option. Such kind of bathymetry is depicted in figure 2.5, with a number of flat zones, or plateaus, equals to 6. In order to fully define the simulation case, we can distinguish seven parameters:

1. R: the radius of the domain.

2. $n_{contour}$: the number of contours to use. Arbitrarily, it has been defined that there would be one plateau more than the number of contours, so that each isobath are situated between two plateaus. For example, a characteristic value for $n_{contour}$ for the shape drawn on figure 2.5 would be 5.

3. lc: The characteristic length of the mesh elements in the simulation. As we use 2D triangles, it is simply the desired size of their edges.



(a) Side view (b) Top view

Figure 2.5: Example of a bathymetry that may have been used for the current test case

4. h: The depth variation of each step of bathymetry, defined to be constant in our case.
5. h_{init}: The depth of the highest plateau.
6. ω: The magnitude of the rotational speed.
7. dt: The time step to use during the simulation.

Although simulations have been launched for various values of these parameters, we will only retain one as most of the results were quite

similar. The chosen parameters in the following of this chapter are:

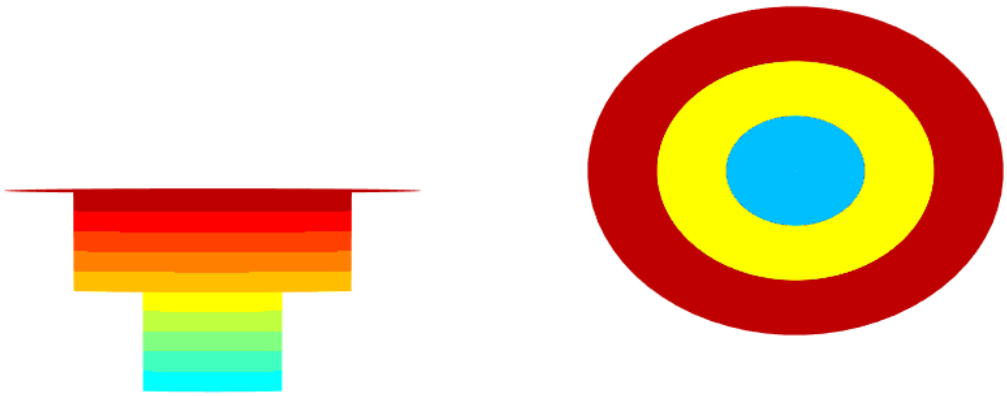
$$\begin{aligned} R &= 5e5 \text{ [m]}; \\ n_{contour} &= 2 \text{ [-]}; \\ lc = \frac{R}{10} \text{ [m]} &= 5e4 \text{ [m]}; \\ h &= 5 \text{ [m]}; \\ h_{init} &= 12 \text{ [m]}; \\ \omega &= 2e-6 \text{ } [\frac{\text{m}}{\text{s}^2}]; \\ dt &= 10 \text{ [s]}; \end{aligned}$$

Figure 2.6 shows the resulting seabed using these values. The careful reader might have noticed that the surface of the border has not been displayed on this plot, contrarily to the one on figure 2.5.

As there were no certitude at this point about the interest of contour-fitted meshes, the mesh-generator Gmsh has been used to compute the triangulations used in this section, as it already allowed to fit a mesh on given analytical curves such as the circles in this test case. The resulting meshes are shown on figure 2.7.

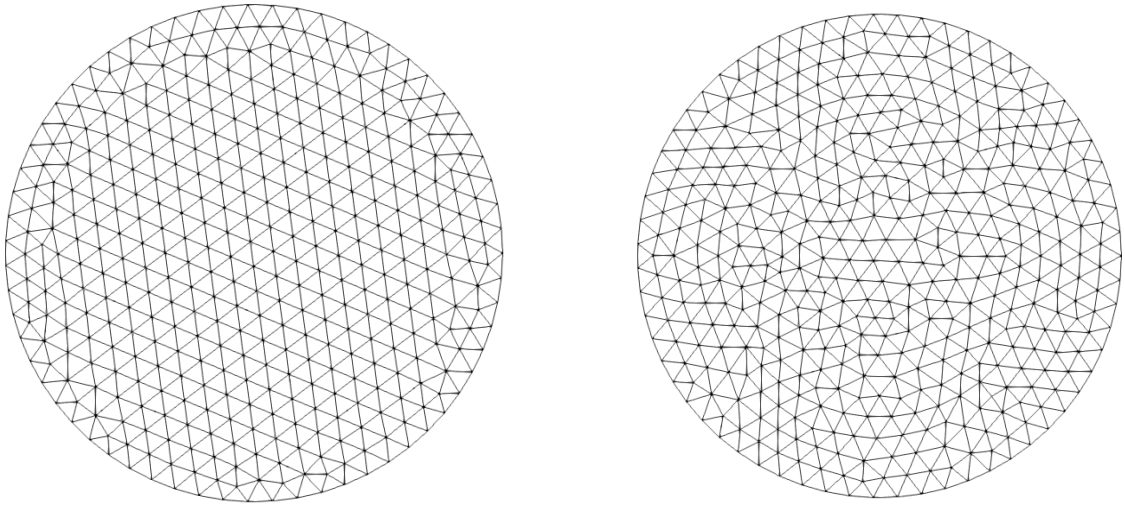
Although this might seem trivial, it is important here to choose at which depth an element crossing two or more contours will be situated. In this section and for the rest of this document, the height of a triangle will always be the one of the lowest plateau that it crosses. The bathymetries obtained when applying this to our two meshes are available at figures 2.11a and 2.11b.

Finally, we can insert those values into the equations 2.2 and 2.7, in order to obtain the initial velocity and the elevation fields displayed in figures 2.9 and 2.10.



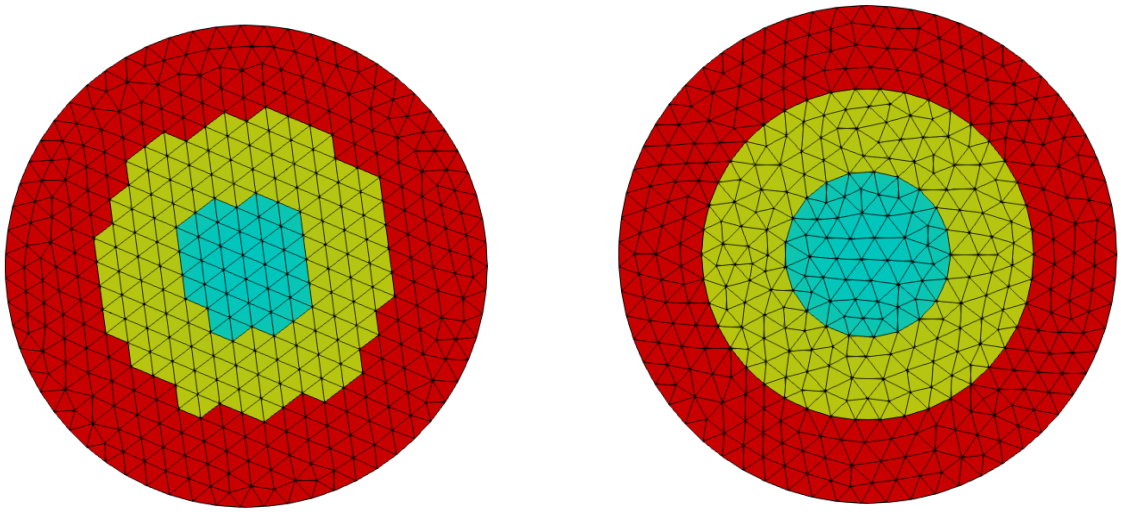
(a) Side view (b) Top view

Figure 2.6: Actual bathymetry used for the current test case



(a) Mesh not fitting the contour lines (b) Mesh fitting the contour lines

Figure 2.7: Obtained meshes for the the current test case



(a) Mesh not fitting the contour lines (b) Mesh fitting the contour lines

Figure 2.8: Obtained meshes for the the current test case, with their faces set to the depth of their lowest vertex

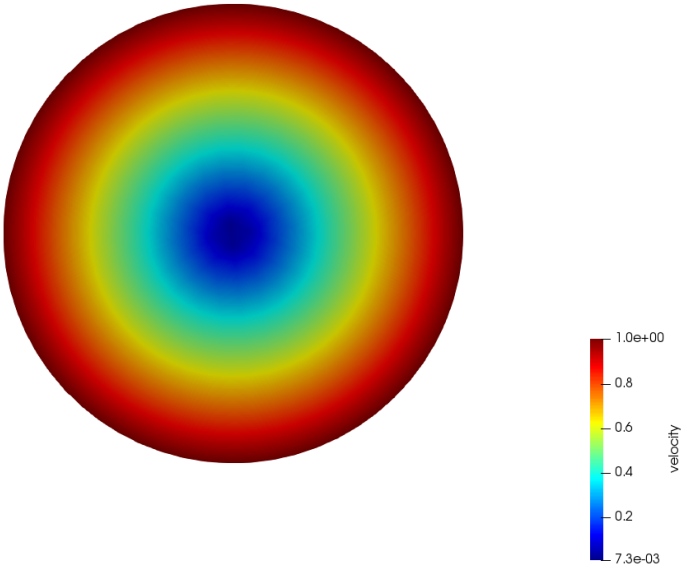


Figure 2.9: Initial velocity field obtained from the analytical calculations

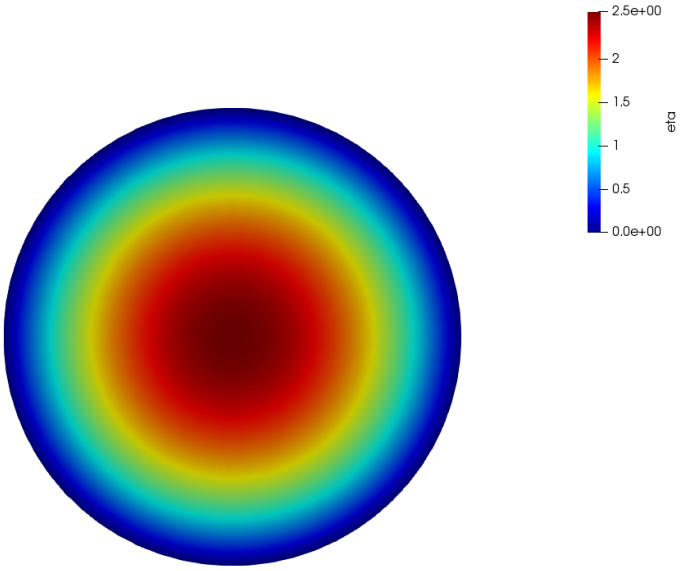
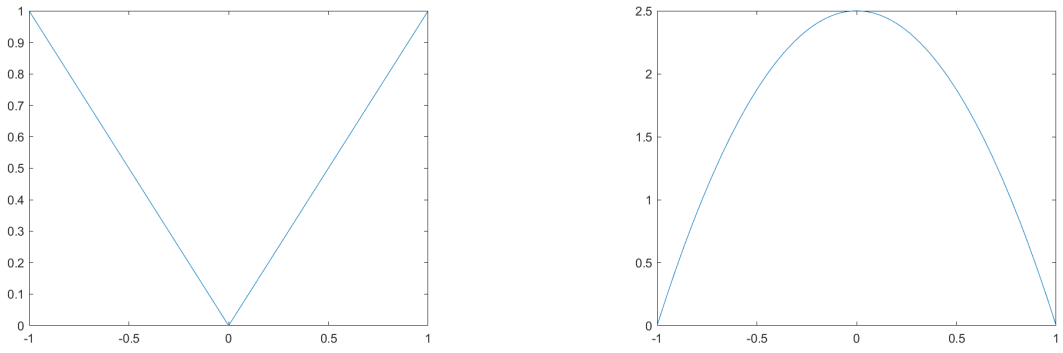


Figure 2.10: Initial velocity field obtained from the analytical calculations



(a) (b)

Figure 2.11: Plot of the fields in function of the radius r at $y = 0$; the x-axis correspond to the value of r divided by the radius of the setting R; the left plot shows the velocity field, the right plot shows the elevation field.

2.3 Results of the simulations

In order to have meaningful results, the simulations have been carried for a relatively long period of time, namely for around 1200 hours, or 50 days. Of course, as the goal was to analyze its degradation over time, no artificial source of energy was used during the simulations. The resulting energies were computed by performing the integral of the energy over all the elements [73, 74].
In total, eight different simulation cases have been kept, and their results will be shown in the following of this section.

1. Simulation (2D, flat, not fitted): using the 2D version of SLIM, with the mesh non-fitted mesh given on figure 2.11a, and with a flat bathymetry (of depth equal to 12 meters, i.e. the highest step);

2. Simulation (2D, flat, fitted): using the 2D version, with the contour-fitted mesh given on figure 2.11b, and with the flat bathymetry (it is expected to behave as the simulation (2D, flat, not fitted));

3. Simulation (2D, depth, not fitted): using the 2D version, with the non-fitted mesh, and with the stepped bathymetry shown on figure 2.6;

4. Simulation (2D, depth, fitted): using the 2D version, with the contour-fitted mesh, and with the stepped bathymetry;

5. Simulation (3D, flat, not fitted): using the 3D version of SLIM, with the mesh non-fitted mesh, and with the flat bathymetry (it is expected to behave as the simulations (2D, flat, not fitted) and (2D, flat, fitted));

6. Simulation (3D, flat, fitted): using the 3D version, with the contour-fitted mesh, and with the flat bathymetry (expected to

behave as the simulation (2D, flat, not fitted), (2D, flat, fitted) and (3D, flat, fitted));

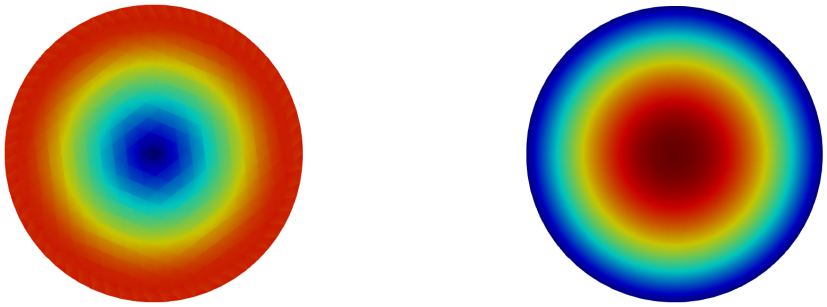
7. Simulation (3D, depth, not fitted): using the 3D version, with the non-fitted mesh, and with the stepped bathymetry (expected to behave as the simulation (2D, depth, not fitted));

8. Simulation (3D, depth, fitted): using the 3D version, with the contour-fitted mesh, and with the stepped bathymetry (expected to behave as the simulation (2D, depth, fitted)).

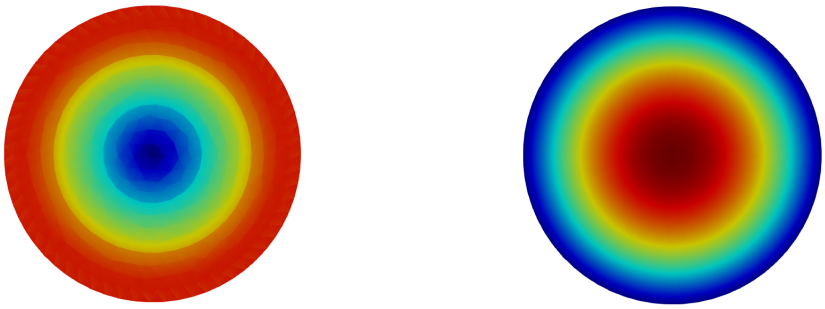
The first interesting results to compare are the outcomes of the simulations that are supposed to bring approximately similar results. The intermediate and final fields of the velocity and the elevation are given on figure 2.12. As expected, they behave almost in the exact same way. What is also important to note is that, for such a flat bathymetry, the fields do not tend to change a lot. This is actually a hypothesis that could have been made initially, as it is obviously a desirable property for a program based on finite elements. In terms of energy dissipation, on figure 2.13, the observations also corresponds to the initial expectations. Two details may seem quite surprising though:

- First, we can see that, even for such a simple configuration, the total loss of energy still amount to 1% of the total initial energy. This shows that it can be important to take care of the numerical energy dissipation for long term simulation, even for ones that look trivial. Nevertheless, we can observe that this is mainly due to kinetic energy losses, which tend to decrease slower over the time.

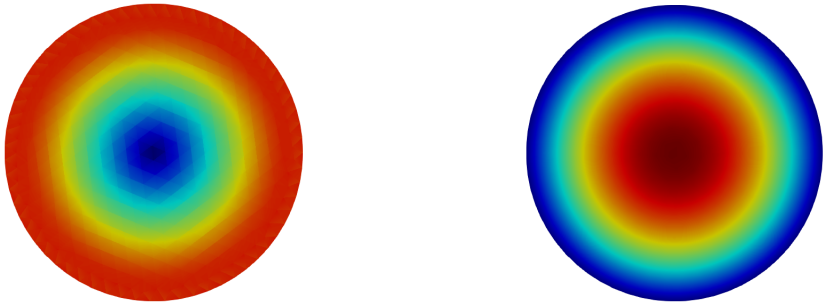
- Second, the energy loss is slightly lower in the 3D version than in the 2D one.



