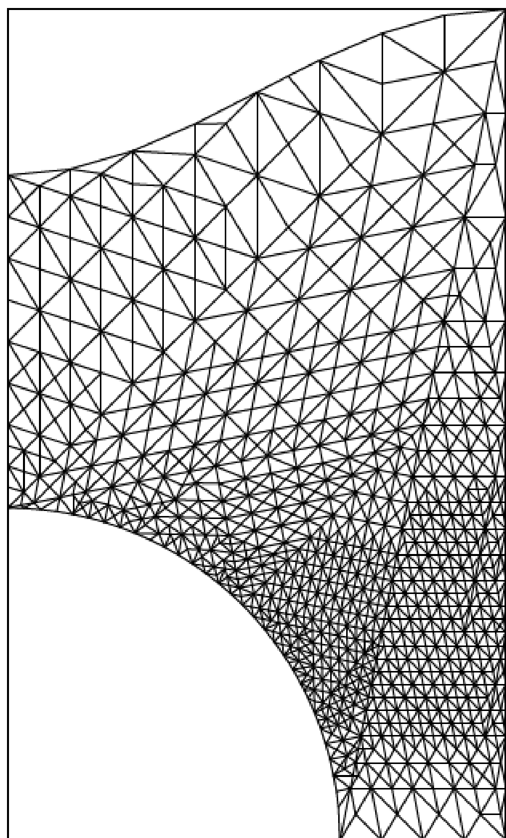
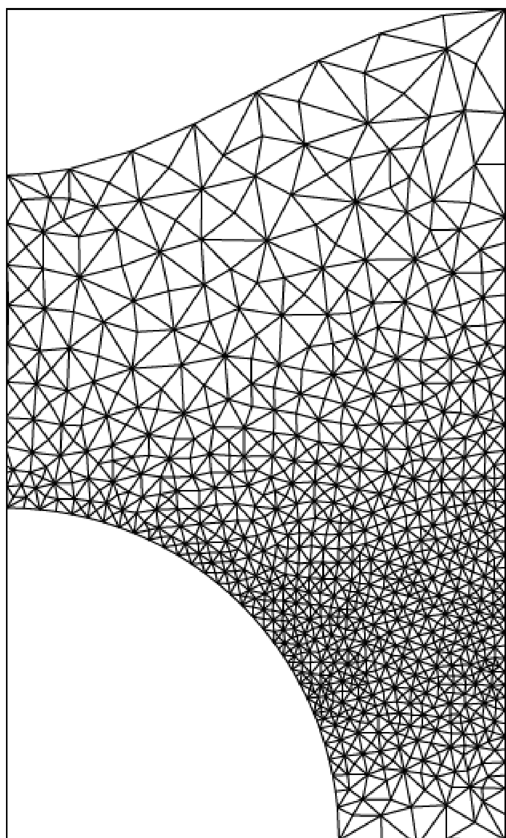
Study of Anisotropic Mesh Adaptivity and its Parallel Execution

Georgios Rokos
Imperial College London
Department of Computing





Abstract

In this report we study the basic principles of mesh adaptivity, a process of vital importance for solving partial differential equations, which dominate the Computational Fluid Dynamics field, in a numerical way using the Finite Element Method. We present the most common techniques that try to improve mesh element quality – in particular adaptive refinement, edge flipping and vertex smoothing, among which the latter seems to be the most interesting. It has the special attribute of leaving mesh topology intact while it can be implemented using a variety of different algorithms. We also discuss the ideas behind one particular category of vertex smoothing algorithms, the optimisation-based ones, which guarantee a minimum quality level of mesh elements. Additionally, we present a novel optimisation-based algorithm that tries to combine the benefits of adaptive refinement and edge flipping with the simplicity of vertex smoothing. The computational cost to perform mesh adaptation is notoriously high; therefore, it is essential to parallelise this process and towards this direction we examine a parallel framework, heavily based on a graph structure, for the correct and efficient execution of the aforementioned algorithms. It is shown that, using the appropriate graph colouring algorithms, it is possible to achieve high levels of parallelism, giving the opportunity to exploit architectures that manipulate thousands of threads, like nVIDIA's CUDA.

Contents

1 Introduction **1**
- 1.1 Computational Fluid Dynamics Background . . . 1
- 1.2 Definitions . . . 2
- 1.3 Motivations & Contribution . . . 3
- 1.4 Recent Work . . . 4
- 1.5 Report Outline . . . 4

2 Mesh Improving Algorithms **5**
- 2.1 Element Angles . . . 5
- 2.2 Algorithms for mesh improvement . . . 5
  - 2.2.1 Adaptive Refinement . . . 6
  - 2.2.2 Edge Flipping . . . 7
  - 2.2.3 Vertex Smoothing . . . 8
- 2.3 General algorithm for the adaptive solution of PDEs . . . 10

3 Optimisation-Based Vertex Smoothing **11**
- 3.1 Maximising Minimum Angle . . . 11
- 3.2 Adjusting Element Area Ratio and Orientation . . . 13

4 Parallel Execution of Mesh Adaptivity **21**
- 4.1 Important graph-structures . . . 21
- 4.2 Correct Parallel Execution . . . 21
  - 4.2.1 Elemental Operations . . . 23
  - 4.2.2 Data Consistency . . . 23
- 4.3 General Parallel Framework . . . 23
  - 4.3.1 Task Graph . . . 24
  - 4.3.2 Parallel Algorithm . . . 24
- 4.4 Elemental Operations . . . 24
  - 4.4.1 Element Bisection Elemental Operation . . . 25
  - 4.4.2 Edge Flipping Elemental Operation . . . 26
  - 4.4.3 Vertex Smoothing Elemental Operation . . . 29

5 Conclusions and Future Work **31**

Chapter 1

Introduction

This report constitutes a literature survey on the topic of anisotropic mesh adaptivity. The aim of this research is to provide the essential algorithmic background for the efficient implementation of a relative application in the future, using modern, parallel, high-performance architectures like nVIDIA's CUDA. This first chapter presents the necessary background knowledge, explains the basic terminology involved with adaptation of meshes, gives the motivations behind the whole venture and outlines the expected contribution.

1.1 Computational Fluid Dynamics Background

Fluid dynamics is the science that addresses problems related to the motion of fluids. Study of fluid motion relies on some fundamental equations, known as *Navier-Stokes Equations*. These are a set of partial differential equations (PDEs) and, in essence, are a form of Newton's Laws applied to fluid elements. In the general case, the fundamental equations cannot be solved in an analytical way. Instead, they can be discretised and approximately solved by using numerical analysis techniques. One such common technique used in computational fluid dynamics (CFD) is the *Finite Element Method* (FEM).

In the Finite Element Method, the domain of interest is discretised into elements, which are essentially smaller sub-domains of the initial one and form a structure referred to as *mesh*. These elements usually have a triangular (for two-dimensional domains) or tetrahedral (for three-dimensional domains) shape. The differential equation is then discretised as well and solved inside each of these elements. In general, there is no regular pattern in the topology of elements, so meshes are considered to be *unstructured*. Unstructured meshes offer greater flexibility in the Finite Element Method, but their representation is more complex and models based on unstructured meshes exhibit higher computational cost compared to structured mesh models [Piggott et al., 2009]. An example of space discretisation is shown in Figure 1.1 [Labelle, 1999].

Mesh quality is crucial in the solution of the Finite Element Method. It can be measured using some criterion, such as angle size or aspect ratio of all elements in the mesh. Elements of low quality, a usual result of attempts to discretise complex geometries, can seriously affect both the computational cost and the approximation accuracy [Freitag et al., 1995]. The importance of mesh quality in the solution of the Finite Element Method has motivated many researchers into finding techniques which improve mesh characteristics. These techniques reside in the wider context of *adaptive algorithms*, which is a major tendency in the field these days.