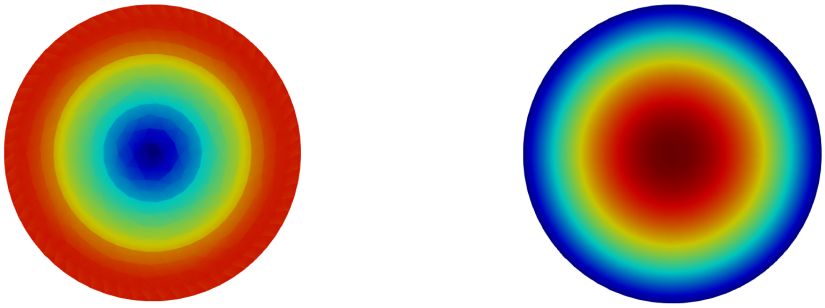
(a) Results of the simulation (2D, flat, not fitted)



(b) Results of the simulation (2D, flat, fitted)

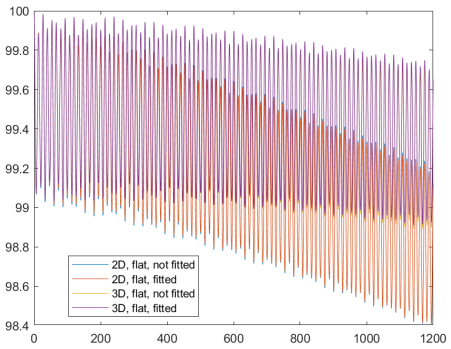


(c) Results of the simulation (3D, flat, not fitted)

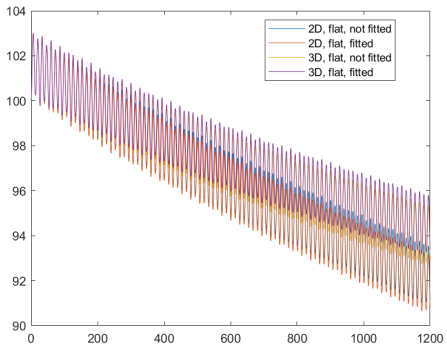


(d) Results of the simulation (3D, flat, fitted)

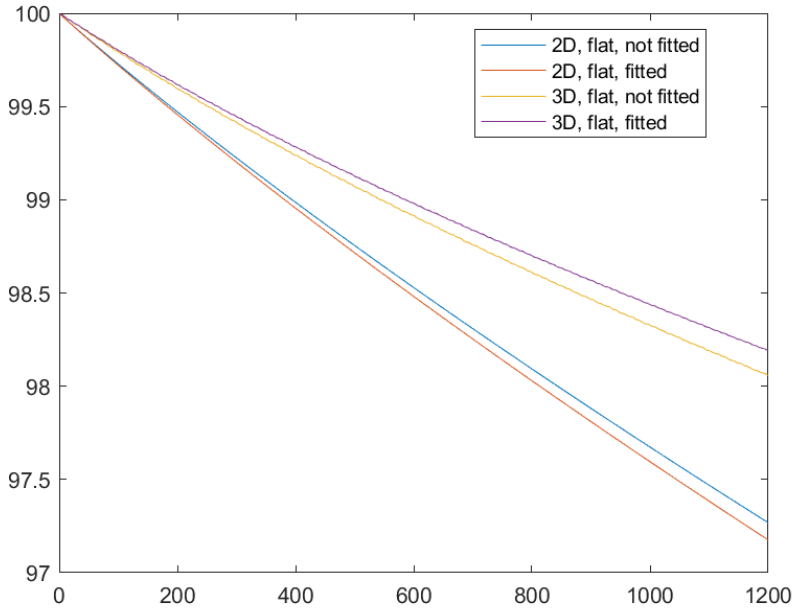
Figure 2.12: Fields obtained for the simulations with a flat bathymetry after 50 days of simulation; the left column shows the velocity fields, and the right column shows the elevation field; the respective colorbars of each field are the same as those used in figures 2.9 and 2.10



(a) Potential energies

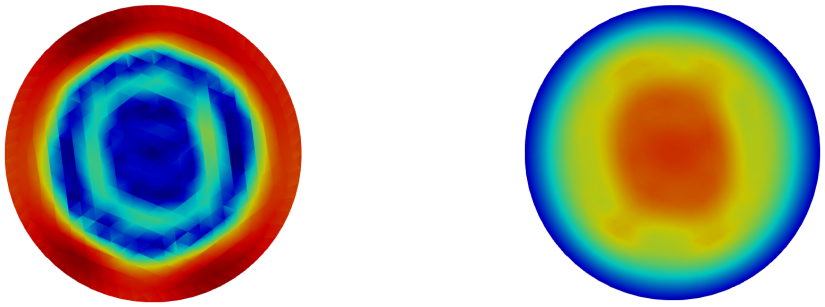


(b) Kinetic energies

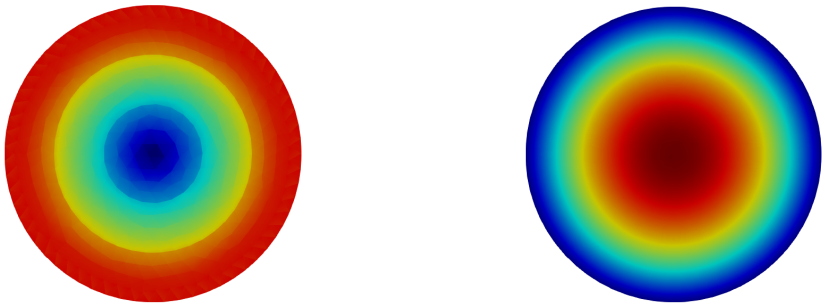


(c) Total energies

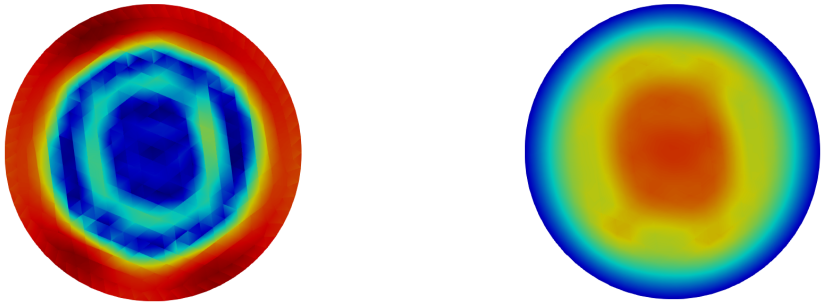
Figure 2.13: Plot of the evolution of the energies for the simulation with a flat bathymetry; the x-axis represents the time in hours, and the y-axis represents the percentage of the energies with respect to their initial values



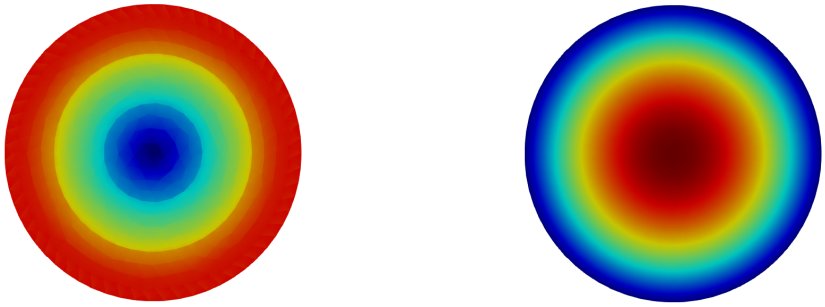
(a) Results of the simulation (2D, depth, not fitted)



(b) Results of the simulation (2D, depth, fitted)



(c) Results of the simulation (3D, depth, not fitted)

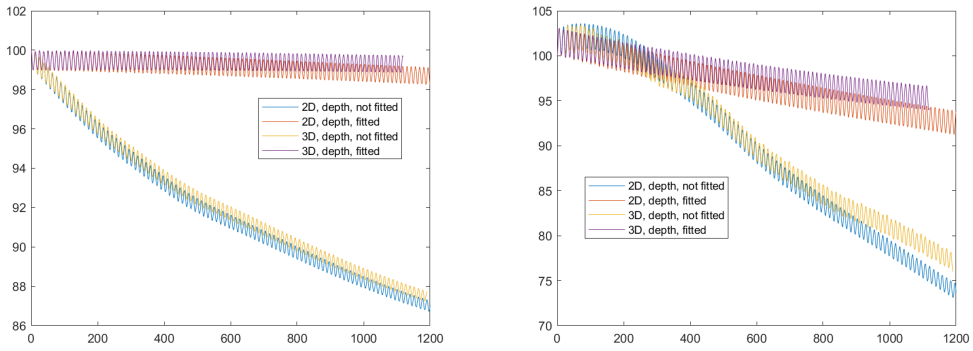


(d) Results of the simulation (3D, depth, fitted)

Figure 2.14: Fields obtained for the simulations with the stepped bathymetry after 50 days of simulation; the left column shows the velocity fields, and the right column shows the elevation field; the respective colorbars of each field are the same as those used in figures 2.9 and 2.10

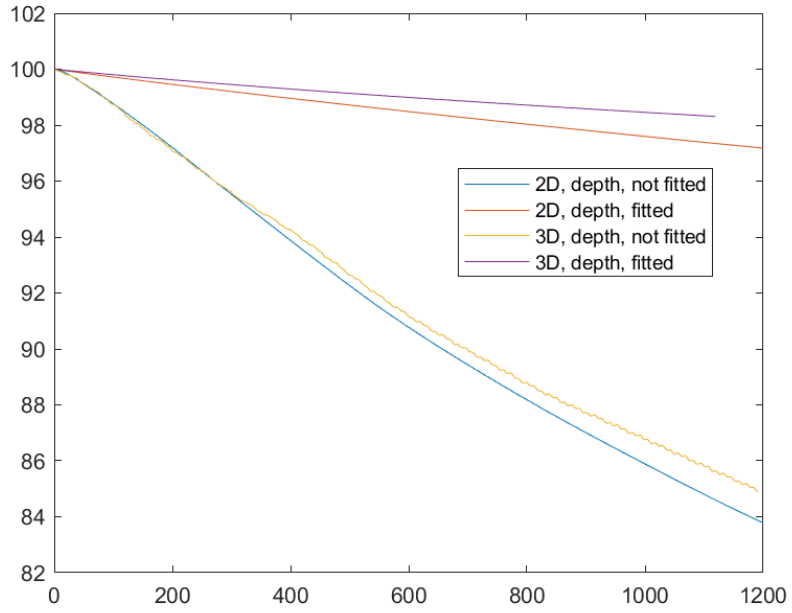
The results of the four other simulation are available on the figure 2.14. In contrary to the previous ones, large differences can be noted on these plots. Fortunately, these results are as one could have expected. Namely, we can see that the shape of the velocity and elevation fields obtained with the non-fitted mesh have completely changed during the process of the simulation, both in the 2D and the 3D versions. We can even see on these top views that the shape of their contour lines tend to align with the isobaths visible on figure 2.11a. In the same way, since these curves formed almost perfect circles on the mesh depicted in figure 2.11b, the contour lines of $\mathbf{u}$ and η obtained with the fitted mesh are almost the same as the one of their initial configuration.

Moreover, figure 2.15 shows that the amount of energy lost when using the unfitted mesh amounts to almost 20% of the initial total energy. This quantity is an order of magnitude higher than the loss obtained with the mesh which takes the contours into account.



(a) Potential energies

(b) Kinetic energies



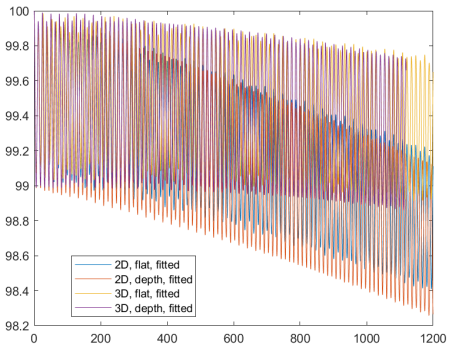
(c) Total energies

Figure 2.15: Plot of the evolution of the energies for the simulation with the bathymetry given in figure 2.6; the x-axis represents the time in hours, and the y-axis represents the percentage of the energies with respect to their initial values

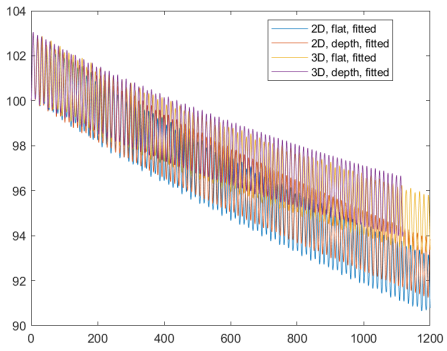
2.4 Discussion and conclusion on the usefulness of the idea

As a last important result to observe, we can compare the plots obtained using a flat bathymetry with the ones obtained with the adapted mesh. Surprisingly enough, these curves, put together on figure 2.16 happen to be almost exactly the same.

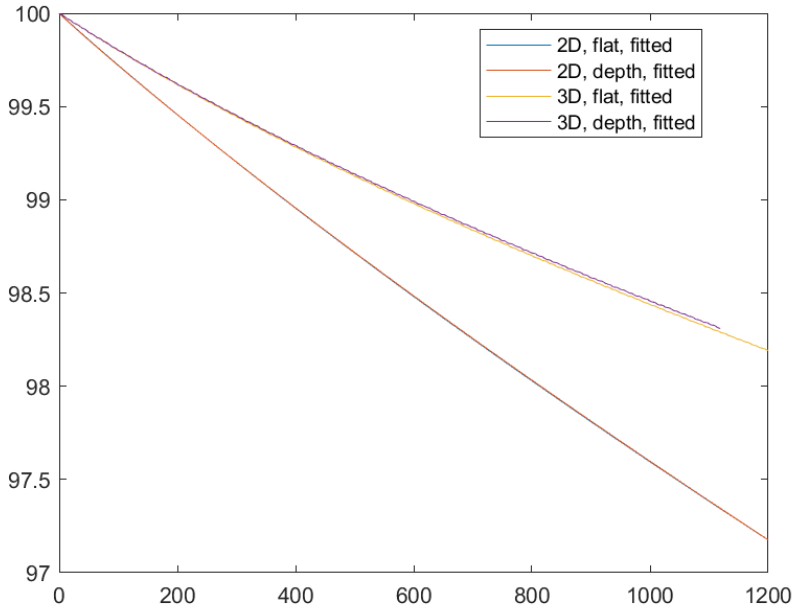
This means that, thanks to the consideration of the lines of isobathymetry during the meshing process, the solver is able to maintain the total energy as good as if there were no variation in depth, at least for such a simple test case where it is possible to obtain the analytical solution quite easily.
Nevertheless, these outcomes are assuredly promising and shows that the concept should definitely be tested on real-life cases. Even though reaching such improvements should not be expected on more complicated seabeds, the hope that a notably better accuracy could be achieved has been strong enough to validate the idea behind the contour-fitted mesh. Consequently, the next chapter will try to automate the process of finding isobaths and adapting existing meshes to them.



(a) Potential energies



(b) Kinetic energies



(c) Total energies

Figure 2.16: Plot of the evolution of the energies for the simulations (2D, flat, fitted), (2D, depth, fitted), (3D, flat, fitted) and (3D, depth, fitted); the x-axis represents the time in hours, and the y-axis represents the percentage of the energies with respect to their initial values

Chapter 3

The contour-fitted mesh adaptation

This chapter aims at describing the structures and the algorithms implemented to obtain the desired contour-fitted meshes. It details how in practice the line of isobathymetry are taken into account, and what modifications had to be made to the local transformations in order to do so. Two methods will be proposed with their advantages and drawbacks.
In the end, the results obtained on real bathymetries will be displayed and commented, emphasizing the difference between the two strategies considered as well as the flaws that should be rectified in the future. The chapter will conclude by pointing the most promising one on which further work still needs to be done.

3.1 Introduction

Based on the structures and the algorithms detailed in Chapter 1, and after having performed the validation in Chapter 2, the actual algorithm to fit the mesh on the isobaths can be implemented.

In its final form, four steps can be highlighted and will be detailed below. Also, it is necessary to mention that two versions to store and treat the contours will be explained. The first and more naive one treat the contours as an ensemble of points, and does not ensure that the contours keep forming closed curves. The second, on the other hand, treats the contours as segments and curves. In consequence, it has to ensure the continuity of the curves all along the following mesh updates. The differences between both will be pointed for each of the four steps.
Detailing those, the first one concerns the finding of the lines, and their inclusion in the mesh. It mainly uses a double for-loop, traversing each edge for each depth where a contour has to be placed. The biggest challenge was to keep the points in the order in which they appear on the contours. This feature was optional for the first version, but was of a uttermost importance for the second method, where contours should be fully traceable. The second step required to filter the isobaths at complicated zones, where only element of bad qualities would be created. As this will be explained, this was supposedly less of an issue for the second method. Nevertheless, it was noted that such zones were inevitable. Therefore, another more sophisticated filtering strategy ended to be implemented in order to ensure the overall good quality of the mesh. The third important step is about the handling of the computed lines during the update algorithm detailed in section 1.2.9. With the first method, as the curves are only considered through points, most of the transformations occurs similarly. The biggest distinctions implies the vertices created by the split of an edge which should be considered as part of the contour, and the vertex relocation, which is simply skipped. Indeed, since the lines are only described by points, and since those are not kept in order, it is not possible to ensure that the points stays on the curve when freely moving it in its cavity. With the second method, more differences are present, for mostly all of the transformations. The most notable changes were performed for the

split of an edge of the contour. A fully new algorithm have been implemented to this aim, using a method to recover arbitrary edges, which was then also added to the filtering step. Finally, an easier but essential last step implies the attribution of their corresponding plateau to each face. As usual, it resulted being harder with one method, namely the first, and trivial for the second. This is due to the fact that, again, the contours were not considered as continuous curve, and they had to be completed before performing this last step. This issue was the main reason for which the second method was developed.

3.2 Description of the algorithm to fit the mesh to the isobaths

This section will cover the four steps that can be highlighted in order to take the contours into account. Besides the first one, they all differ more or less for the two different method. Again, the biggest distinction between those methods is that the one treats the contours as set of points while the second one considers segments and continuous curves.