The solution process and accuracy can be further benefited by exploiting the fact that many fluid dynamics problems exhibit some kind of *anisotropy*. This means that the solutions to such problems exhibit directional dependencies. In these cases, the solution of the Finite Element Method can be sped up if the mesh has a suitable orientation, i.e. it is *anisotropic* [Formaggia et al., 2004].

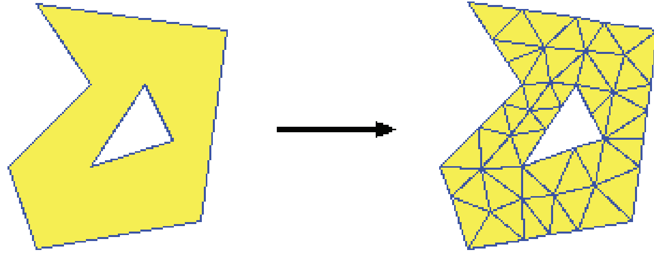


Figure 1.1: Example of space discretisation, resulting in an unstructured mesh. (figure from [Labelle, 1999])

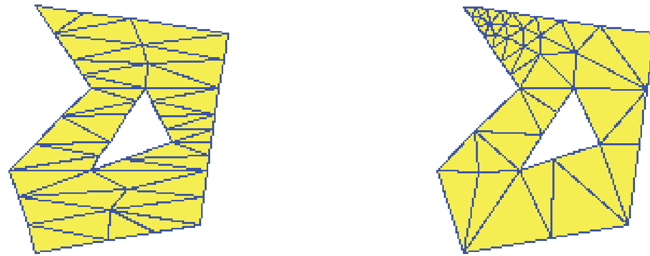


Figure 1.2: Example of an anisotropic mesh. Mesh elements have different shape and size in different locations on the mesh. (figure from [Labelle, 1999])

In practice, it means that the elements of the anisotropic mesh (triangles or tetrahedra) can have different shape and size in different locations on the mesh. An example of an anisotropic mesh is shown in Figure 1.2 [Labelle, 1999].

1.2 Definitions

There are many definitions which will be encountered throughout this report. It should be mentioned that many of these terms are use among different groups of people in different contexts, i.e. the same concept can be described by two different terms or the same term is used to describe different things, depending on the researcher. The definitions given here are in accordance with [Farrell, 2009] and constitute our way of referring to the concepts of this research.

- *Adaptive Algorithm*: An algorithm used in this context: the goal of the computation is set and the algorithm modifies the quality of the approximation to reach it.
- *A posteriori Error Estimation*: Error bounds that provide an estimation of the approximation error cannot generally be known prior to the solution of the PDE, because they depend on the unknown exact solution. A posteriori error estimates are based only on the approximate solution, so they are computable. An adaptive algorithm uses these a posteriori error estimates to "adapt" the discretisation of the problem accordingly.
- *h-adaptivity*: In this kind of adaptivity, connectivity of the mesh may be changed by adding vertices, removing vertices, swapping edges and other operations that modify the topology of the mesh.
- *r-adaptivity*: In this kind of adaptivity, connectivity of the mesh remains intact; only the locations of existing vertices are changed.

- *p-adaptivity*: The act of changing the local polynomial order of the basis functions associated with the mesh. We do not examine this kind of adaptivity in this report.
- *Mesh Adaptivity*: The kind of adaptivity which encompasses h-adaptivity and r-adaptivity.
- *Adaptive Re-meshing*: Mesh adaptivity method which produces adapted meshes that may be entirely different from the original mesh.
- *Global Re-meshing*: The act of generating an entirely new mesh of the same domain of interest, which satisfies the error approximation criteria.
- *Local Re-meshing*: The act of optimising only a small area of the mesh, instead of the whole one.

1.3 Motivations & Contribution

The computational cost related to this family of applications is notoriously high, even when the most efficient algorithms are used. Among all steps involved in the solution of a CFD PDE, the construction of a convenient mesh is a heavy task of unparalleled importance. Therefore, it is interesting to make an attempt at porting the *Anisotropic Mesh Adaptation* process on a modern, parallel, scalable, high-performance platform in order to speed up time-consuming computations, increase their accuracy and establish the base for meaningful simulations, such as ocean flow or weather prediction.

nVIDIA's *Compute Unified Device Architecture* (CUDA) is a novel, revolutionary platform thanks to which the tremendous computational power lying inside a graphics processing unit can be unleashed and used for non-graphics purposes. At this critical moment in the industry of processing units, where traditional architectures compete heavily against modern tendencies and their unification is more probable than ever before, the results of this venture are more than just interesting. The particular characteristics of CUDA can be summarised in two phrases: simultaneous multi-processing (SMP) and great floating-point computational power. This is the reason behind our choice to study the anisotropic mesh adaptivity in the context of this architecture and its special characteristics, trying to describe parallel algorithms and efficient, floating-point intense techniques like optimisation-based vertex smoothing, which can take advantage of CUDA's hundreds of FPUs.

Contribution of this research can be summarised in the following points:

- Accumulate necessary background knowledge – mesh improving algorithms, optimisation-based techniques, parallelisation algorithms – with the aim of building a relative CFD application
- Propose a new method for improving mesh quality by taking advantage of anisotropic characteristics of the CFD problem under consideration, an approach which might produce a better mesh without changing the number of its elements, resulting in a speed up of the solution process
- See how this new algorithm can be combined with other ones in order to achieve even better results
- Investigate the extent to which algorithms proposed for parallel execution suffice to port the application on massively multi-threaded architectures, like nVIDIA's CUDA
- Provide C++-like pseudo-algorithms which outline the basic functions and variables a concrete C++ implementation should consist of

1.4 Recent Work

As it was mentioned before, this literature survey was conducted with the aim of building a future application specifically for the CUDA platform. There exists a relative, general-purpose application for the numerical solution of fluid dynamics problems called "Fluidity". Fluidity is an open-source project developed at the Applied Modelling and Computation Group of Imperial College London. It is composed of multi-phase computational fluid dynamics code and can manipulate arbitrary, unstructured finite element meshes. It can solve a wide variety of CFD problems including, but not limited to, modelling of complex oceanic flow, heat transfer and fluidised beds [AMCG, 2009].

From an architectural point of view, Fluidity runs on conventional hardware, engaging multi-block explicit domain decomposition and the Message Passing Interface (MPI) in order to run on parallel processing systems such as clusters and supercomputers. GPU hardware has started finding its place in the supercomputing field and it would be interesting to enable Fluidity to run on such hardware. Our future application could be either a standalone one or it could be implemented as a CUDA module embedded into Fluidity.

1.5 Report Outline

The rest of the report is organised as follows: Chapter 2 presents the algorithms involved in the mesh adaptation process and in Chapter 3 we present the proposal by [Freitag et al., 1995], which takes vertex smoothing a step further with optimisation-based algorithms. In the same chapter we demonstrate our own optimisation-based proposal which, unlike [Freitag et al., 1995], tries to take advantage of anisotropic characteristics exhibited by a PDE. Chapter 4 focuses on a framework for the efficient and scalable parallel implementation of a mesh improving application. The report ends with an exposition of the conclusions we were led to by this research along with proposals for future work in Chapter 5.

Chapter 2

Mesh Improving Algorithms

This chapter gives a description of three common mesh-optimisation methods: adaptive refinement through element bisection, edge flipping and vertex smoothing. It is largely based on the ideas presented by [Freitag et al., 1998]. These three improvement strategies were chosen among other ones because a lot of work has already been done on their efficient and correct parallel execution.

2.1 Element Angles

The quality of an element in a two-dimensional case is determined to a great extent by the size of its angles. It is generally considered critical that any angle of an element is not small. This condition is known as the *minimum angle condition*. It has been shown by [Babuška and Aziz, 1976] that the minimum angle condition is not essential – in contrast it can be restrictive in anisotropic functions – and the *maximum angle condition* should be used instead, i.e. no angle should be close to $180\,^\circ$. The community, however, takes into consideration the first condition in many cases.

The strategies of adaptive refinement and edge flipping, which will be presented in the following sections, are based on this condition to improve the quality of a mesh. Vertex smoothing, on the other hand, may or may not be based on it. The most common vertex smoothing approach is the *Laplacian* one, which is renowned for its simplicity and speed of execution, but it does not take into account the minimum angle condition. In the third chapter we will present a variant of vertex smoothing which performs angle-based optimisation.

Before describing these optimisation algorithms, we will give some definitions regarding the mesh. Let V be the set of mesh vertices, $V = \{v_i | i = 1, \ldots, n\}$ and T the set of mesh elements, $T = \{t_a | a = 1, \ldots, m\}$. Although the algorithms presented here can be executed using mesh elements of any shape, we will only consider triangular elements in this report. Let E be the set of mesh edges, i.e. vertex pairs $(u, v), u, v \in t, t \in T$. Let D be the set of edge pairs $(s, t), s, t \in E$, so that s and t share a common vertex.

2.2 Algorithms for mesh improvement

Mesh improving algorithms can be generally grouped into two main categories: one category contains algorithms that affect the mesh topology by changing the number of elements (adding new ones, merging two or more old ones or replacing a group of elements with a different group); the other one contains algorithms that leave mesh topology intact and only attempt to improve quality by relocating element connection points.

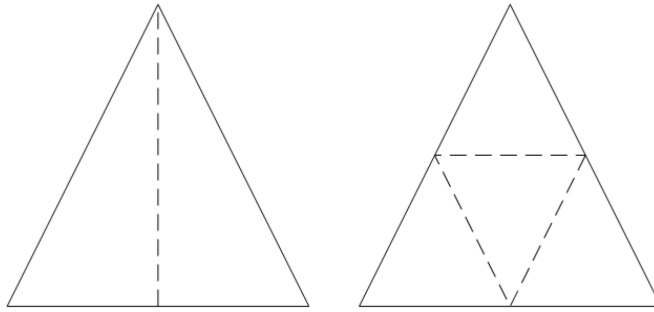


Figure 2.1: Bisection (left) and regular refinement (right) of an element.

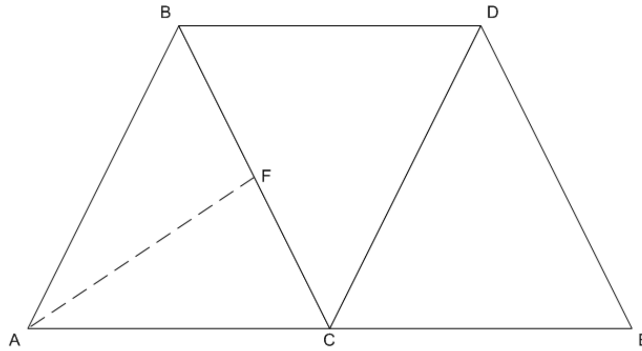


Figure 2.2: Example of a non-conforming mesh as a result of bisection: bisecting triangle ABC across the edge BC results in a new vertex F which lies in the middle of edge BC, being the "fourth" vertex of triangle BCD.

2.2.1 Adaptive Refinement

Adaptive refinement is a global optimisation method which tries to improve overall mesh quality by dividing mesh elements to smaller ones. It is a kind of *h-adaptivity*. There are two main techniques for dividing a mesh element, *bisection* and *regular refinement*. Pictorially, these are shown in Figure 2.1. In bisection the element is divided across one of its edges (usually the longest one) whereas regular refinement is accomplished by creating new edges connecting the middle points of the existing ones. Algorithms that implement the above techniques are relatively cheap in the computational sense and for two-dimensional mesh refinement the angles of the resulting elements lie in a range away from 0° and 180° [Freitag et al., 1998].

On the other hand, bisection or regular refinement can lead the the creation of non-conforming meshes, i.e. meshes in which one or more vertices of an element lie in the middle of edges without belonging to the element these edges are part of. In other words, every element in a conforming mesh meets another element at a vertex or shares a whole edge with the other element or these two elements are not adjacent at all. For example, in Figure 2.2, bisection of triangle ABC across the edge BC creates a new vertex F which lies in the middle of edge BC, without being an edge of triangle BCD, therefore creating a non-conforming element BFCD. Due to this property, adaptive refinement has to be *propagated* to neighbouring elements in order to eliminate non-conformities (for example, in Figure 2.2, triangle BCD should be bisected along vertices D and F).

Algorithm 1 describes a simple method to implement adaptive refinement, ending up with a conforming mesh. This algorithm is known as *Rivara's Bisection Algorithm*. It should be noted that the length of propagation is equal to the number of iterations of the while-loop and it depends

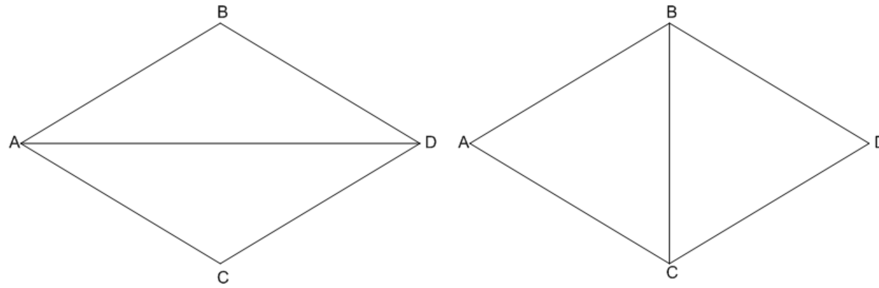


Figure 2.3: Flipping the edge AD results in two triangles whose minimum angle is larger than that of the original ones.

on the division algorithm that is used. It has been shown that this length is finite, i.e. the loop actually comes to an end at some time [Rivara, 1984]. In practice it is $O(1)$, but in the worst case it is $O(n)$, where n is the number of vertices in the mesh.

Algorithm 1 Simple element division algorithm that results in a conforming mesh.

```
Set S = {elements that need to be divided};
while(!S.empty())
{
    Set R = {};
    for(Element e : S)
    {
        e.divide();
        R.add(non-conforming elements generated
                        by the above intersection);
    }
    S.setEqualTo(R);
}
```

2.2.2 Edge Flipping

Edge flipping is a local optimisation technique which tries to improve global mesh quality. Like adaptive refinement, it is a kind of *h-adaptivity*. The act of edge flipping is depicted in Figure 2.3 and consists of flipping an edge shared by two elements. There are many criteria to decide whether an edge should be flipped. The most common one is the *maxmin angle* criterion, i.e. the edge is flipped only if the minimum angle of the resulting elements is larger than the minimum angle of the original elements. This procedure results in a *Delaunay Triangulation*, a triangulation in which the minimum element angle in the mesh is the largest possible one with respect to all other triangulations of the mesh [Kulkarni et al., 2009].