3.2.1 Computation of the lines and their inclusion in the mesh

In contrary to the following steps, no differences arose here between the two methods to compute the contours. Figure 3.1 shows a detailed flow chart in the finding of the points belonging to the isobaths. The last box on the path, representing the filtering, is actually the second of the four steps, which thus takes place right after this first one. The most important part is the traversal of each edges in order to detect the points where a contour is crossed. When an intersection is found, it is useful to know the relative position of this point with respect

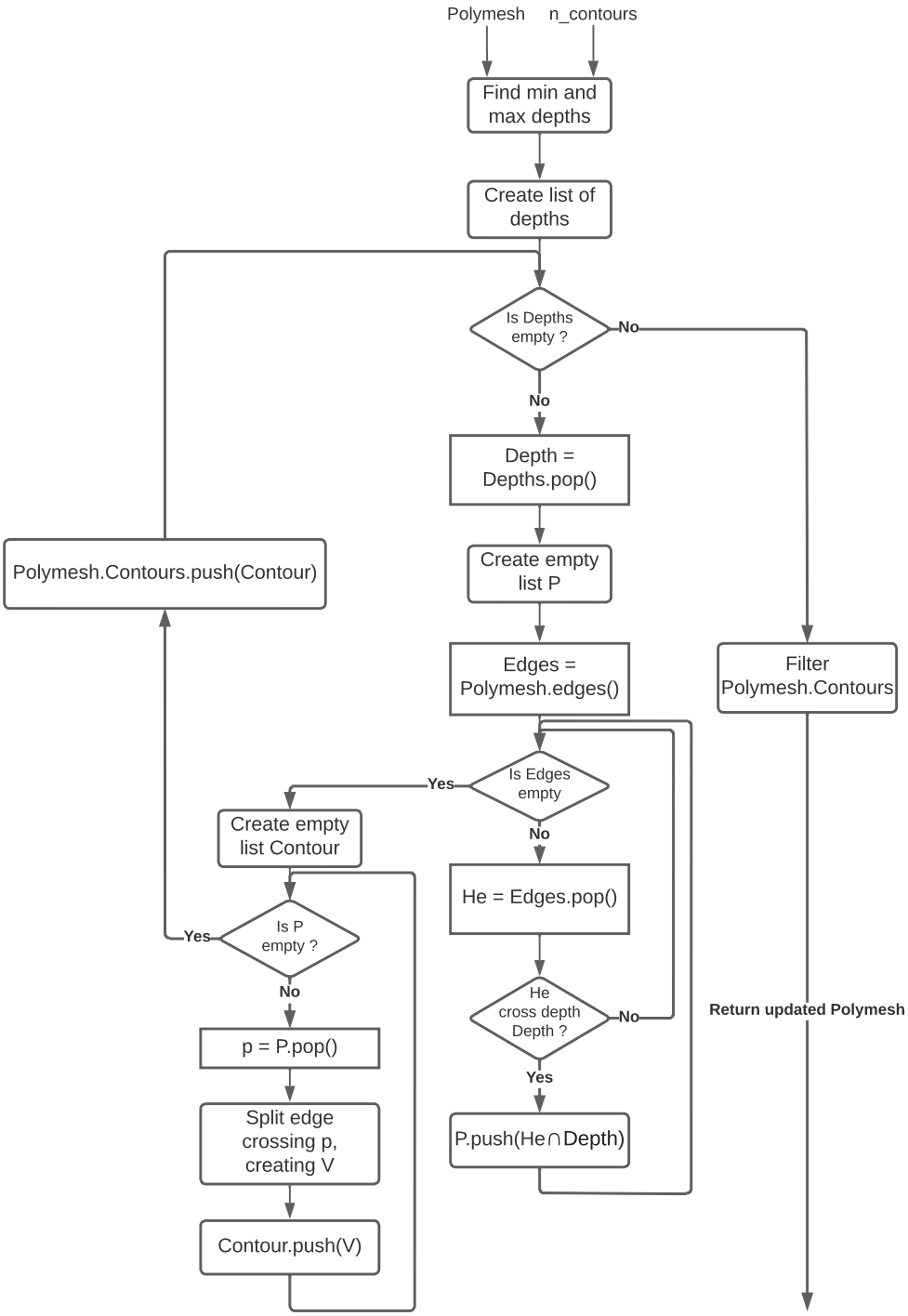


Figure 3.1: Flow-chart detailed the first steps of the treatment of the contours

to the other ones in the current line. It is important to precise that the depth is considered to vary linearly over the edges. Their slope thus strictly increasing, decreasing, or constant with respect to the z coordinate. This has the interesting implication that, for a given triangular element, the number of intersections will be at maximum equal to two: it would be zero when there is no intersection, one if it crosses a vertex, and two if an edge is intersected. Here, we will not detail the special case of an edge at constant depth crossing a contour, but such cases are trivially treated in the code. Having this information about the number of intersections, it is possible to compute, for each intersected edge, which edges of its neighbouring faces will also be crossed by a contour line.
For this purpose, three arrays were created:

1. An array containing, in the order in which they have been crossed, the data of the edge which will be split.

2. An array containing, in the order in which they have been crossed, the data values of the neighbouring crossed edges of the current.

3. An array with a number of rows equal to the number of edges, and with three columns. It is initially filled with negative values. Those will be changed each time an edge is crossed. For example, let the edge 10 being the second edge crossed, and its neighbour on the right being edge 5 and on the left 15. As it is the second, its data will be present at index 1 in the first array, and its neighbour's data values will be at index 1 in the second array. This means that, in this third array of three columns, the value in the middle at index 10 will be set to 1, the leftmost value at index 5, the right neighbour, will also be set to 1, and similarly for the rightmost column of the left neighbour.

The algorithm now to order the contours is now the following:

1. We traverse the first array to obtain the data d an edge crossed by the contour. This has to be done twice : first to find the curves which are not close, and then to find the closed ones.

2. As we first aim for the open curves, a check is done in the third array to check that the leftmost column at the index d has a negative value, meaning that the edge is the first of the open curve.

3. Since the position of the edge in the first array is known, as it is given by the middle value in the third array, we can find its neighbours by looking in the second array at the same position.

4. The current edge is split. Then, this current edge is updated by setting it to the right neighbour. Indeed, the third column of the third array gives its index in the first array, as well as the one of its neighbours in the second array. This allows to continue until the end of the open curve, i.e. when the next index, given by the third column is negative.

5. The process is repeated until all the open curves are found. The same operations are then performed for the closed lines, with the difference that the curve is considered as ended when the next index equals to first one that was found for the current curve.

Although not fully expanded in the flow-chart, this may imply the creation of several contours for one depth, each contour representing a full closed or open curve. As it may be guessed from figure 3.1, the contours are represented by lists of ordered vertices, themselves being stored in the `Polymesh` structure in a list a contours. Once this step has been performed, we can continue to the filtering of the complicated zones.

3.2.2 Filtering of the problematic zones/areas that would contain badly shaped elements

The concerned zones here are simply those that would inevitably lead to badly shaped elements if they were not treated previously. Here, the distinction between the two different methods has to be made.
As mentioned previously, the first and more naive one simply considered the contours as a set of points. Nevertheless, they are still initially separated in different contours, thanks to the method described in the previous section. The filtering method was in consequence obvious: it has to ensure that there is enough space between the points. The update algorithm will then be able to fit elements of good quality which respects the size map the points left. The methodology followed here can be seen on figure 3.2. Even though not fully explained in the flowchart, each point of each contour is compared to the other points in the other contours. When a point is considered as close, the other point is remove. The definition of close here needs to be fixed. With the size map, it is possible to obtain for each vertices the element size required at its position. Two points are then simply considered to be close when there distance is lower than a factor multiplied by the average of the prescribed sizes.
Here, it is important to mention a structure which has been used to make this search more efficient. Indeed, the search would have required, for each point of each contour, to perform the check for all the subsequent points. To make this process faster, the vertices are first mapped into the cells of a virtual grid dividing the whole space, as visible on figure 3.3. The size of these cells have been set equal to the maximum length prescribed by the size map. By doing so, for a given point, it was only required to look in its current cell, and in the 8 neighbouring ones, to find all the other points which would have to be filtered.

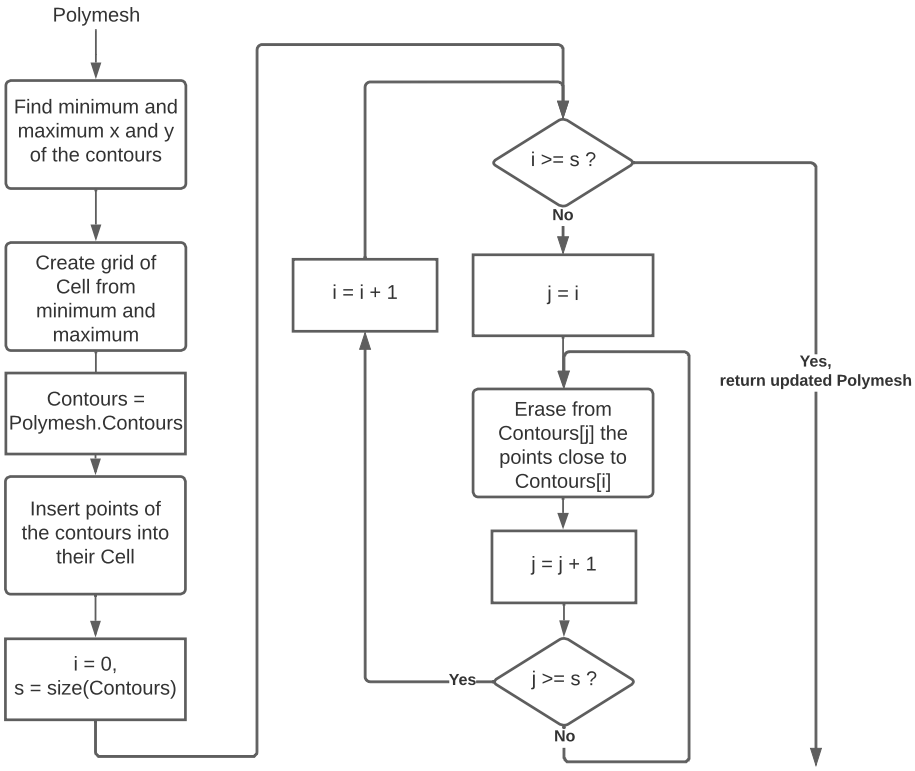
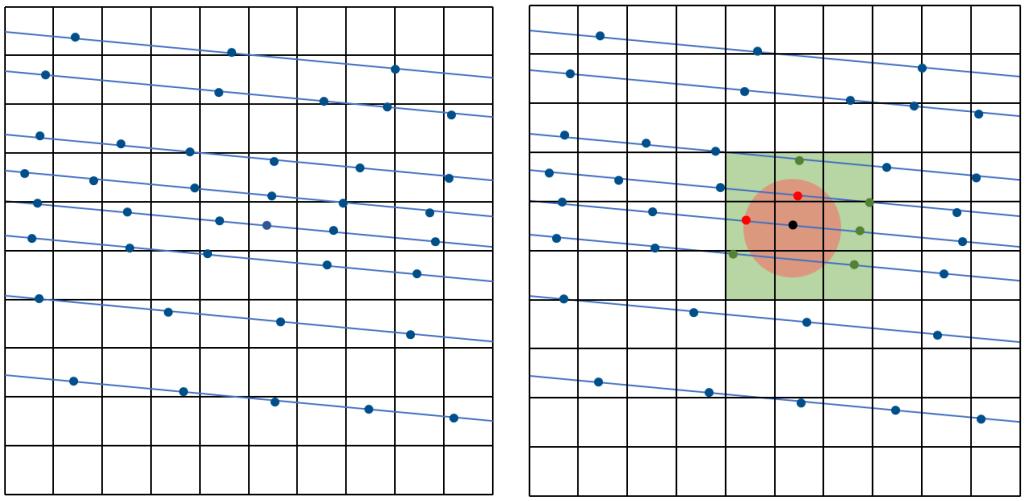


Figure 3.2: Flow-chart of the algorithm used to filter the contours with the pointwise method



(a) (b)

Figure 3.3: Representation of the cells used for the first method. The left image shows parallel, oblique contour lines, where the blue dots are the points stored in memory for each of them. The right image shows the filtering process from the black points. Only the neighbouring cells of the black point are scanned, so that only the green and red points are checked. The red circles represent the distance of influence depending of the size map, which is considered constant in this drawing. The red points are then simply removed from the contours.

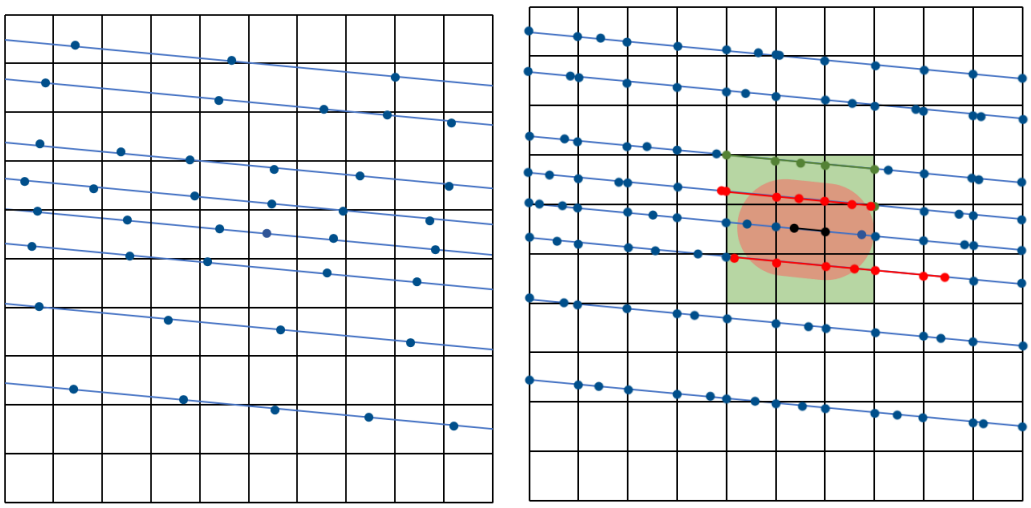
The second method to store and treat the contours considers them as segments, forming continuous curves in the mesh. It was thus important to verify that the continuity property was valid initially when the vertices were created. This lead to the necessity of marking also the edges as being part of a contour, in contrary with the first method where only the vertices were marked. Although this was not expected initially, this method required much more considerations than the previous one. Without entering into all of the details, two additions needs to be explained. First, cells are still used, as for the previous method, but they now contains segments. Those segments are represented by the coordinates of their two endpoints, and not simply a set of disconnected points, as shown on figure 3.4. More precisely, a segment from a contour is split into different pieces if it crosses different cells in the grid. Additional points are then used to represent on segment of the contour, but they are still considered to be part of the original contour segment. This means that, when performing the comparison, if one piece of a segment is too close from a piece of segment from another contour, the whole segment is erased.

With this new storing strategy, the comparison during the filtering step has drastically changed. A schematic representation of it is detailed at figure 3.5. This algorithm works as follow:

1. If, by using the new cells, the minimal distance between a less important segment h_{weak} and a more important segment h_{strong} is considered as too low, i.e. $d < D$ on the figure, a filtering has to be performed.

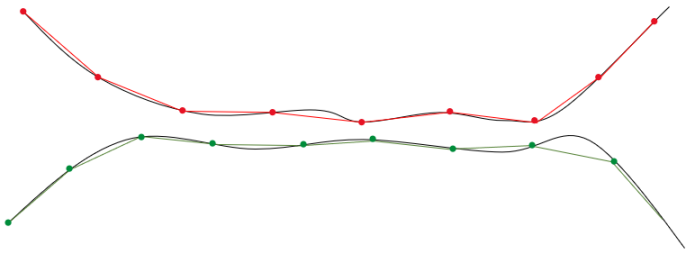
2. In order to do so, we start from the current close segment on the contour having a weaker importance.

3. The segment is removed, and we set h_{weak} to be one of its neighbours.

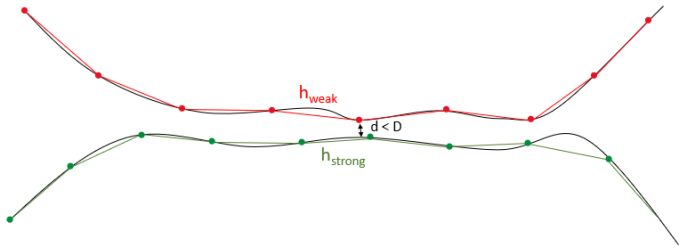


(a) (b)

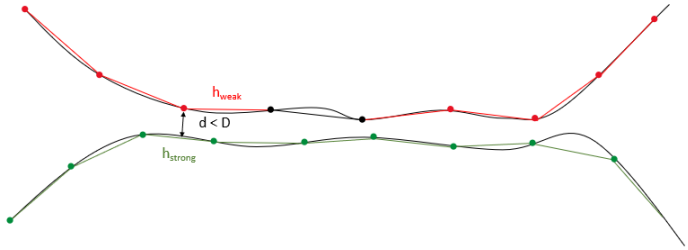
Figure 3.4: Representation of the cells used for the second method. The left image shows parallel, oblique contour lines, where the blue dots are the initial points stored in memory for each of them. The right image shows the filtering process from the black piece of segment, as well as the additional points appearing at the intersections between the segments and the cells. Even if some pieces of segments are not crossed, they are erased because another piece of this segment has been considered as too close from the black piece of segment. The red zone represent again the distance of influence with the same size map as in figure 3.3, but now for the whole piece of segment in black.



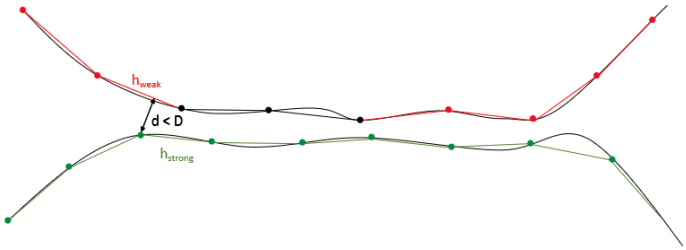
(a) Initial contour lines considered. Contours have different importance. Here, the green contour has a stronger importance than the red one.