In practice, we usually want to make several passes over the mesh in order to improve its quality and edge flipping, being a local optimisation technique, could be combined with other techniques to finally achieve a global optimum in mesh quality. As we have already seen in the case of adaptive refinement, edge flipping can also propagate to neighbouring elements. If an edge is flipped, the topology of the mesh is changed and so is the criterion for its adjacent edges. A representative example can be seen in Figure 2.4. Initially, we do not want to flip edge AD as the resulting triangles would have a smaller minimum angle than the original ones. Edge BD, however,

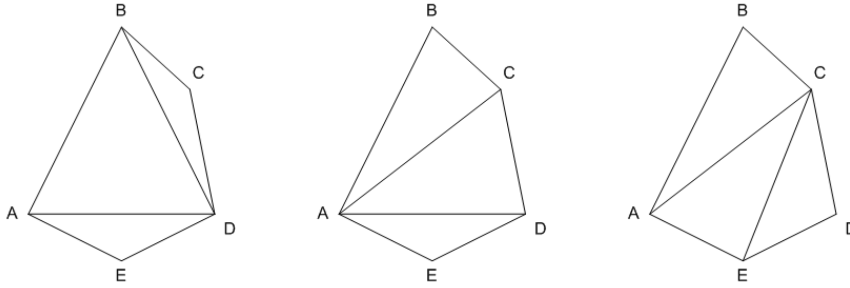


Figure 2.4: In the left figure, edge AD should not be flipped, but edge BD should, resulting in the topology of the middle figure. After this flipping, however, edge AD should be flipped as well, resulting in the topology of the right figure.

satisfies our *maxmin angle* criterion, so we flip it. After that, the mesh topology has changed and now flipping edge AD does make sense indeed, so we flip it as well. As it was stated earlier, the resulting triangulation is a *Delaunay Triangulation*.

Having taken the need for propagation into account, the algorithm for edge flipping is described in Algorithm 2. Just like in adaptive refinement, the propagation length is equal to the number of iterations of the while-loop. Probably, the worst case length is equal to the number of mesh edges.

Algorithm 2 Edge flipping algorithm that takes propagation into account.

```
Set S = {edges that should be flipped};
while(!S.empty())
{
    Set R = {};
    for(Edge e : S)
        if(flipping satisfies maxmin angle criterion)
        {
            e.flip();
            R.add(e.getIncidentEdges());
        }
    S.setEqualTo(R);
}
```

2.2.3 Vertex Smoothing

Vertex smoothing is a technique which tries to improve mesh quality by relocating vertices. Contrary to the previous two methods, vertex smoothing does not change mesh topology, i.e. it is a kind of *r-adaptivity*. Like edge flipping, it is a local optimisation technique since it considers a single vertex at a time. The global optimum in mesh quality is achieved through a series of such local optimisation problems, each one involving only a single vertex and the vertices belonging to the elements that share the single vertex. All elements that are affected by a vertex relocation form an area which we refer to as *cavity*. An example of applying this technique can be seen in Figure 2.5. The vertex under consideration is the one marked with a big black circle. Neighbouring vertices that are taken into account while performing smoothing are marked with smaller black circles. The cavity is marked with light-orange colour.

Due to the nature of this method, it may be necessary to relocate a vertex over and over

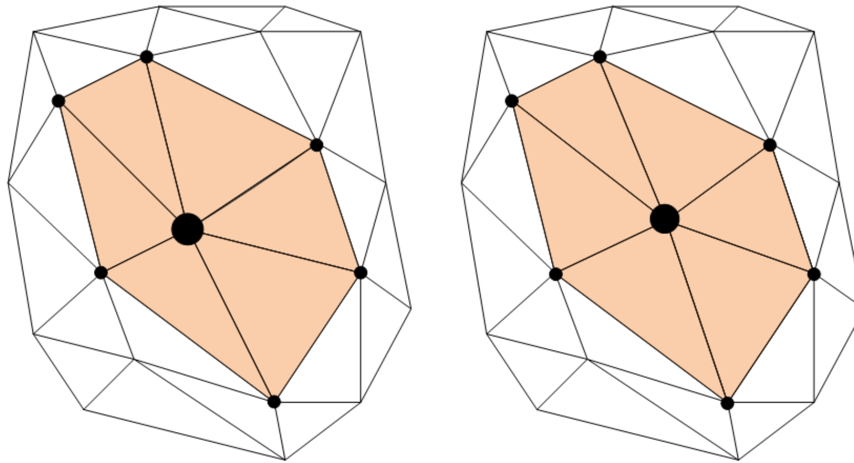


Figure 2.5: Vertex smoothing example. The vertex under consideration is the one marked with a big black circle. The local problem area is the light-orange one. All neighbouring vertices are denoted with small black circles. Left figure shows the local problem cavity before smoothing. Right figure shows the result of local smoothing.

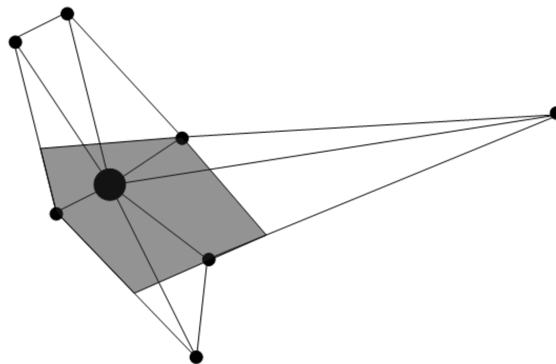


Figure 2.6: Example of an invalid mesh. If the vertex under consideration is relocated to a position outside the grey zone, some elements will have negative area and the mesh will be invalid.

again as neighbouring vertices also change position in the mesh. Therefore, the local optimisation problem should be computationally inexpensive. The simplest and fastest solution is the *Laplacian smoothing* technique. In Laplacian smoothing, the vertex under consideration is relocated to the geometric centre of the positions of all neighbouring vertices. Although Laplacian smoothing is an inexpensive technique, the result is not necessarily the optimal one, i.e. there is no guarantee that the geometric centre is the best position to relocate a vertex to. In fact, element quality might even get worse [Freitag, 1997]. In addition, the new location for the vertex under consideration is restricted to be within a certain area. This can be easily seen in Figure 2.6. If the central vertex is moved outside the grey area, then one or more triangles will be inverted, i.e. have negative area, and the mesh will become invalid. This grey area is called the *interior convex hull* of the cavity.

Optimisation-based smoothing is a family of techniques which guarantee that the resulting mesh will be valid and at least of the same quality as the original one. In the third chapter we will present an optimisation-based algorithm by [Freitag et al., 1995] that is based upon optimisation techniques for non-differentiable equations and keeps computational load to reasonable levels. No matter what smoothing technique we use, the general description of the vertex smoothing procedure is given

in Algorithm 3. It should be noted that the total number of iterations is usually pre-defined as MAX_ITER; however, the value of MAX_ITER needed to achieve a desired quality level is not known a priori and, in fact, it is not even guaranteed that such a quality level can be actually achieved [Freitag et al., 1998].

Algorithm 3 The vertex smoothing algorithm running MAX_ITER times.

```
Set S = {vertices that should be relocated};
int iteration = 0;
while(iteration < MAX_ITER)
{
    for(Vertex v : S)
        v.relocate();

    iteration++;
}
```

2.3 General algorithm for the adaptive solution of PDEs

In this section we will describe a general algorithm that can be used in order to solve a PDE using one or more of the aforementioned mesh improvement techniques. The algorithm definition is given in Algorithm 4. The algorithm begins by swiping over the mesh and making some early improvements performing edge flipping and vertex smoothing. Avoiding element bisection in the first place has the advantage of retaining the anisotropic characteristics of the mesh. Following step is to solve the PDE on the improved mesh and and make estimations about solution errors. While these errors are above a predefined tolerance, the algorithm tries to improve local quality in problematic areas and re-solves the PDE to check for new erroneous values in the solution. It should be noted that when solving a PDE on a mesh, i.e. during the *solution process*, we want this mesh to be conforming. However, we allow non-conforming vertices to lie on an element's edges during the *refinement process*.

Algorithm 4 General algorithm for the adaptive solution of a PDE.

```
Mesh mesh = new Mesh(PDE);
mesh.refineUsingFlippingAndSmoothing();
PDE.solveOn(mesh);
while(PDE.getSolutionErrorEstimation() > USER_DEFINED_THRESHOLD)
{
    Set S = {elements of low quality};
    mesh.refineUsingBisectionOfElements(S);
    mesh.refineUsingFlippingAndSmoothing();
    PDE.solveOn(mesh);
}
```

In the following chapter it is shown how the process of vertex smoothing can be executed using optimisation criteria, in an attempt to get better results in element quality or to "align" mesh adaptation with a problem's anisotropic behaviour.

Chapter 3

Optimisation-Based Vertex Smoothing

As it was mentioned in the previous chapter, there are many ways in which vertex smoothing can be performed. The most common algorithm is Laplacian smoothing, which tries to place the vertex under consideration in the geometric center of the cavity which is enclosed by the polygon formed by all adjacent vertices. This approach, although computationally cheap, does not result in an optimal position for the vertex under consideration – in fact quality may even deteriorate – and, even worse, may place the vertex in a location where some elements of the cavity are inverted, which means that they have negative area and the mesh becomes invalid.

[Freitag et al., 1995] have proposed an optimisation-based algorithm for vertex smoothing. This approach still does not guarantee an optimal result, but does not invalidate the mesh and the achieved quality is at least the same as the quality of the initial cavity. The price for these benefits is a much higher computational load. This fact is not necessarily a bad thing, since modern GPUs consist of hundreds of cores and can simultaneously execute thousands of threads. Heavy kernels favour scalability of parallel applications, so optimisation-based vertex smoothing is a perfect candidate for parallel computing.

The basic idea behind an optimisation technique is using functions and their gradients in order to find out where in the solution space these functions are maximised or minimised. Moving towards this direction, we need to express mesh quality in the form of an analytic function of mesh points' coordinates. As it was described earlier, there are many ways in which mesh quality can be measured. [Freitag et al., 1995] have studied the case where we are interested in maximising the minimum element angle, but any other quality metric could be implemented upon the same framework. After presenting the case of maximising the minimum element angle we will present a different approach which tries to give mesh elements a certain orientation, better suited to the problem's anisotropic characteristic.

3.1 Maximising Minimum Angle

Let $\mathbf{p_i} = \{x_i, y_i\}$ be the position vector of mesh vertex v_i. A mesh element t consists of vertices v_i and edges e_i. Let l_i be the length of edge e_i and θ_i the angle between two element edges. This notation can be seen in Figure 3.1. Our quality measure is element angle, therefore it has to be expressed in form of an analytic function. Edge length l_i can be expressed as:

$$l_i = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2}, i = 1, 2, 3, j = 2, 3, 1, k = 3, 1, 2$$

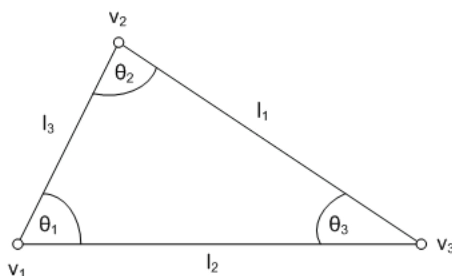


Figure 3.1: Example of a mesh element and the notation used to describe various attributes.

so using the cosine law:

$$\cos\theta_i = \frac{l_i^2 - l_j^2 - l_k^2}{2l_j l_k}$$

Then, the gradient function is:

$$\nabla(\cos\theta_i) = \left(\frac{l_j l_k \left(l_i \frac{\partial l_i}{\partial x_i} - l_j \frac{\partial l_j}{\partial x_i} - l_k \frac{\partial l_k}{\partial x_i} \right) - \frac{1}{2}\left(l_i^2 - l_j^2 - l_k^2 \right)\left(\frac{\partial l_j}{\partial x_i} l_k + l_j \frac{\partial l_k}{\partial x_i} \right)}{(l_j l_k)^2}, \right.$$
$$\left. \frac{l_j l_k \left(l_i \frac{\partial l_i}{\partial y_i} - l_j \frac{\partial l_j}{\partial y_i} - l_k \frac{\partial l_k}{\partial y_i} \right) - \frac{1}{2}\left(l_i^2 - l_j^2 - l_k^2 \right)\left(\frac{\partial l_j}{\partial y_i} l_k + l_j \frac{\partial l_k}{\partial y_i} \right)}{(l_j l_k)^2} \right)$$

Recall that the solution space has to be restricted to the interior convex hull of the cavity formed by the vertex to be smoothed and its adjacent vertices. If we symbolise the cavity as C_i, its interior convex hull as H_i and the set of all angle element inside C_i as A_i then the function we want to maximise is:

$$\phi(\mathbf{p}) = min\left(\theta_j(\mathbf{p})\right), \forall\theta_j \in A_i$$

A projection on the x-field of an example $\phi(\mathbf{p})$ can be seen in Figure 3.2. As it can be seen in that figure, each $\theta_j(\mathbf{p})$ is a smooth and differentiable function. Moving along a certain line of $\theta_j(\mathbf{p})$, there are more than one angles that tend to obtain the minimum value. We say that these angles form the *active set* A. The active set, however, changes at the points denoted by black circles in the figure and at these points $\theta_j(\mathbf{p})$ is non-differentiable. In order to find the maxmin angle we have to solve the non-smooth optimisation problem

$$max(\phi(\mathbf{p})), \mathbf{p} \in H_i$$

This problem has a solution at some point $\mathbf{p_s}$ if all directional derivatives of $\phi(\mathbf{p})$ at $\mathbf{p_s}$ are negative or zero. This solution is unique because all functions $\theta_j(\mathbf{p})$ are monotonic while we move towards a fixed search direction. The non-smooth problem can be solved using a technique similar to the *Gradient Ascent* method. This method, also called *Steepest Ascent*, is used when we deal with twice-differentiable functions. The modified algorithm for non-smooth functions is the one described in Algorithm 5.

Let the original position of the central vertex be $\mathbf{p_{init}}$. At first, the algorithm calculates the interior convex hull of the cavity under consideration and chooses a starting point $\mathbf{p_0}$. If $\mathbf{p_{init}}$ does not coincide with $\mathbf{p_s}$, in which case we already have the optimal location for the central vertex and the algorithm exits, then a first guess for $\mathbf{p_0}$ is the geometric centre $\mathbf{p_c}$ of this hull. If this guess coincides with $\mathbf{p_s}$ then the algorithm stops. Recall that the criterion to determine whether $\mathbf{p_s}$ has been found is that all directional derivatives of $\phi(\mathbf{p})$ at this guess-point are zero or negative. If

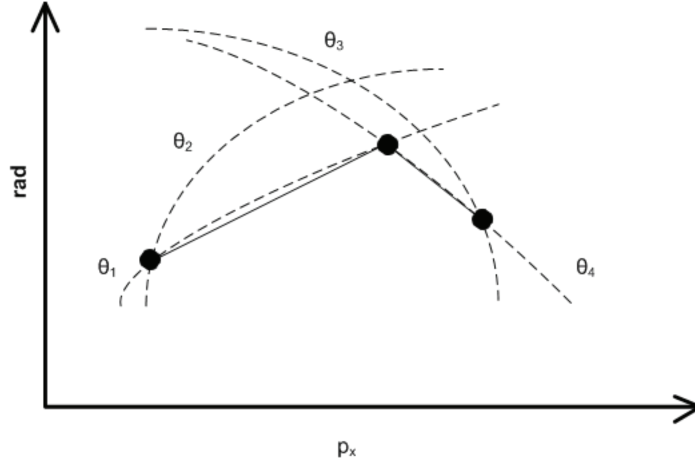


Figure 3.2: Projection of an example $\phi(\mathbf{p})$ on the x-field.