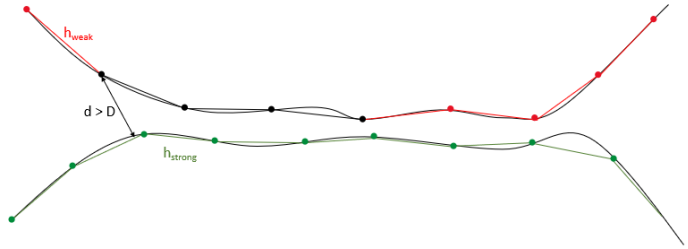
(b) When performing the comparison step from the edge h_{weak}, the edge h_{strong} is considered as too close, namely the distance between the segments d is smaller than the one based on the size map D.



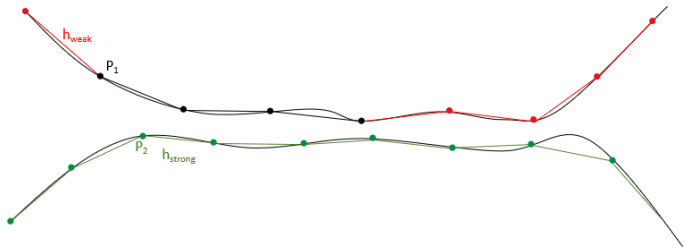
(c) The initial edge h_{weak} is removed from the red contour, and the comparison step is performed on its left neighbour. Again, a close stronger edge is found.



(d) The process continues, erasing the previous h_{weak} and traversing the green contour in the same direction.

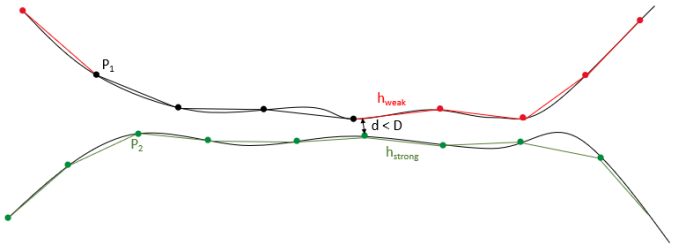


(e) Until an edge h_{weak} is found such that the minimal distance from a stronger segment d gets lower than D.

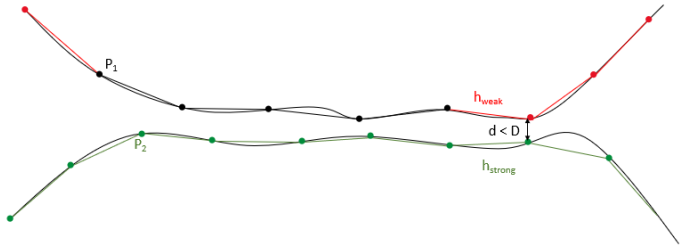


(f) The two points P_1 and P_2 are stored.

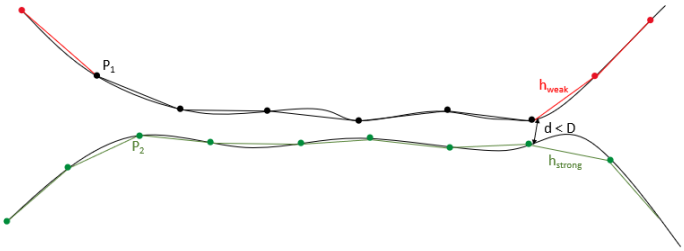
Figure 3.5: First part of the algorithm to compare the contours with the second method



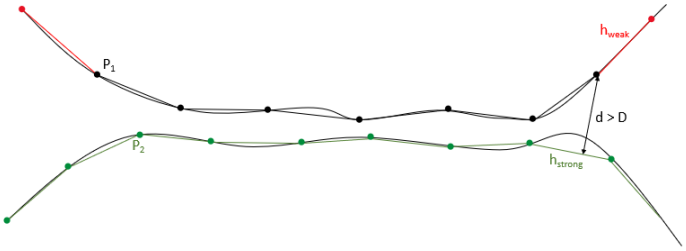
(g) Then, a second traversal is launched by starting from the right side of the initial h_{weak}.



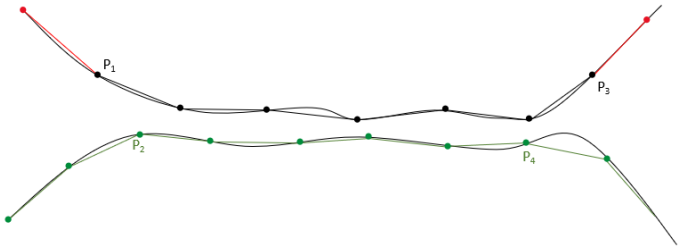
(h) Again, the previous h_{weak} is removed, the next one being found by traversing the contour in the right direction.



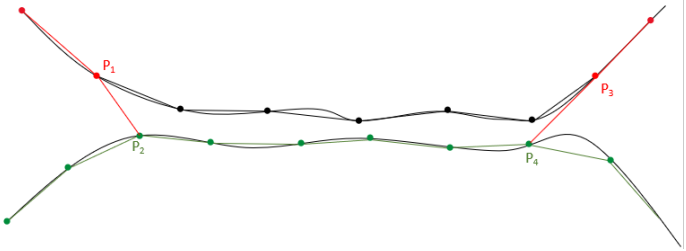
(i) Of course, P_1 and P_2 are still known and fixed during this process. Also, each value of d and D might be different, following the found distance and the size map defined.



(j) The process still goes on, until another edge h_{weak} is found to be sufficiently far from every stronger segments.



(k) The points P_3 and P_4 are then found and stored.



(l) Finally, an optional edge recovery may occur between the points P_1 and P_2, and between the points P_3 and P_4. This first recovery step may also have been done directly after P_1 and P_2 were found.

Figure 3.5: Final steps of the algorithm to compare the contours with the second method

4. Then, from this new edge, we find the closest segment on a stronger contour.

5. If we find another segment which is too close, we go back to step 3. Otherwise, we have found two distant enough segments. We then conserve two points, one on the less important contour, and one on the most important one.

6. Those two points, called P_1 and P_2 are stored, and the procedure is performed in the other direction until we find the points P_3 and P_4.

7. In the end, an edge recovery may be performed between points P_1 and P_2, and another between P_3 and P_4, in order to maintain the continuity of the weaker contour.

All these steps are also detailed in the flow-chart on figure 3.6, which gives more insight on the actual implementation of the algorithm. At this point, two features have to be detailed in order to fully understand this flow-chart: the arbitrary edge recovery algorithm and the definition of the concept of importance and of order in which each contour is traversed.
The arbitrary edge recovery is a well known subject which has been described in several papers [75, 76]. Its main goal is to start from two arbitrary points, which may or may not be part of the mesh, and to connect them through an edge. To do so, the mesh is modified and new elements get formed. While those operations are performed, we have to take care that the mesh still remains manifold [33] and that no triangle gets inverted in the process, as detailed when discussing the swap and the collapse operations in Chapter 1. The chosen algorithm here is the recursive swapping algorithm [76], and we will only consider the case where the two endpoints are part of the mesh. Basically, it requires performing the following steps, also visible at figure 3.9:

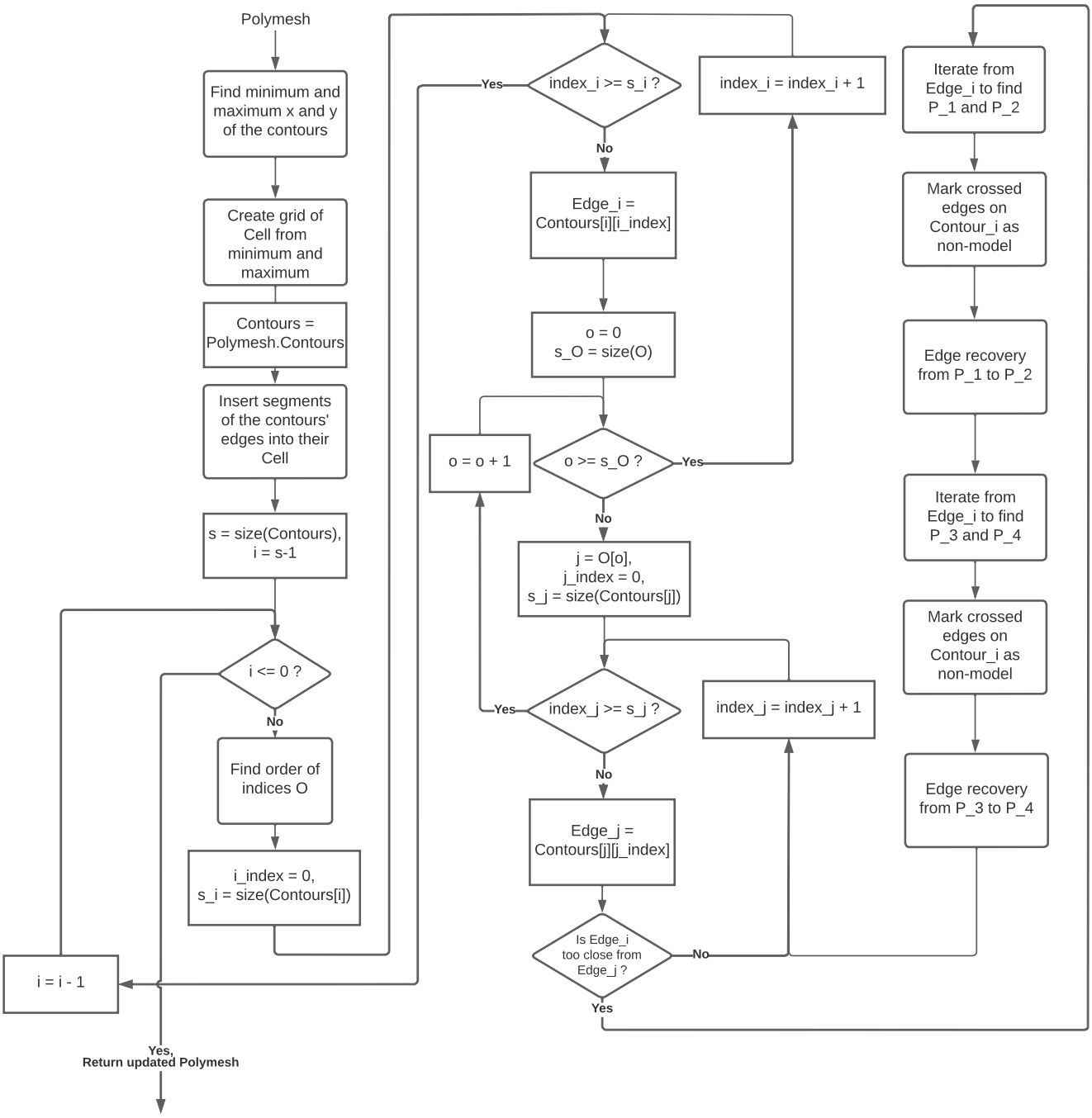


Figure 3.6: Flow-chart of the algorithm used to filter the contours with the segmentwise method

1. We define a starting vertex, like v_1 or v_2, and a target vertex, v.

2. A virtual line is traced between the two vertices. From the starting vertex, we scan its neighbouring faces to find one crossing the virtual line linking the two vertices.

3. Take the edge of this face opposed to the starting vertex, called h_1 on the figure, and insert it into a queue of edges.

4. On one side of this edge, there is the face containing the starting vertex, and on the other side, we know that either one of vertex of the face is the target vertex, or that another of its edge is crossed by the virtual line. If it exists, this edge h_2 can be found, and added to the queue of edges.

5. From this next edge, step 4 can be repeated until the target is reached.

6. The final step requires to traverse the queue of edges, and to pop each edge until the queue gets empty. For a given edge, we check if it is possible to swap it keeping the mesh manifold. If not, it is put back at the end of the queue. Otherwise, it is swapped, and only put back into the queue if it still crosses the virtual linking line.

7. The process at step 6 stops when the queue is empty, meaning that the last swapped edge is the one linking the starting and the target vertex that we wanted to recover.

This edge recovery algorithm will also be used in the following subsection when detailing the split of a contour edge with the second method. The second feature concerns the concept of importance of the contours and the order in which they contours are treated. The figure 3.6 shows two traversals of the contours: one from the last included contour to

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	9	5	10	3	11	6	12	2	13	7	14	4	15	8	16	1

Table 3.1: Example of the importance given to each contour in terms of their index. The first line represents the indices, ordered by increasing depth. The second line is the importance attributed to each of them, where the most important contours are the one with the lowest value in this row.

the first one, and a second traversing the contours in an order that needs to be defined for each line. Even though it was only briefly mentioned earlier, the isobaths have an intrinsic order of importance which is fixed and defines which contour should be filtered in algorithm shown at figure 3.5. This order has been chosen to obtain the most space possible between two adjacent contours, when ordered in terms of their importance. As an example, table 3.1 shows the resulting order for $n_{contours} = 17$. The first row indicate their order in terms of their depth, while the second line is their order of importance, the lowest one being the strongest. We can see that contours at indices 0, 16 and 8 are the most important, then we have the indices 4 and 12 and so forth. This order is quite simple to find:

- First, we take the highest value i such that i is a power of 2 lower to the number of contour, so here $i = 16$.
- Then, we set the first contour to have the most importance, and the one at index i to be the second.
- After that, i is recursively divided by 2 after all of its multiples have been given their corresponding importance, known by maintaining an index which grows when encountering a contour which has not been given its importance yet.

Although not specified, this order is actually the one followed when initially inserting the contours in the structure, in the algorithm shown on figure 3.1.
From this order of importance, the order of traversal of the second loop can be found. Simply, it imply doing the inverse transformation of the one explained, in order to cross to contour from the closest to the furthest one in terms of depth. For example, let us consider that we want to find the order to traverse the contours from the one at index 9. We can see that it has an importance of 13. We will start the traversal with the contours of index 8 and 10, which have been included respectively in the third and seventh position. We would then continue by traversing the contours 7, 6, 12, 5, 4, 14, 3, 2, 16, 1 and 0, the others being skipped as having a smaller importance. Although this was not required, it was preferred to choose this order to avoid as much as possible to cross a contour edge when performing the edge recovery. It is of course necessary to mention that since the bathymetry is not forced to be strictly increasing or decreasing from a given depth, it is not because two contours are close in depth that they are in terms of their actual distance.

3.2.3 Handling of the contours in the algorithm used to improve the mesh qualities

In this section, similarly to the previous one, the distinction between the two methods is really important. Indeed, the second one inferred a different consideration of the edges and lead to more complicated functions in order to keep the contours as continuous as possible. The common initial step was to consider the vertices which are part of the computed isobaths as another kind of vertex. For this purpose, they have been given a field called *is_model*, which has a value of zero for a usual vertex, a value of 1 for a vertex part of the border of the mesh,

and a value of two for a vertex part of a contour.
With the first method, almost all the operations in the update algorithm were performed in the same way. The swap between two points of the same contour was allowed, as their continuity did not have to be imposed. This actually relied on the fact that the points had been already filtered to be at a distance slightly lower than the one prescribed by the size map at their positions. One could thus have expected that an edge would have to link two consecutive points of a contour in order to maximize the overall quality of the elements while respecting the size map as much as possible. The problem arising from this assumption is that we rely too much on the update algorithm. This lead to the definition of another function to complete the contour in order to obtain continuous curves, required for the last step. This small algorithm will be detailed in the dedicated section. Another difference with respect to the initial transformations concerns the split operation. Indeed, it is desirable that a point created between two points of a contour results in another points part of that contour. When it was detected that such a split occurred, the point was simply put at the end of the contour structure, and it was also given a model value of 2. In consequence, the end of the contour structure contained all the new contour vertex created after the previous steps. Concerning the collapse operation, they are simply not performed when both vertices are models, and thus applied only when the vertex to collapse was not a model, or when none of the two was from a contour or a border. Finally, as mentioned previously, a last operation to change was the relocation of a vertex. As the order is not preserved with the pointwise method, it is not possible to know which points are supposed to be the previous or the next one in the current contour. Consequently, this operation was simply skipped when encountering a contour vertex.
The second method lead to more modifications. First, in order to maintain the continuity property of the curves, not only the vertices but also the edges had to be marked as model segments of type 2.

Their consideration in the swap operation is simple: they are simply never swapped during the whole simulation. Concerning the collapse operation, most of the considerations summarize to the ones for the pointwise method, up to some minor differences. For example, when an edge is collapse into another, a check has to be done so that when two apart edges are combined, the fact that one half-edge is a model edge is also transmitted to its opposite.
Two transformations remain to be detailed: the edge split and the vertex relocation. Since, with this second strategy, the order of the vertices in the structure had to be maintained, it was possible to know and to get for each contour node the previous and the next vertex on the current contour. This also means that, when storing the contour, it was possible to consider points between the vertices which were not part of the mesh, but only part of the contours. Indeed, with unordered set of coordinates, those points would have been useless as there would be no clue about the line which they would belong. Since it was possible with this method, the idea was then to keep, in addition to the initial points and segments of a contour, via-points in between, which may or may not be part of the mesh. By doing so, the initial form of the contour can be kept all along the simulation, even if those are not necessarily visible on the mesh. Figure 3.7 shows in green the initial points of a contour, in black the points created which are not part of the mesh but well of the contour, and in red the points created which are part of both. It is useful to mention that the segments separating green points are straight lines, due to the consideration that the depth varies linearly on the edges. Now if the algorithm would require to split a half-edge, care should be taken such that its mid-points remains situated on the contour. The algorithm used to perform this operation is detailed in the flow-chart of figure 3.8, along with a schematic representation in figure 3.9, which also details the arbitrary edge recovery algorithm. First, the mid-point has to be found, but this step not as trivial as it might seems.