$\mathbf{p_c}$ does not coincide with $\mathbf{p_s}$ then the algorithm decides which point ($\mathbf{p_{init}}, \mathbf{p_c}$) corresponds to a larger value for $\phi(\mathbf{p})$ and sets it as the starting point $\mathbf{p_0}$.

Having found a starting point, the algorithm iteratively tries to find a solution. At every estimation-point $\mathbf{p_i}$ the algorithm calculates all directional gradients. The search direction, i.e. the "overall" steepest direction, is calculated by taking the gradients of all $\theta_j \in A_i$ and finding all possible convex linear combinations of them (the latter implies solving a respective quadratic programming problem). After that, the algorithm has to decide how to move along the search direction, i.e. solve the "line search sub-problem". This task is accomplished by predicting the points at which the active set A_i will change. This prediction can be made by taking the first-order Taylor series approximation of $\theta_j(\mathbf{p}), \forall \theta_j \in A_i$ and calculating the intersection of each such approximation with the projection of $\phi(\mathbf{p})$ in the search direction. The distance between $\mathbf{p_i}$ and the intersection point closest to it is the initial step length. If moving $\mathbf{p_i}$ along the search direction by this length improves $\phi(\mathbf{p})$ by {the estimated improvement $\pm$ some percentage} ([Freitag et al., 1995] propose $\pm 10\%$) then this length is accepted and $\{\mathbf{p_i} + \textit{step_length} \times \textbf{search_direction}\}$ becomes the new estimation-point $\mathbf{p_i}$. Otherwise, the step length is halved again and again until either a suitable length is found or the step length becomes smaller than a user-defined threshold. Additionally, a step is accepted on the condition that the next step improves $\phi(\mathbf{p})$ by a smaller amount.

The process described in the paragraph above is repeated until one of the following conditions are met:

1. $\mathbf{p_i}$ coincides with $\mathbf{p_s}$
2. the step length becomes smaller than the user-defined threshold
3. the improvement of $\phi(\mathbf{p})$ between two successive steps falls below a user-defined threshold
4. the number of iterations exceeds a pre-defined value

3.2 Adjusting Element Area Ratio and Orientation

The maxmin angle criterion is a general attempt to improve element quality. It is based on research results which show that extremely small angles constitute an obstacle towards both convergence

Algorithm 5 The optimisation-based smoothing algorithm.

```
Cavity c = {cavity under consideration};
Function phi = {function phi(p)};

Point pInit = c.getCentralPoint();
Function gradPhi = phi.getGradientFunction();

if(gradPhi.isNonPositiveInAllDirectionsAt(pInit))
    return;

Point p = c.getConvexHull().getGeometricalCentre();
if(gradPhi.isNonPositiveInAllDirectionsAt(p))
{
    c.setCentralPoint(p);
    return;
}

if(c.testMinimumAngleWithCentre(p) <
   c.testMinimumAngleWithCentre(pInit))
    p = pInit;

int iteration = 0;
double step = INF;
while(!gradPhi.isNonPositiveInAllDirectionsAt(p) &&
      iteration < MAX_ITERATIONS && step > MIN_STEP_LENGTH &&
      c.getLastImprovement() > MIN_IMPROVEMENT)
{
    Direction searchDirection = c.findSteepestDirectionAt(p);
    step = p - c.getClosestIntrPointInDirection(searchDirection);

    while((c.testMinAngleImprAt(step, searchDirection) <
           0.9 * c.getEstimatedImpr() ||
           c.testMinAngleImprAt(step, searchDirection) <
           c.testMinAngleImprAt(step / 2, searchDirection))
          && step > MIN_STEP_LENGTH)
    {
        step /= 2;
    }
    if(step > MIN_STEP_LENGTH)
        p += step * searchDirection;

    iteration++;
}

c.setCentralPoint(p);
```

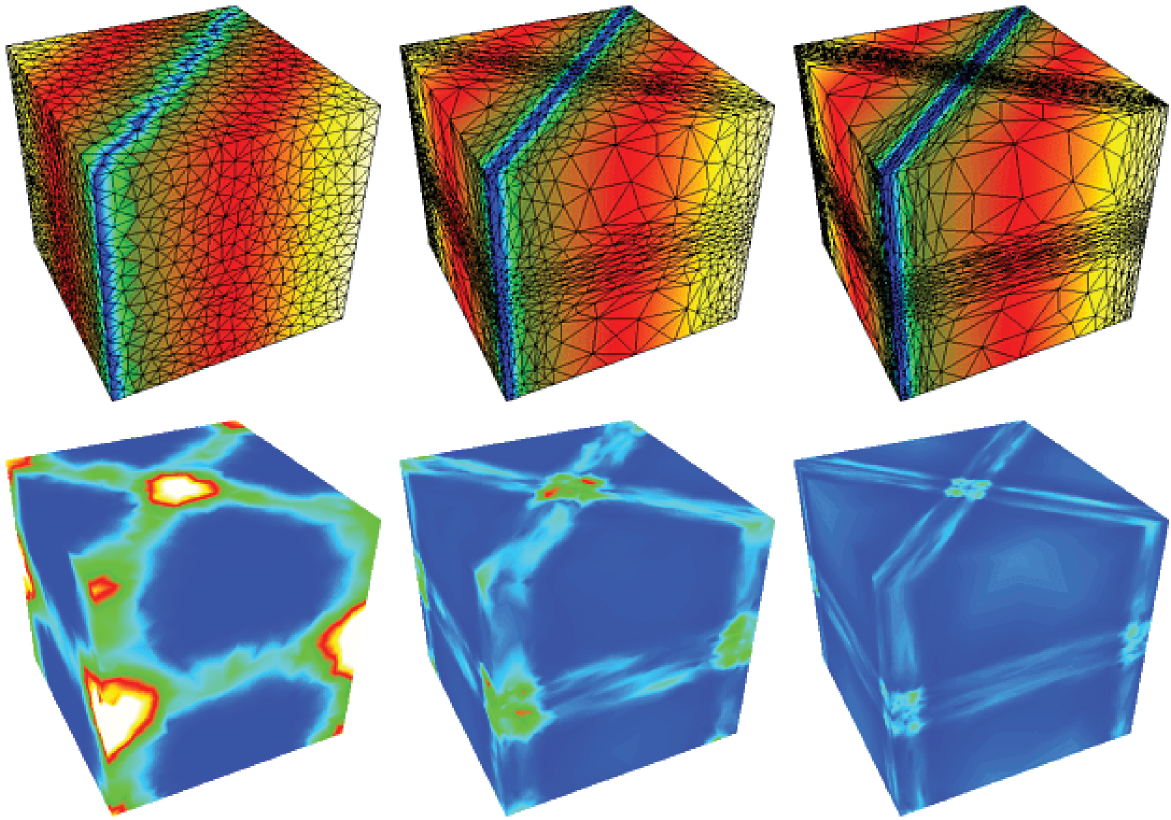


Figure 3.3: Example of mesh adaptation, taking advantage of PDE anisotropy. On the left, the original mesh is completely isotropic. This fact has an impact on the solution error, which is depicted in the bottom left figure. After one refinement iteration (middle figures) the mesh is better adapted to PDE's anisotropy and the solution error is greatly reduced. After two iterations (right figure) the results are even better. (figure from [AMCG, 2006])

speed and solution accuracy. However, it does not take advantage of any information / indication about the solution itself or mesh topology. When a PDE exhibits (more or less intense) anisotropic characteristics, the solution process would benefit if we tried to relocate vertices in such a way that the shape of mesh elements follows this anisotropy. An example of adapting a three-dimensional mesh taking advantage of anisotropic characteristics is shown in Figure 3.3 [AMCG, 2006]. In the same figure, it can be seen how the solution error improves over iterations.