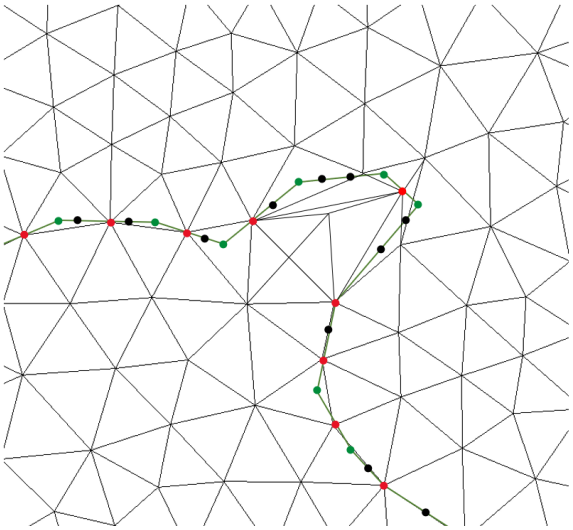


Figure 3.7: Types of points stored for a contour with the second method

Indeed, two pieces of data have to be taken into account: the distance between the two points following the shape of the contour, and the side of the contour on which this mid-point should be placed. This second one is only valid for closed curves, as there are two points which are equally distant from the two end-vertices of the edge to split. The first one is handled by conserving, for each curve, a parametric value ranging from 0.0 to 1.0. The first point of the curve is given a value of 0.0, and the last one a value of 1.0, while all the intermediary points get their value by computing their distance on the curve from the first point, and dividing it by the total length of the curve. The required parametric value of the node to find is thus equal to the mean of the parametric values two end-vertices, or this average plus or minus 0.5 if we have to place the mid-point on the other part of the contour.

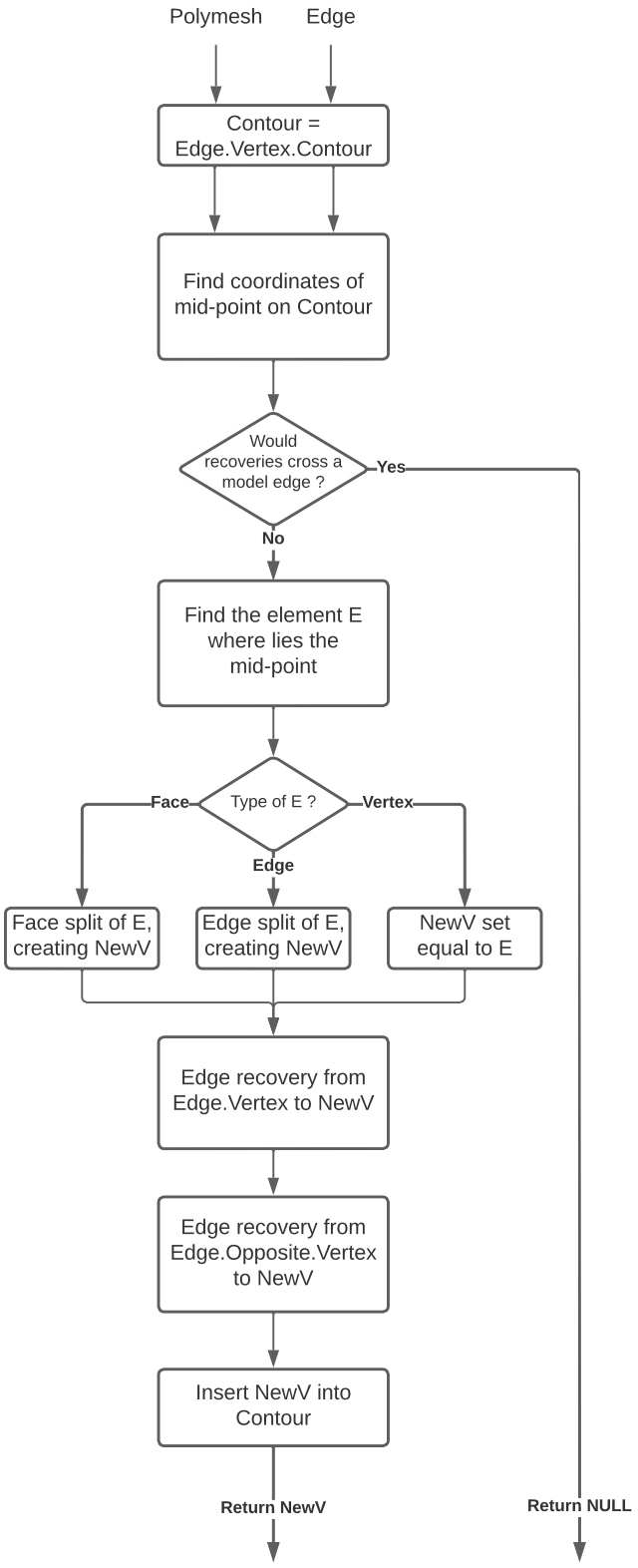
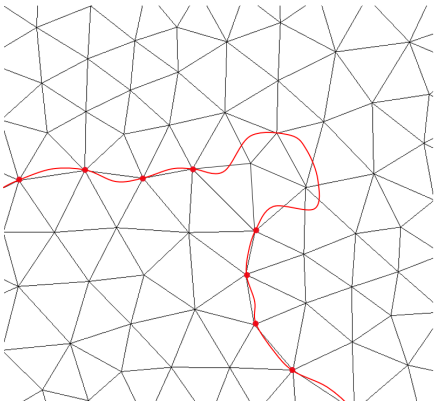
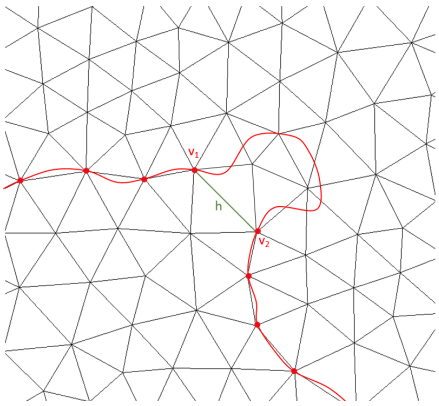


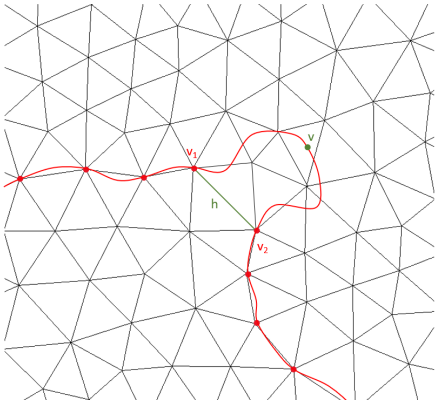
Figure 3.8: Flow-chart of the function created to split a contour edge



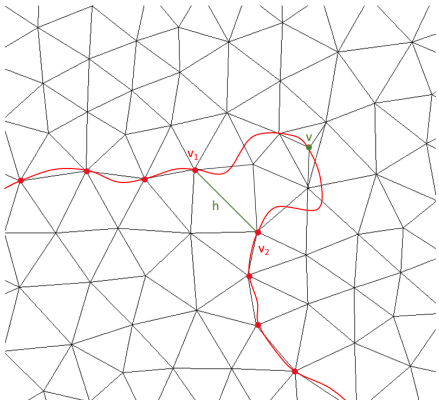
(a) Initial shape of the contour stored. Only the edges between two successive red dots represent the contour in the mesh.



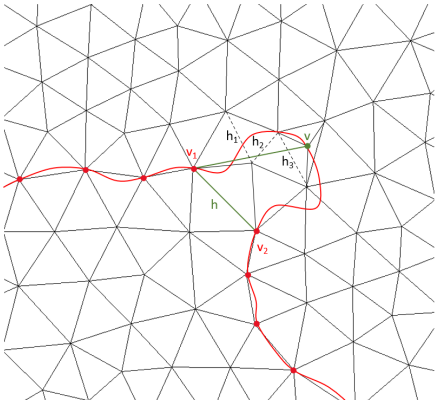
(b) We can imagine that edge h, between the points v_1 and v_2 should be split.



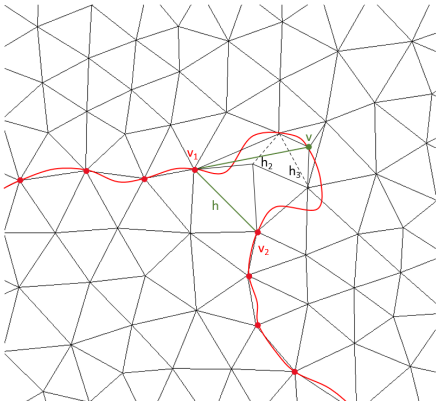
(c) Point v is the point situated on the contour and between v_1 and v_2



(d) v is found to be inside a face. Consequently, a face split occurs to include into the mesh.

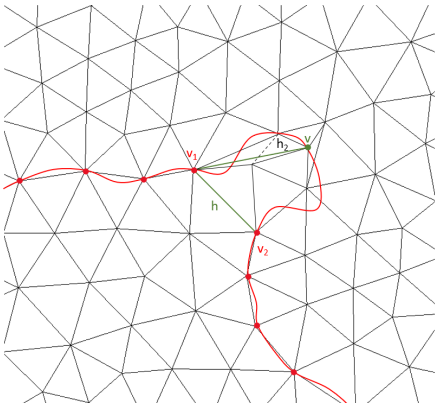


(e) A virtual line is traced linking v_1 and v. It crosses three edges, identified as h_1, h_2 and h_3.

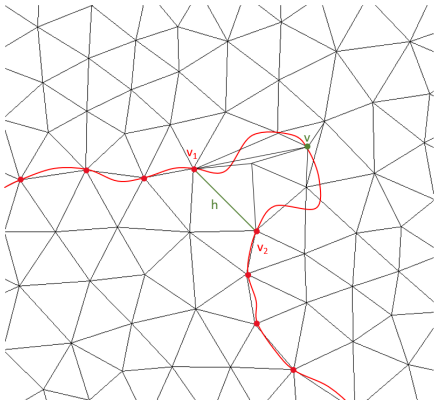


(f) As the validation check mentioned in section 1.2.2 is verified for h_1, it is simply swapped.

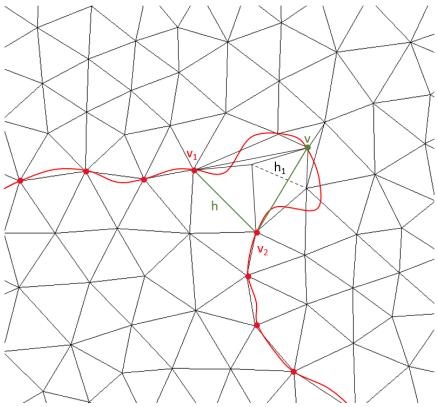
Figure 3.9: First part of the schematic representation algorithm to split an edge of a contour



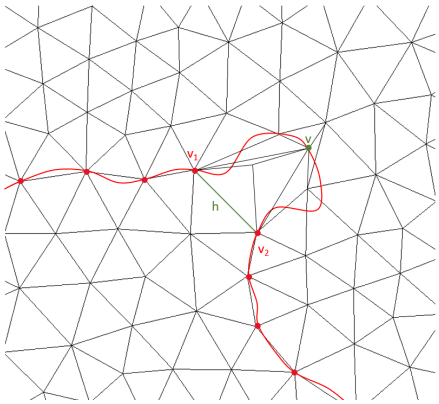
(g) h_2 does not verify the condition, so it is put at the end of the queue. h_3, in contrary, satisfy the condition, and is thus swapped.



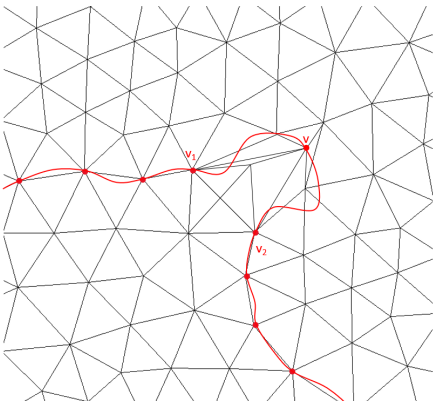
(h) Then, h_2 can be swapped, in order to recover the desired edge.



(i) The same process is repeated by starting from v_2. Only one edge h_1 is found.



(j) h_1 is then simply swapped, revering the desired edge.



(k) Finally, the initial edge h is split as a usual edge, while the new vertex v is inserted in consistently in the contour.

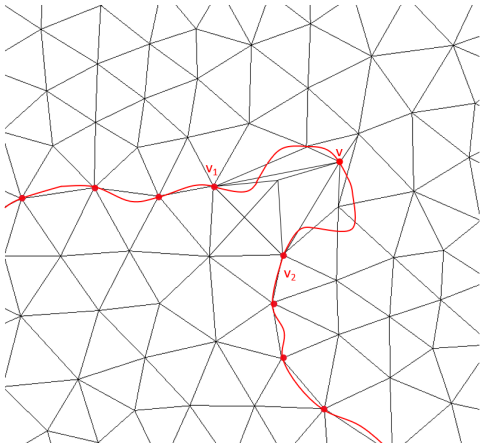
Figure 3.9: Continuation of the schematic representation algorithm to split an edge of a contour

The way to handle the second issue and to know which part of the contour should be considered will not be detailed, but this mainly requires to keep, for each vertex in a contour and present in the mesh, a pointer to the next and the previous vertex in the same contour and also present the mesh. When this is done, we obtain the coordinate of the mid-point. The next step is to check if the two recoveries will be feasible, namely if we will be able to connect both end-vertices of the edge to split to the new position found, without crossing a model edge. This is performed by starting the edge recovery algorithm detailed in the previous section for both edges to be recovered. After gathering for each side the edges crossed, a check is performed to ensure that none is model. For now, the function simply return `NULL` if this happens. Otherwise, the actual operations are performed. Since we followed the first steps of the edge recovery algorithm, we can know on which element the vertex should be placed. This can either be inside a face, on an edge, or simply on a vertex. Following the type of the element found, a local transformation is performed to obtain the vertex that will be included in the contour, as detailed in the flow-chart on figure 3.8. In the end, the two edge recoveries are performed, and the new vertex is inserted in its correct position in the contour.
Finally, a last operation can be modified in order to be consistently used with the second method to compute the contours, namely the relocation of a contour vertex. This function behaves in a similar way to the original. As a reminder, it involves to virtually removing the vertex and considering the formed cavity as empty, then tracing a line linking the initial position of the vertex and the center of mass of the surrounding vertices, and performing a bisection method which tries to maximize the lowest quality of the neighbouring elements of the vertex. The difference here is that, instead of considering the line mentioned, we consider the part of the contour which goes from the previous contour node to the next contour node of the vertex to relocate. Furthermore, only the fraction which traverse the cavity is

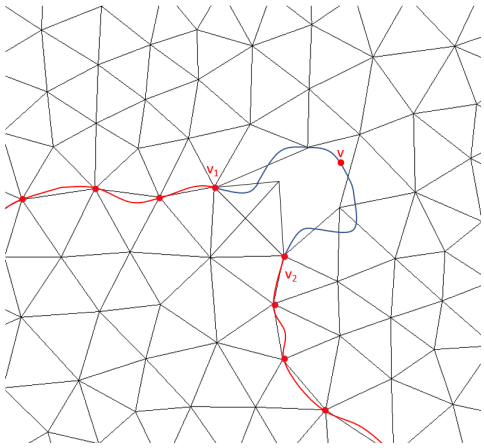
considered. The figure 3.10 shows a representation of such a line. Also, as the curve is not a straight line, the bisection method has been replaced by a traversal of the curves with a given constant step, making the function relatively slow for now. Furthermore, it is also important to take care that no triangle gets inverted during the process, as for the initial relocation operation. Unfortunately, it was noted that this method frequently lead to badly shaped element for a currently unknown reason. In consequence, it has not been used in the resulting meshes shown on section 3.3.

3.2.4 Attribution of the elements to their plateau

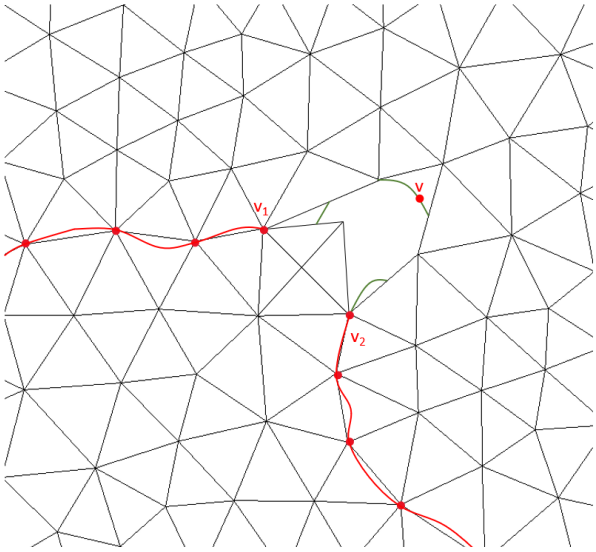
A last step requires to find for each faces the depth at which it should be set, or more precisely between which contour lines it was situated. As mentioned in chapter 2, the depth of a face is considered to be the one of the lowest contour that its plateau is in contact with. A special treatment has thus been applied to the zone touching only one isobath, i.e. which are completely encircled by only one such line, but this will not be detailed here.
The algorithm starts with the creation of an array of size equal to the number of vertices. Each of its values is set to `DBL_MAX`, the maximum value of a float defined in the package <float.h> in C. All the faces are then traversed in a loop. When encountering a face where one the three values of its vertices in the array are equal to `DBL_MAX`, it means that the face belongs to a zone which has not been checked and set to its correct depth yet. Consequently, its vertices which are not models are put in a queue q_1, which is traversed by popping its elements. When popping a vertex from q_1, we check all of its neighbours: if it is a model vertex from a contour, it is not put in the queue and its depth is stored; otherwise, it is put into two queues, into q_1, and another queue q_2 which will contain all the touched vertices of the current zone. Once q_1 gets empty, it means that all the vertices of the zone have been found, and



(a) Actual representation of the stored contour line



(b) If v is supposed to be relocated, we ensure that it stays somewhere on the blue segment.



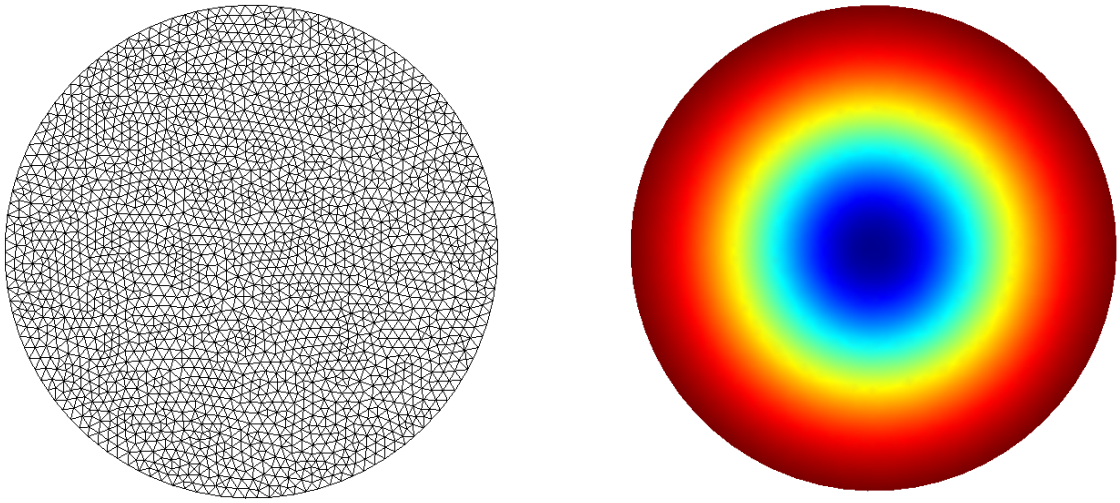
(c) The green segment shows the part of the blue segment contained in the initial cavity of v. Only those lines are considered when moving v, applying also a check to ensure no triangle gets inverted. This check is the same as the second one mentioned in section 1.2.2.

Figure 3.10: Line segments considered during the relocation of a vertex in a contour

that we stopped either at boundaries of the map, or at contour lines. From the stored depths of the contours encountered, we can compute the minimum depth and assigned it to all the vertices touched and store in q_2. After all the faces have been traversed, all the vertices have been given the depth of the lowest contour on the plateau which they belong to, and the final mesh can be written by taking this new assigned depth in consideration.
A last problem, which has already been mentioned, is the fact that the contour were not continuous with the first method. This lead this coloring method to leak on different plateaus, giving them the same depth while they should not. In order to overcome this, a last function had to be added to complete the contours. This algorithm has been only made in a quite naive way and will thus not be detailed here. Indeed, it simply consisted in traversing all the edges, and when one crossed a contour, set on of its end-vertex to be a model one, or do nothing if both were already model. This initial strategy has not been improved, as it was then decided to focus on the second method, considering the contours as continuous curves.

3.3 Obtained results

In this section, three methods will be considered and their outcomes will be compared based on three different bathymetries, already used in Chapter 1. An additional method to compute the contours has been used here, which is simply a naive method taking a mesh as input and setting each elements to the closest contour in terms of depth. This is the strategy which has been defined as the usual strategy, without applying the modifications to obtain a contour-fitted mesh described in this chapter.

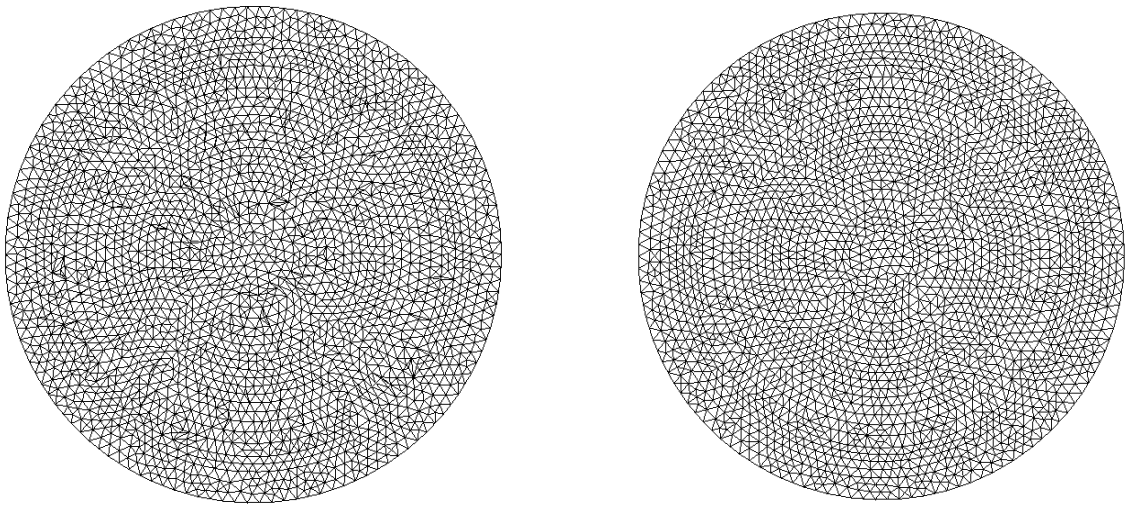


(a) (b)

Figure 3.11: On the left, the initial meshed used; on the right, the defined bathymetry

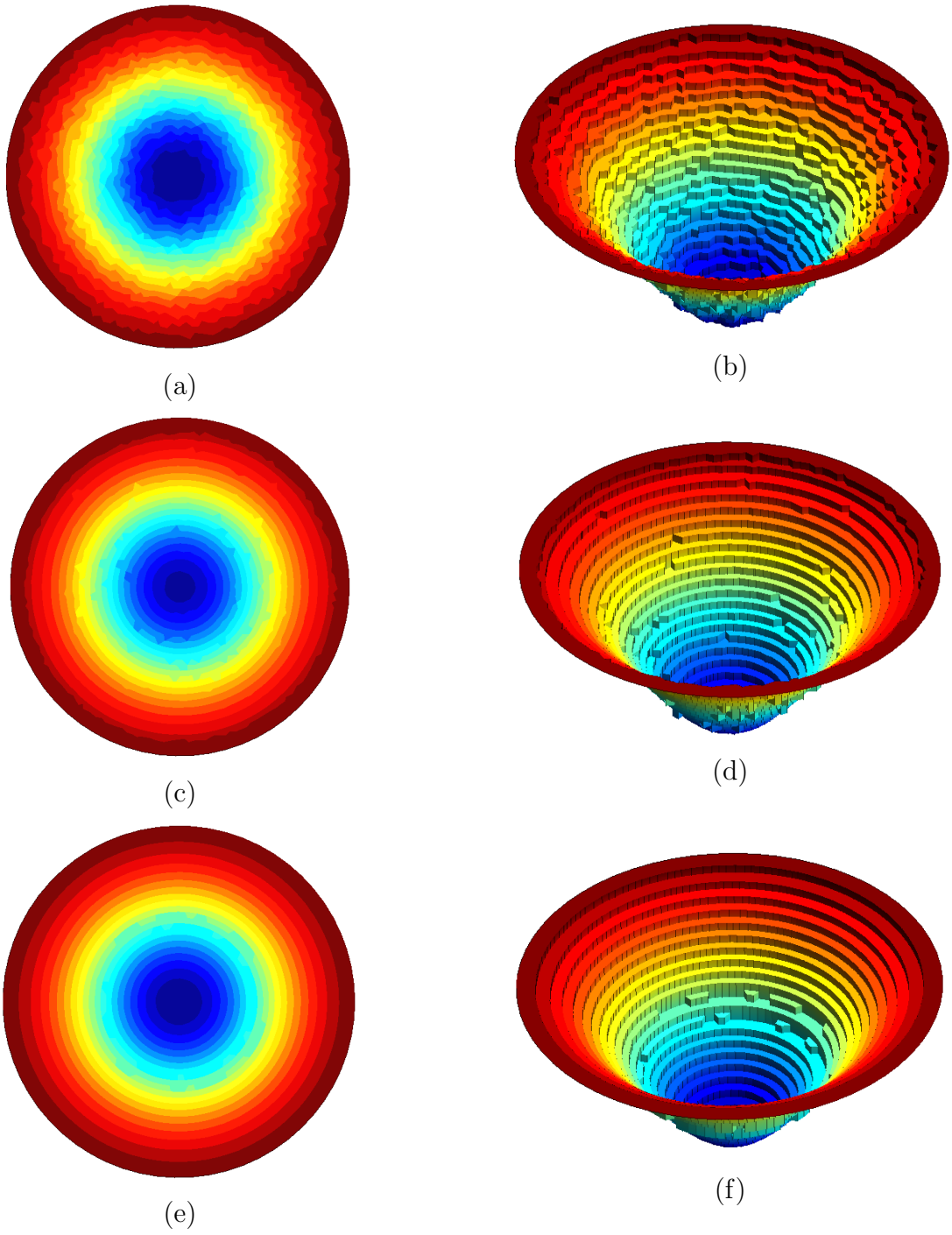
3.3.1 The circular mesh using a smooth axisymmetric bathymetry

The initial mesh used along with the bathymetry considered in this subsection are depicted on figure 3.11. It can be guessed that the size map has been set to constant. Furthermore, it is easy to see that the contour lines should have a circular shape.
When applying the two methods to fit the mesh to the contours on this initial mesh, with a number of contours equal to 20, the two meshes visible on figure 3.12 are obtained. The final bathymetries obtained when using the naive method, the pointwise method and the segmentwise method are then displayed on figure 3.13. Even though the contours are already more smooth when using the first method with respect to the naive one, they are much more when using the second method.



(a) (b)

Figure 3.12: Meshes obtained after applying the contour-fitting methods to the mesh of figure 3.11; on the left, using the first method; on the right, using the second method. We can see that the iso-contours are visible on both images, even though they are clearer with the second method



(a) (b) (c) (d) (e) (f)

Figure 3.13: Bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.11 using 20 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method; the left column shows the result from the top view; the right column shows the results in perspective

In order to show the robustness of both methods, figure 3.14 shows the resulting bathymetries obtained with a larger number of contours. We can see that the naive method here completely fails retrieving the shape of the contours. Even with respect to the first method where the contours are distinguishable, we can see that the second method performs roughly similarly as with a lower number of contours.

3.3.2 The custom mesh based on the bathymetry of the Lake Tanganyika

The next results were obtained using the initial mesh and bathymetry displayed at figure 3.15. The mesh used has been already pre-treated to have smaller elements around an iso-contour, as in Chapter 1. The results for the three methods, using 20 contours, are visible at figure 3.16 with a top view and at figure 3.17 in perspective. We can observe that the segmentwise method, in addition to having much smoother contour lines, also fits more correctly to the initial bathymetry and to the one obtained with the naive method. This proves again it superiority with respect to the pointwise method.

3.3.3 The custom mesh based on the smoothed bathymetry of the Seychelles

As a last example, we can compare the results obtained on the smoothed bathymetry of the Seychelles, obtained in chapter 1. The initial mesh and this bathymetry are shown on figure 3.18. It is useful to mention that the smoothed bathymetry used has been obtained in the same way as detailed in Chapter 1. Again, a top view and a view in perspective for the three different algorithm are visible respectively at figure 3.19 and 3.20.

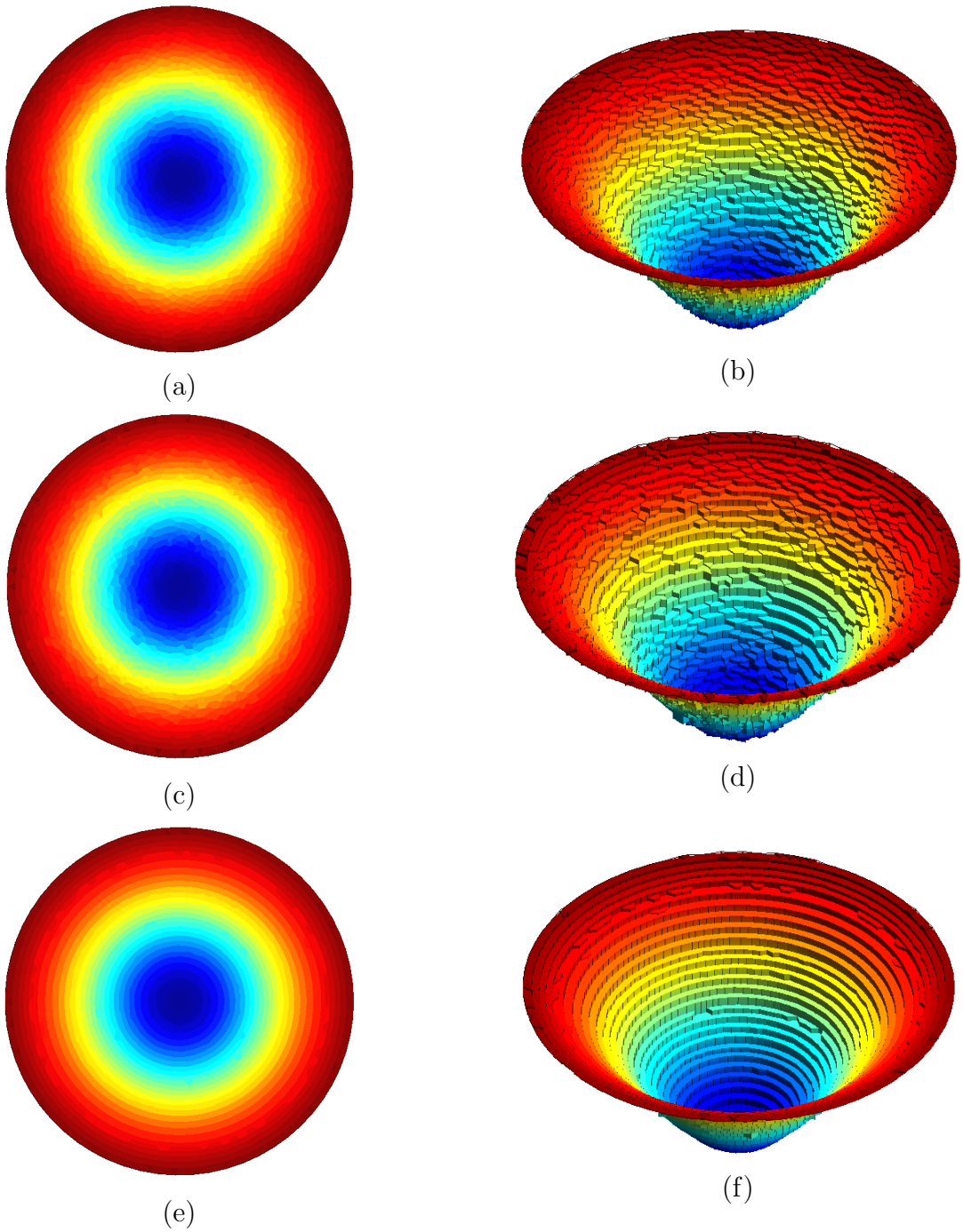
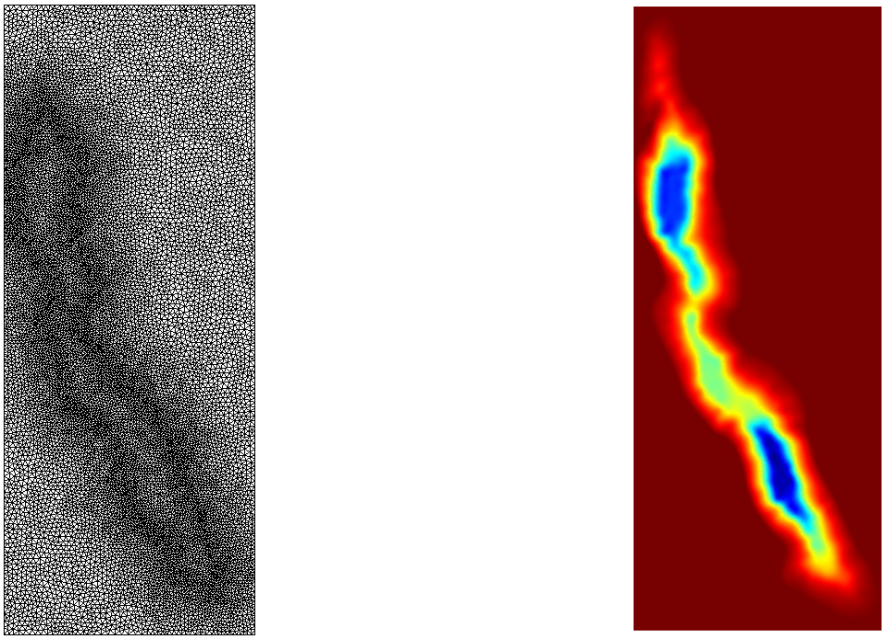
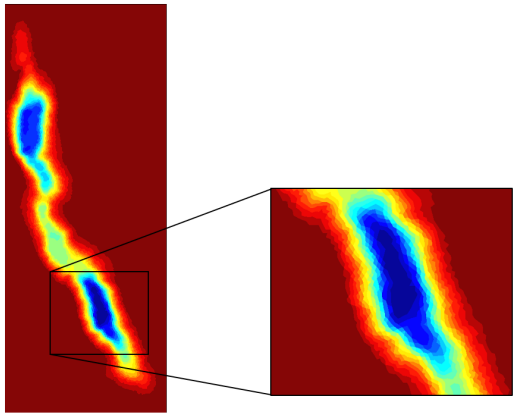


Figure 3.14: Bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.11 using 100 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method; the left column shows the result from the top view; the right column shows the results in perspective

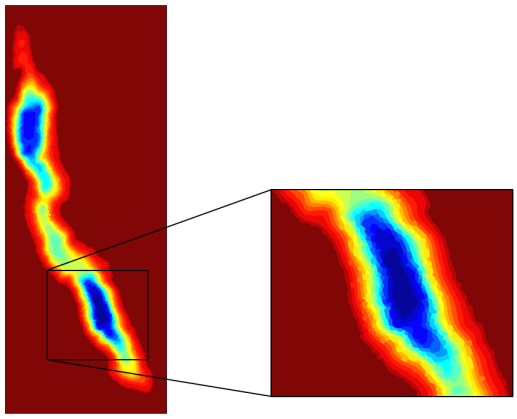


(a) (b)