In that example, however, element anisotropy is expressed by adding new elements (for example using the element bisection algorithm) in areas where the solution error is intense and coarsening the mesh (for example using some element merging algorithm) in areas where this error is negligible. In the example above, the initial mesh consists of 45471 elements, the first refinement results in 64389 element mesh and, finally, we end up with a mesh consisting of 91781 elements. In cases like this one, in which more elements are added than removed, the number of mesh elements increases as the refinement procedure goes on. More elements means heavier load, both in terms of memory required for storing all necessary information and time needed to solve the discretised PDE on each of these elements. It would be interesting, therefore, if we could achieve similar results without affecting mesh topology.

One way to do this could be by performing optimisation-based vertex smoothing, attempting to relocate vertices so that elements of a cavity have a specific area ratio and orientation. The idea

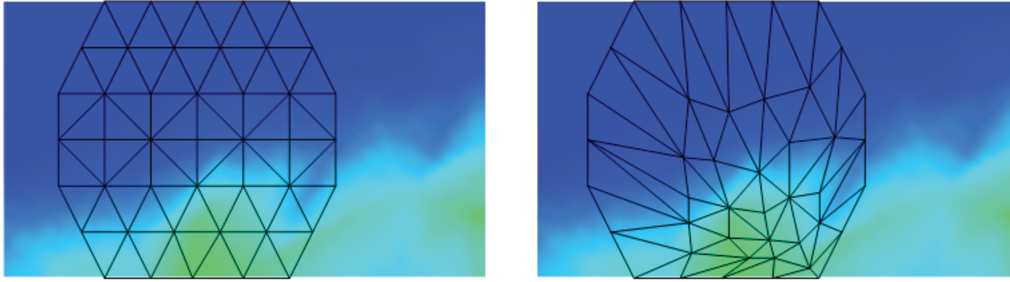


Figure 3.4: Example of adjusting element orientation in order to follow the anisotropic characteristics of a given problem.

behind this procedure is simple. Moving along cavities in the direction in which the solution error increases at the fastest rate, vertices are pulled towards this direction so that gradually element density in erroneous areas increases while elements away from erroneous areas become larger and elongated. An example of this idea is depicted in Figure 3.4. In this example, the solution error using an initial, uniform mesh is negligible in upper-most and left-most areas and increases as we move right/down. By gradually pulling vertices towards that direction, element density increases in green areas, where the error is large, with the hope that following iterations will yield more accurate results.

In a more general case, solution error can be distributed over the whole mesh. The rational question that arises is towards which direction a vertex should be moved. This problem resembles classic problems in gravitational fields or electromagnetism, where there is a field over some surface and we try to find how a free mass or charged particle will move as a result of all elemental forces that occur between the particle and the field vector of every elemental area dA of that surface. The same method could be applied in vertex smoothing, treating the error distribution as a positive electromagnetic field and each vertex as a negatively charged particle. The error value in each element is the equivalent of the magnitude of the electromagnetic field in each elemental area dA and the distance between a vertex and an elemental area dA can be used to make the influence of dA on that vertex fade away as the distance increases. Being inspired by electromagnetic and gravitational problems, we suggest using the inverse square law to apply this fading.

This approach, however, can result in a procedure of high computational complexity. In the general case, the number of elements in which the solution error is non-negligible is $O(m)$ and we have to find the influence of all these elements on $O(n)$ vertices. This would imply an optimisation problem with a complexity of $O(m \times n) \approx O(n^2)$. A more viable approach would be if we transformed the error distribution, producing only a few point-errors (the equivalent of point masses – for example, treat a whole planet as a point mass) which accumulate the error values of all elements in the neighbourhood. In classic physics, an equivalent approach would be "instead of calculating the gravitational force between the Earth and all billions of stars in the universe, group stars into galaxies, treat each galaxy as a point body with mass equal to the sum of masses of all stars comprising it and calculate the gravitational force between the Earth and each galaxy". This approach is shown in Figure 3.5, where the initial error distribution has been accumulated into a few points.

This conversion of error distribution can be done using various methods. The simplest and most common of all is the *Fast Multipole Method*, a method proposed by Vladimir Rokhlin and Leslie Greengard which transforms dense matrices into a $O(n)$ or $O(n \log n)$ element representation, instead of the initial $O(n^2)$ representation [Raykar, 2006]. While there are other, more sophisticated methods which yield better results, FMM has the advantage of being relatively fast. Since

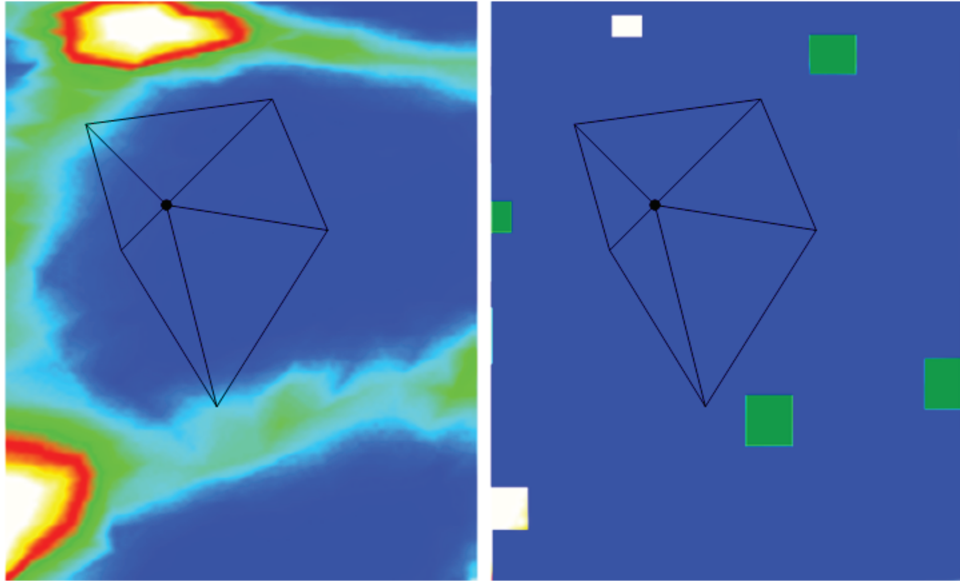


Figure 3.5: Example of an error distribution that is accumulated into a few point-errors.

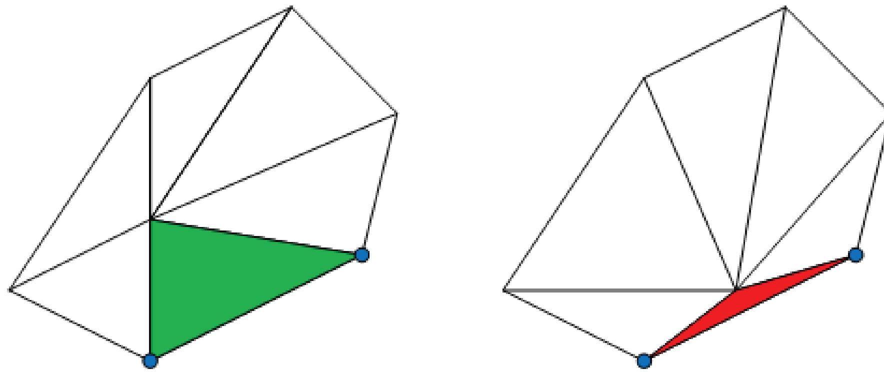


Figure 3.6: Example of an element being excessively compressed, a result of which is that its angles take unwanted values close to either 0° or 180°.

transforming the error distribution is only an auxiliary problem and not the main goal, limited accuracy should not be a problem.