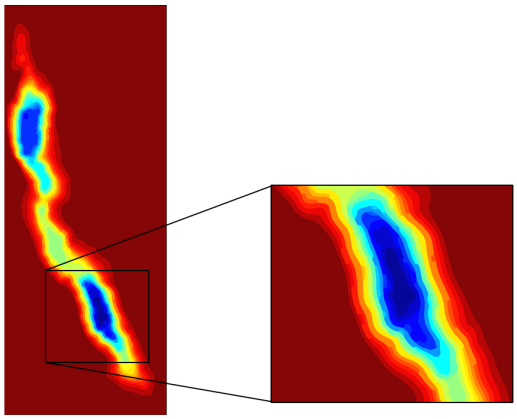
Figure 3.15: On the left, the initial meshed used; on the right, the bathymetry used



(a)

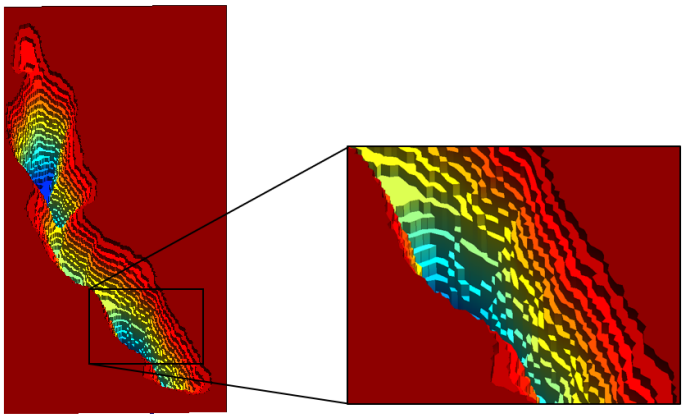


(b)

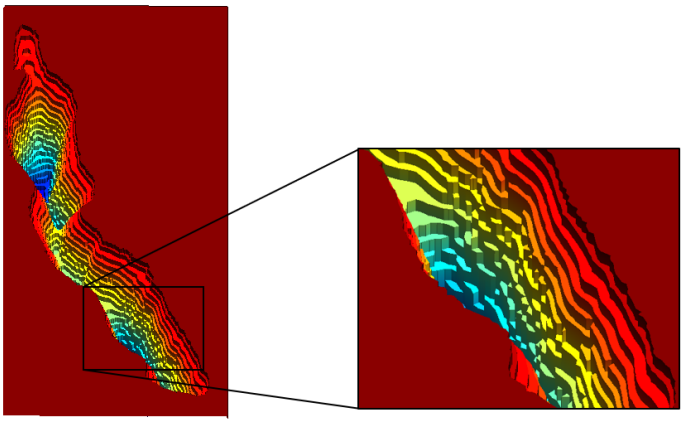


(c)

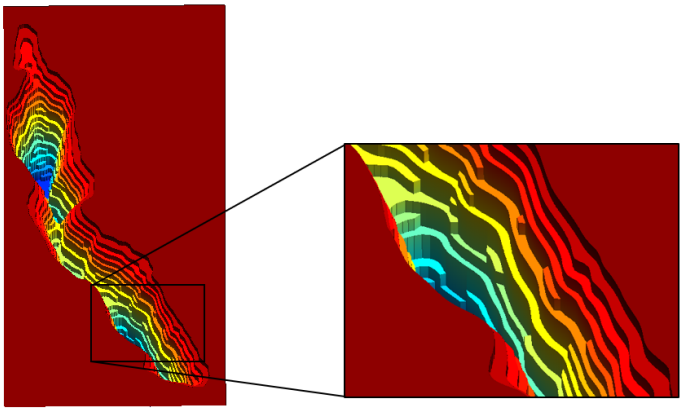
Figure 3.16: Top view with a zoom of the bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.15 using 20 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method.



(a)

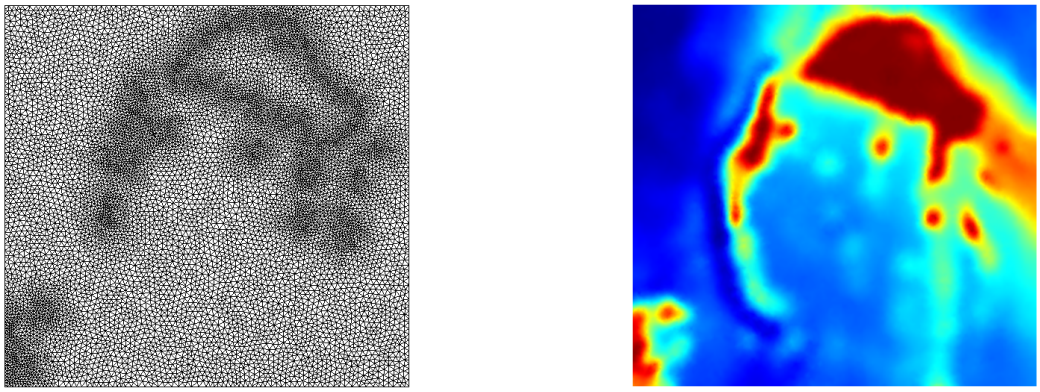


(b)



(c)

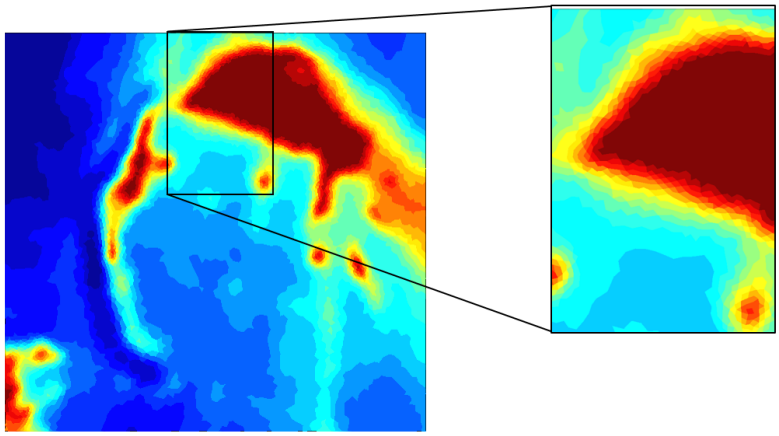
Figure 3.17: View in perspective and with a zoom of the bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.15 using 20 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method.



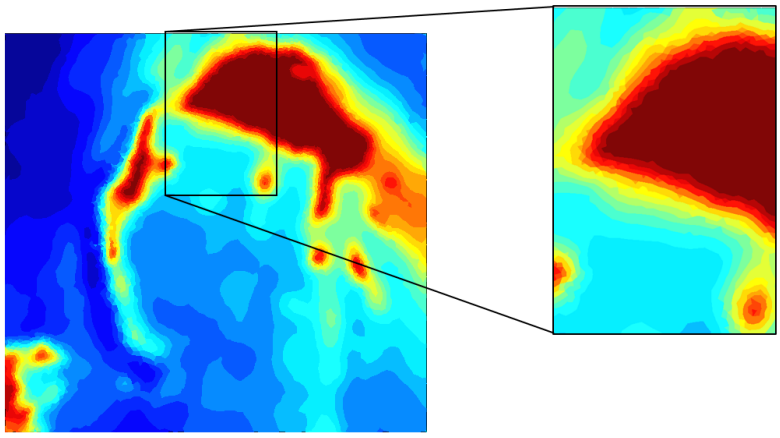
(a) (b)

Figure 3.18: On the left, the initial meshed used; on the right, the bathymetry used

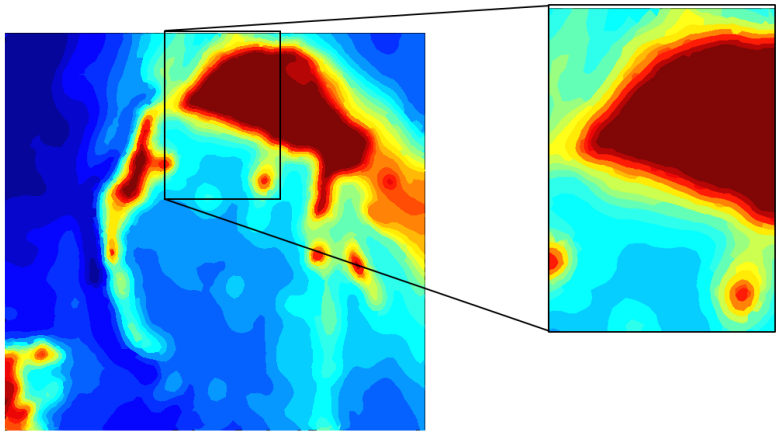
The number of contours used is still 20. We can clearly see that, even with such a complicated field of bathymetry, the two methods developed in this chapter allow to obtain much smoother lines between two plateaus. Also, it is even more obvious that the segmentwise method is more faithful to the original bathymetry, as most of the shapes of its plateaus looks like the ones obtained with the naive method, but with much smoother contours obviously.



(a)

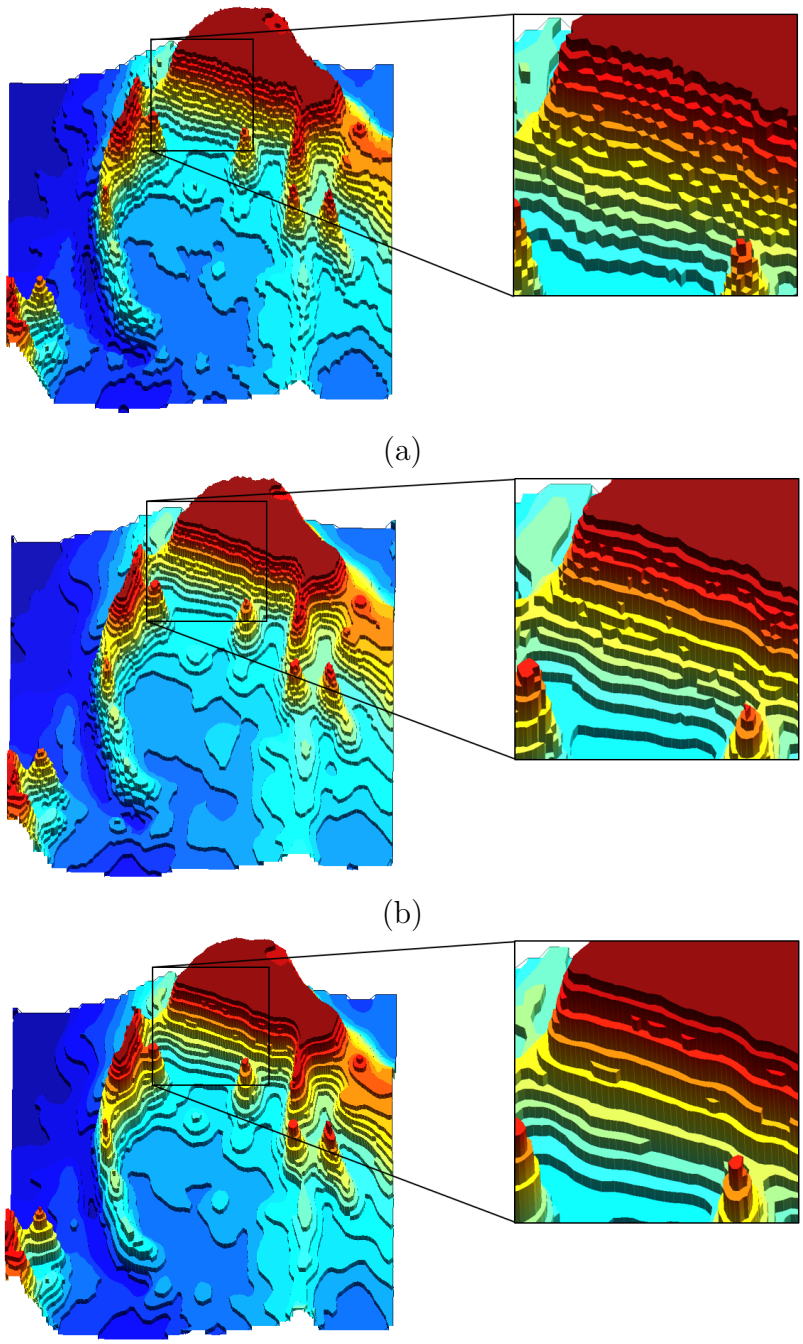


(b)



(c)

Figure 3.19: Top view with a zoom of the bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.18 using 20 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method.



(a)

(b)

(c)

Figure 3.20: View in perspective and with a zoom of the bathymetries obtained after applying the contour-fitting methods to the mesh of figure 3.18 using 20 contours; the first line shows the result obtained with the naive method; the second line with the pointwise method; and the third line with the segmentwise method.

3.4 Discussion of the results and on further improvements

As it can be visually concluded, the meshes obtained with the segmentwise method are definitely more faithful to the initial lines of isobathymetry than those obtained with the pointwise method. Furthermore, it obviously shows much smoother isobaths than the one obtained with the naive method, and even so with respect to the first method. Those final meshes happen to be really promising when used in real-life simulation, allowing to fit the bathymetry closely while containing a reasonable number of elements.
It is important to note that, throughout this report, most of the meshes qualities were comparable. The first quality index on the edges length τ oscillated around 72%, while the mean quality of the elements were always at least equal to 90%. The minimal element quality though, vary greatly, being sometimes lower than 10% when using the pointwise method, at least equal to 25% with the segmentwise mehtod, and being above 50% when the contours were not taken into account.
This last index is mainly due to some consideration that are still required to perform. More precisely, the elements of lowest quality mainly appears around the borders. Indeed, in the filtering strategy,it should be interesting to take it into consideration, for example by considering them as contour and giving them an order of importance of -1.
As we can see when comparing the results on figure 3.13 and figure 3.14, the segmentwise method is much more robust. Indeed, this lack of robustness of the pointwise method, mainly due the the fact that the contours had to be completed afterwards, was the main reason motivating the development of the segmentwise method. Nevertheless, we can note that it still suffers from some defaults.
For example, the edge recoveries happening when filtering the contour

are not guaranteed to complete. Indeed, we could face the case where a contour will be crossed while performing this step. The recovery will thus simply returns `NULL` and terminate. This has a dangerous implication, leading to the fact that some contours not be continuous anymore, provoking wrong results when attributing each faces to their plateaus.
Furthermore, the contours does not filters themselves. Of course, a segment should not filter other segments in its close neighbourhood, but we can easily imagine a contour line going far in both direction and reappearing close to the initial point. This would lead to a similar case as the one depicted on figure 3.5, but with the two contours having the same importance. It could be interesting to detect such occurrences, and to preform the filtering in the two direction of both parts of the curve.
Also, although not specifically detailed, a small additional step traverses all the contours and removes the curves which boiled down to only one vertices. A small add-on to this could be to removes also the curves composed of less than three of four vertices, in order to avoid plateaus of one or two elements.
Finally, a last suggested improvement concerns the final steps of attributing each elements to its plateau. Of course, this step is important as it is the one which gives the final shape to the boundary. As mentioned in section 3.2.4, the vertices are traversed. This has the implication that some faces are not given their correct plateau. Such cases only happens on the border of the mesh, where the finding of the plateau stops, and occurs in consequence mostly on the corner element. It could be interesting to use a traversal on the edges instead of the vertices, in order to properly overcome these issue. Indeed, the corner elements may have three vertices belonging to the boundary, but the usually at least one of their edges does not.

Conclusion and discussion on further work and possible improvements

This thesis aimed at developing a new kind of remeshing strategy in order to take the contour line of the bathymetry into account. The idea was to verify that meshes formed in such a way would help to reduce the numerical errors which are inevitably present during large scale finite element simulations. The type of remeshing could then be an interesting feature that could be added to mesh generator software, such as Gmsh and Seamsh [77].
The first chapter started by detailing the initial structures and algorithm that could be useful to perform this task. For this purpose, a data structure called the *half-edges* were used. Additional function allowing to read, write and enhance a mesh were also implemented in order to conveniently manipulate existing meshes. In addition, the possibility to refine a mesh based on line of isobathymetry was provided, giving the ability to choose which zones should contain the smoother contours. This refinement strategy goes along with the possibility to smooth the bathymetry of the mesh. Indeed, the smoothing being per-

formed based on the provided mesh, it is stronger on the zones where the mesh is coarser. This allows to define on which part of the mesh we should focus, and compute the most accurate contour lines.
In the second chapter, the interest of this new remeshing strategy was tested on a simple test case. The chosen problem consisted in solving the shallow water equations for a geostrophic equilibrium. The setting of the velocities and the bathymetry were detailed, and the corresponding elevation field was computed. Four different simulations settings were used: two in 2D, with a flat and a stepped bathymetry, and two in 3D with those same bathymetries. Each of those have been performed by using two distinct meshes, namely one which did not take the bathymetry into account, and one which was remeshed to fit to the isobaths. The results of the simulation revealed quite promising, allowing to obtain a similar decrease in energy with the flat bathymetry than with the stepped bathymetry with a contour-fitted mesh. Those preliminary results were considered satisfying enough to motivate the development of a process automating the fitting of a mesh the the contours of bathymetry.
In consequence, the third and final chapter dedicated to the complete description of this algorithm. Throughout the chapter, four steps were distinguished in order to completely fulfil this complicated task: the computation of the contours, their filtering at problematic areas, their handling during the main loop, and the attribution of each faces to its corresponding plateau. When detailing all those steps, two different methods were compared, the pointwise and the segmentwise method. Both had their corresponding challenges, although the first one required to implement easier algorithms than the second one. Finally, the obtained bathymetries were shown, using the two aforementioned methods and a really naive one serving as initial comparison point. Although the results were already convincing in some cases with the pointwise method, they were definitely much better using the segmentwise method. But the end of the chapter detailed the cost of such

strategy, namely tedious its implementation, resulting in the fact that some points needs to be improved. Nevertheless, the actual version already suffices to use the meshes to perform simulations.
Unfortunately, the time did not permit to achieve such simulations in order to compare the differences in the energy decay. Although we can be convinced that the would be a change for the better thanks to the results of Chapter 2, a considerable amount of work should still be provided in order to definitely validate the concept and quantify its impact. Finally, it would be interesting to focus on the flaws noted in the discussion of Chapter 3, and maybe also on the other strategy to efficiently retrieve the boundary of the initial mesh, briefly discussed in section 1.2.7. Even though the resulting contour-fitted mesh follows correctly the isobaths while maintaining a good quality, it could improve the robustness of the algorithms when dealing with some extreme cases.
Nevertheless, we really hope that this new method will be enhanced, tested and used in the future, and that it will be a useful feature to add for real-life simulation,s in order to improve their precision and their time efficiency.

Bibliography

[1] M. Lundstrom, "Moore's law forever?" *Science*, vol. 299, no. 5604, pp. 210–211, January 2003, https://doi.org/10.1126/science.1079567.

[2] R. Benoit and J. Mailhot, "A finite-element model of the atmospheric boundary layer suitable for use with numerical weather prediction models," *Journal of the Atmospheric Sciences*, vol. 39, no. 10, pp. 2249—-2266, September 1982, https://doi.org/0.1175/1520-0469(1982)039⟨2249:AFEMOT⟩2.0.CO;2.

[3] T. Maet, "Discontinuous galerkin models for coupled surface-subsurface flow," Ph.D. dissertation, Université catholique de Louvain, 2019, https://www.slim-ocean.be/index.php/phd-thesis/.

[4] C. Thomas, "Modelling marine connectivity in the great barrier reef and exploring its ecological implications," Ph.D. dissertation, Université catholique de Louvain, 2015, https://www.slim-ocean.be/index.php/phd-thesis/.

[5] R. J. S. et al., "Investigating the causes of the response of the thermohaline circulation to past and future climate changes," *Journal of Climate*, vol. 19, no. 8, pp. 1365—-1387, April 2006, https://doi.org/10.1175/JCLI3689.1.

[6] H. I. E. et al, "Simulation of aerobic and anaerobic biodegradation processes at a crude oil spill site," *AGU*, vol. 31, no. 12, pp. 3309–3327, December 1995, https://doi.org/10.1029/95WR02567.

[7] W. S. School, "How much water is there on earth?" November 2019, https://www.usgs.gov/special-topics/water-science-school/science/how-much-water-there-earth#:~:text=About%2071%20percent%20of%20the,in%20you%20and%20your%20dog.

[8] O. C. Zienkiewicz and I. K. Cheung, *The Finite Element Method in Engineering Science*, 2nd ed. McGraw-Hill, 1971.

[9] E. O. et al., "The particle finite element method — an overview," *International Journal of Computational Methods*, vol. 1, no. 2, pp. 267–307, 2004, https://doi.org/10.1142/S0219876204000204.

[10] R. L. K. et al., "Finite element method accuracy for wave propagation problems," *Journal of the Soil Mechanics and Foundations Division*, vol. 99, no. 5, May 1973, https://doi.org/10.1061/JSFEAQ.0001885.

[11] U. C. for Atmospheric Research, "Introduction to ocean models," 2007, http://stream1.cmatc.cn/pub/comet/MarineMeteorologyOceans/ocean_models/comet/oceans/ocean_models/print.htm.

[12] S. Legrand, "Maillages non-structurés en modélisation océanique," Ph.D. dissertation, Université catholique de Louvain, 2006, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=ER7H73KQ&content_type=application/pdf.

[13] E. Deleersnijder, J.-P. van Ypersele, and J. Campin, "An orthog-

onal, curvilinear coordonate system for a world ocean model," *Ocean Modelling*, vol. 100, pp. 7–20, 1993.

[14] S. Legrand, V. Legat, and E. Deleersnijder, "Delaunay mesh generation for an unstructured-grid ocean general circulation model," *Ocean Modelling*, vol. 2, pp. 17–28, 2001.

[15] S. Blaise, "Development of a finite element marine model," Ph.D. dissertation, Université Catholique de Louvain, November 2009, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=C9HVDYQC&content_type=application/pdf.

[16] J. F. Thompson, F. C. Thames, and C. W. Mastin, "Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies," *Journal of Computational Physics*, vol. 15, no. 3, pp. 299–319, July 1974, https://doi.org/10.1016/0021-9991(74)90114-4.

[17] J. Cui, "Body-fitting meshes for the discontinuous galerkin method," Ph.D. dissertation, Technical University of Darmstadt, February 2013, https://portal.dnb.de/opac.htm?method=simpleSearch&cqlMode=true&query=idn%3D1107770556.

[18] V. Legat, "Thematic map overlay," Université Catholique de Louvain, 2021, https://perso.uclouvain.be/vincent.legat/documents/meca2170/annotated-2122-slides/meca2170-3-LineSegment-2122.pdf?fbclid=IwAR3TkTSdnYiK-e1g0rsv_3NLhVfOJeJqPG0g4xm4hD40ZpFLAdX8nayDGa0.

[19] M. de Berg et al., *Computational Geometry : Algorithms and Applications*, 3rd ed. Springer, March 2008, vol. 3, https://people.inf.elte.hu/fekete/algoritmusok_msc/

terinfo_geom/konyvek/Computational%20Geometry%20-%20Algorithms%20and%20Applications,%203rd%20Ed.pdf.

[20] C. Geuzaine and J.-F. Remacle, "Reference manual of gmsh," 2022, https://gmsh.info/doc/texinfo/gmsh.html.

[21] E. Hanert, "Slim: a multi-scale model of the land-sea continuum," 2022, https://www.slim-ocean.be/.

[22] P.-E. Bernard, "Discontinuous galerkin methods for geophysical flow modeling," Ph.D. dissertation, Université Catholique de Louvain., 2008, https://www.slim-ocean.be/index.php/phd-thesis/.

[23] T. Kärnä, "Development of a baroclinic discontinuous galerkin finite element model for estuarine and coastal flows," Ph.D. dissertation, Université Catholique de Louvain, 2012, https://www.slim-ocean.be/index.php/phd-thesis/.

[24] P. Katz and S. S. Vincent, "Shapefile overlay using a doubly-connected edge list," in *Proceedings of the Class of 2007 Senior Conference*, C. S. Department, Ed., 2007, pp. 31—37, https://www.cs.swarthmore.edu/~adanner/cs97/f06/papers/proc.pdf#page=36.

[25] K. Crane and S. Coros, "Scotty3d developer manual," Carnegie Mellon University, 2016, http://15462.courses.cs.cmu.edu/fall2016/article/15.

[26] O. Lietaer, "Finite element methods for sea ice modeling," Ph.D. dissertation, Université Catholique de Louvain, October 2011, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=87SJEU8P&content_type=application/pdf.

[27] P.-E. Bernard, "Discontinuous galerkin methods for geophysical flow modeling," Ph.D. dissertation, Universite Catholique de Louvain, October 2008, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=BQS2QMPW&content_type=application/pdf.

[28] G. L. et al., "3d mesh compression," *ACM Computing Surveys*, vol. 47, no. 3, pp. 1–41, February 2015, https://doi.org/10.1145/2693443.

[29] K. Crane and S. Coros, "Halfedge makes mesh traversal easy," Carnegie Mellon University, 2016, http://15462.courses.cs.cmu.edu/fall2016/lecture/meshes/slide_022.

[30] C. A. D. F. Filho, *Smoothed Particle Hydrodynamics*, 1st ed. Springer, 2018.

[31] K. Crane and S. Coros, "Edge flip (triangles)," Carnegie Mellon University, 2016, http://15462.courses.cs.cmu.edu/fall2016/lecture/meshes/slide_025.

[32] C. Forest, H. Delingette, and N. Ayache, "Removing tetrahedra from a manifold mesh," *Proceedings of Computer Animation*, 2002, http://www-sop.inria.fr/asclepios/Publications/Forest/CA2002Forest.pdf.

[33] J.Huang and C. Menq, "Combinatorial manifold mesh reconstruction and optimization from unorganized points with arbitrary topology," *Computer-Aided Design*, vol. 34, no. 2, pp. 149–165, February 2002, https://doi.org/10.1016/S0010-4485(01)00079-3.

[34] B. Prabhakaran, "3d geometric modeling," UTD, https://personal.utdallas.edu/~praba/6v81/3dModeling.pdf.

[35] H. Chen, "Cs184 - computer graphics and imaging," UC Berkeley, 2022, http://hughchen.github.io/html/cs184/proj/index.html.

[36] R. Manual, "Isotropic remeshing," 2022, https://cmu-graphics.github.io/Scotty3D/meshedit/global/remesh/.

[37] C. F. Ollivier-Gooch and L. A. Freitag, "A comparison of tetrahedral mesh improvement techniques," *ResearchGate*, October 1996, https://doi.org/DOI:10.2172/414383.

[38] M. S. Shephard and M. Georges, "Automatic three-dimensional mesh generation by the finite octree technique," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 4, pp. 709–749, September 1991, https://doi.org/10.1002/nme.1620320406.

[39] C. Gutierrez, F. Gutierrez, and M.-C. Rivara, "Complexity of the bisection method," *Theoretical Computer Science*, vol. 382, no. 2, pp. 131–138, August 2007, https://doi.org/10.1016/j.tcs.2007.03.004.

[40] M. J. et al., "An efficient parallel algorithm for mesh smoothing," *ResearchGate*, December 1997, http://www.doi.org/10.2172/414390.

[41] C. Reid, "Hilbert sort," https://charlesreid1.com/wiki/Hilbert_Sort.

[42] D. Hilbert, "Ueber die stetige abbildung einer linie auf ein flächenstück," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891, https://charlesreid1.com/w/images/2/26/HilbertCurvePaper.pdf.

[43] H. V. Jagadish, "Analysis of the hilbert curve for representing two-dimensional space," *Information Processing Letters*, vol. 62,

no. 1, pp. Pages 17–22, April 1997, https://www.sciencedirect.com/science/article/abs/pii/S0020019097000148.

[44] M. Filipiak, "Mesh reordering in fluidity using hilbert space-filling curves," Ph.D. dissertation, University of Edinburgh, March 2003, http://www.hector.ac.uk/cse/distributedcse/reports/fluidity-decomp/fluidity-decomp.pdf.

[45] S. Aluru and F. E. Sevilgen, "Parallel domain decomposition and load balancing using spacefilling curves," *Electrical Engineering and Computer Science*, vol. 34, 1994, https://surface.syr.edu/cgi/viewcontent.cgi?article=1033&context=eecs.

[46] H. V. Jagadish, "Linear clustering of objects with multiple attributes," *ACM SIGMOD Record*, vol. 19, no. 2, pp. 332–342, June 1990, https://doi.org/10.1145/93605.98742.

[47] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete Computational Geometry*, vol. 18, pp. 305–363, 1997, https://doi.org/10.1007/PL00009321.

[48] B. Delaunay, "Sur la sphère vide," *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et naturelles,*, no. 6, pp. 793—800, 1934, http://www.mathnet.ru/links/585d8984881dc696febe42f459704cdd/im4937.pdf.

[49] P. J. Frey and P.-L. George, *Mesh Generation application to finite elements*. HERMES Science Publishing, January 2008, https://doi.org/10.1002/9780470611166.

[50] P.-L. George and H. Borouchaki, *Delaunay Triangulation and Meshing: Application to Finite Elements*. Hermès, 1998,

https://books.google.be/books/about/Delaunay_Triangulation_and_Meshing.html?id=HZGfI61PSUQC&redir_esc=y.

[51] S. Rebay, "Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm," *Journal Of Computational Physics*, vol. 106, pp. 125–138, 1993, https://doi.org/10.1006/jcph.1993.1097.

[52] N. Drakos, "Cs 373 combinatorial algorithms geometric triangulation," University of Leeds, 1993, https://www.cise.ufl.edu/~ungor/delaunay/delaunay/delaunay.html.

[53] H. Si, "On monotone sequences of directed flips, triangulations of polyhedra, and structural properties of a directed flip graph," *ArXiv*, vol. abs/1809.09701, 2018, https://arxiv.org/pdf/1809.09701.pdf.

[54] C. L. Lawson, "Software for c1 surface interpolation," NASA, TECHNICAL REPORT, 1977, https://ntrs.nasa.gov/api/citations/19770025881/downloads/19770025881.pdf.

[55] J. Gallier, *Dirichlet-Voronoi Diagrams and Delaunay Triangulations*, 2nd ed. Springer, July 2011, https://doi.org/10.1007/978-1-4613-0137-0_9.

[56] Y. N. Grigoryev, V. A. Vshivkov, and M. P. Fedoruk, *Numerical "Particle-in-Cell" Methods*, reprint 2012 ed. Walter de Gruyter GmbH, 2002, https://doi.org/10.1515/9783110916706.

[57] D. A. Field, "Laplacian smoothing and delaunay triangulations," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 4, no. 6, pp. 709–712, December 1988, https://doi.org/10.1002/cnm.1630040603.

[58] F. R. B. of Dallas, "Smoothing data with moving averages," 2022, https://www.dallasfed.org/research/basics/moving.aspx.

[59] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities," *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, vol. 79, no. 11, pp. 1309–1331, May 2009, https://doi.org/10.1002/nme.2579.

[60] P. P. Pebay and T. J. Baker, "Analysis of triangle quality measures," *MATHEMATICS OF COMPUTATION*, vol. 72, no. 244, pp. 1817—-1839, January 2003, https://www.ams.org/journals/mcom/2003-72-244/S0025-5718-03-01485-6/S0025-5718-03-01485-6.pdf.

[61] T. F. E. M. U. MATLAB, *Y. W. Kwon and H. Bang*, 2nd ed. Taylor Francis Group, 2000, https://doi.org/10.1201/9781315275949.

[62] A. Khennane, *Introduction to Finite Element Analysis Using MATLAB and Abaqus*. Taylor Francis Group, 2013, https://books.google.be/books?hl=fr&lr=&id=IHXfSNYXRD8C&oi=fnd&pg=PP1&dq=finite+element+abaqus&ots=cvs37kisYI&sig=LgurMZtsnP3Hlob3hR9KP146dZE#v=onepage&q&f=false.

[63] S. Chai, *Finite Element Analysis for Civil Engineering with DIANA Software*. Springer, 2020, https://books.google.be/books?id=JsXnDwAAQBAJ&pg=PA39&dq=finite+element+type+of+software&hl=fr&sa=X&ved=2ahUKEwjbgunF6Zj4AhVkgv0HHYusCBsQ6AF6BAgDEAI#v=onepage&q=finite%20element%20type%20of%20software&f=false.

[64] C. Geuzaine and J.-F. Remacle, "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing

facilities." *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009, https://gmsh.info/.

[65] D. W. Zingg, "Numerical dissipation," 2018, https://www.nas.nasa.gov/assets/pdf/ams/2018/introtocfd/Intro2CFD_Lecture7_Zingg.pdf.

[66] N. S. Wang, "Number system : Computer methods in chemical engineering," University of Maryland, 2012, https://user.eng.umd.edu/~nsw/chbe250/number.htm.

[67] T. Navarro and A. Piccialli, "Équilibre géostrophique," L'université Paris Sciences Lettres, 2022, https://media4.obspm.fr/public/ressources_lu/pages_planetologie-dynamique/equilibre-geo.html.

[68] J. Lambrechts, "Finite element methods for coastal flows:application to the great barrier reef," Ph.D. dissertation, Université Catholique de Louvain, March 2011, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=YXYLC55X&content_type=application/pdf.

[69] R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of the Atmospheric Sciences*, vol. 32, no. 4, pp. 680—-689, 1975, https://doi.org/10.1175/1520-0469(1975)032⟨0680:TDOFDM⟩2.0.CO;2.

[70] K.-H. Chang, *Computer-Aided Engineering Design: Chapter 18 Structural Design Sensitivity Analysis*, ser. e-Design. Academic Press, 2015, https://doi.org/10.1016/B978-0-12-382038-9.00018-1.

[71] E. HANERT, "Towards a finite element ocean circulation model," Ph.D. dissertation, Université Catholique de Louvain, May 2004, https://www.slim-ocean.be/wp-content/plugins/zotpress/lib/request/request.dl.php?api_user_id=778265&dlkey=YA7DY24C&content_type=application/pdf.

[72] I. Doghri, *Mechanics of Deformable Solids: Linear, Nonlinear, Analytical and Computational Aspects*. Springer, 2000.

[73] N. Grisouard, "Extraction of potential energy from geostrophic fronts by inertial–symmetric instabilities," *Journal of Physical Oceanography*, vol. 48, no. 5, pp. 1033—-1051, May 2018, https://doi.org/10.1175/JPO-D-17-0160.1.

[74] R. Ferrari and C. Wunsch, "Ocean circulation kinetic energy: Reservoirs, sources, and sinks," *Annual Review of Fluid Mechanics*, vol. 41, pp. 253–282, January 2009, https://doi.org/10.1146/annurev.fluid.40.111406.102139.

[75] C. Wang, "Bilateral recovering of sharp edges on feature-insensitive sampled meshes," *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, vol. 12, no. 4, pp. 629–639, 2006, https://doi.org/10.1109/TVCG.2006.60.

[76] B. K. Karamete, R. V. Garimella, and M. S. Shephard, "Recovery of an arbitrary edge on an existing surface mesh using local mesh modifications," *International Journal for Numerical Methods In Engineering*, vol. 50, no. 6, pp. 1389–1409, February 2001, https://doi.org/10.1002/1097-0207(20010228)50:6⟨1389::AID-NME75⟩3.0.CO;2-5.

[77] J. Lambrechts, "Seamsh," 2020, https://jlambrechts.git-page.immc.ucl.ac.be/seamsh/index.html.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**