Having found the direction along which a vertex should be moved, we need to find the step length. One obvious restriction is to make sure that the vertex remains inside the interior convex hull of its cavity. This restriction, though, is not enough to ensure mesh quality. As it can be seen in Figure 3.6, the green element is excessively compressed, greatly losing its quality as its angles approach 0° and 180° (the latter being a more severe problem, as it was shown by [Babuška and Aziz, 1976]). An aggressive approach to this problem is to leave the element in its red state with the hope that its angles will regain a better value when the blue vertices are also smoothed and relocated towards the error direction. Another aggressive option is to try and flip the edge defined by the blue vertices. In an average case, doing so will approximately halve the obtuse angle (bringing it closer to 90°, which is just fine) and increase the two small angles.

A more conservative (therefore less risky) approach would be to use the maxmin angle criterion directly, i.e. apply Algorithm 5 restricting the search direction to coincide with the direction computed by FMM. In many cases, however, this solution will leave cavities intact, as moving a

vertex towards a restricted direction will only decrease minimum angles, so Algorithm 5 is very likely to stop as soon as it starts. A more meaningful technique is to use the area ratio between the largest and the smallest element of the cavity and make sure it does not exceed a user-defined limit. This idea is based on the observation that element quality gets worse due to the compression applied to that element and element area is the most obvious metric to measure this compression.

The area of a triangle is given by the cross product of two of its edges. If the triangle is defined by three vertices $A(x_1, y_1, 0), B(x_2, y_2, 0), C(x_3, y_3, 0)$ then its area can be expressed as:

$$
\begin{aligned}
A &= \frac{1}{2}|\vec{AB} \times \vec{AC}| \\
&= \frac{1}{2}|(x_2 - x_1, y_2 - y_1, 0) \times (x_3 - x_1, y_3 - y_1, 0)| \\
&= \frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3)
\end{aligned}
$$

If the largest element in a cavity is defined by two constant vertices $A(x_a, y_a), B(x_b, y_b)$ and shares its third vertex (the centre of the cavity) $P(x, y)$ with the smallest element, also defined by two constant vertices $C(x_c, y_c), D(x_d, y_d)$, then the area ratio function is:

$$
r(\mathbf{p}) = \frac{A_{ABP}}{A_{CDP}} = \frac{x_ay_b - x_by_a + x_by - xy_b + xy_a - x_ay}{x_cy_d - x_dy_c + x_dy - xy_d + xy_c - x_cy}
$$

It should also be noted that, after relocating a vertex, the algorithm must recalculate the area of all cavity elements in order to find which one is the largest and which one is the smallest given the new position of the central vertex. This fact can lead the smoothing algorithm into an infinite loop, as the central vertex can oscillate between two positions that correspond to different {largest element, smallest element}-states. To avoid this case, the user has to define an upper limit for the number of iterations.

Algorithm 6 summarises the whole process. The aim is to bring the central vertex to a position in which $r(\mathbf{p})$ is within a percentage (we suggest 10%) of the pre-defined threshold. At first, the algorithm pushes the central vertex as far as possible, without crossing the interior convex hull boundary. Provided that this initial step was the largest possible and an element's area decreases monotonically as we move along a certain direction, the current value of $r(\mathbf{p})$ is the largest possible. If the ratio is less than $threshold - 10\%$ the algorithm exits, as it will never be able to reach a greater value than this. If the area ratio criterion is not satisfied, i.e. $r(\mathbf{p})$ is larger than the user-defined limit, the algorithm calculates the difference (distance) between the old and new vertex position as well as the difference between the old and new ratio. These two differences are related – we can approximate this relation, assuming it is linear. More precise approximations could be made; for the sake of simplicity, however, we will consider the linear case in this report. The relation can be written as

$$
r_{old} - r_{current} = K(p_{old} - p_{current})
$$

where K is a factor used to linearly relate these differences. By calculating K, the algorithm guesses a new step length $r(\mathbf{p})$

$$
step = p_{new} - p_{current} = \frac{r_{new} - r_{current}}{K}
$$

where always $r_{new} = 0.95 \times threshold$ (aiming at a ratio between $[0.9 \times threshold, threshold]$) and the process is repeated until the ratio is within the desired values.

A final attribute of orientation-based vertex smoothing is its connection with edge flipping, which makes the combination of these two techniques a meaningful choice. As it is shown in Figure 3.4, elements away from erroneous areas tend to become elongated. Due to the nature of orientation-smoothing, a vertex of a triangular element tends to be pulled away from the opposite

Algorithm 6 The orientation-adjusting algorithm.

```
Cavity c = {cavity under consideration};
Point p = c.getCentralPoint(), p_old = c.getCentralPoint();
Direction searchDirection = Mesh.getFMMErrorDistribution().findForceOn(p);
double step = c.getConvexHull().findDistanceFromPoint(p, searchDirection);
p += step * searchDirection;
if(c.getAreaRatioLargestOverSmallest(p) > 0.9 * MAX_AREA_RATIO)
{
    int iteration = 1;
    while((c.getAreaRatioLargestOverSmallest(p) > MAX_AREA_RATIO ||
           c.getAreaRatioLargestOverSmallest(p) < 0.9 * MAX_AREA_RATIO)
         && iteration < MAX_ITERATIONS)
    {
        double k = (c.getAreaRatioLargestOverSmallest(p_old) -
                    c.getAreaRatioLargestOverSmallest(p)) / (p_old - p);
        step = (0.95 * MAX_AREA_RATIO
                - c.getAreaRatioLargestOverSmallest(p)) / k;
        p_old = p;
        p += step * searchDirection;
        iteration++;
    }
}
c.setCentralPoint(p);
```

edge and this results in triangles that tend to have two angles close to $90\,^{\circ}$ and one angle close to $0\,^{\circ}$. If we content ourselves with the minmax angle criterion, discarding maxmin, element elongation should not be a problem, as no angle (of elements away from erroneous areas) will be close to $180\,^{\circ}$. If we still need to take maxmin into consideration, edge flipping is a good way of trying to increase minimum angles. Now, it becomes clear why [Babuška and Aziz, 1976] claimed that the maxmin angle criterion can be restrictive in anisotropic PDEs.

Concluding this chapter, we described two algorithms for optimisation-based vertex smoothing. One technique, proposed and evaluated by [Freitag et al., 1995], is proved to yield far better results than simple Laplacian smoothing, although execution time is greatly increased. On the other hand, driven by the idea to exploit anisotropic characteristics of a given PDE, we took vertex smoothing a step further, proposing a computationally cheap technique which we expect to speed up the mesh improving process. Evaluation of actual improvement is left as future work. Our speculation is that, although orientation adjustment is based on a meaningful metric (element orientation with respect to problem's anisotropy), its potential for improvement could fade out fairly quickly. Nonetheless, it could be used during the early iterations of Algorithm 4 to give an initial, better suited element orientation, followed by adaptive refinement in order to increase element density in erroneous areas thereafter.

Chapter 4

Parallel Execution of Mesh Adaptivity

In this chapter, having presented three common mesh improving algorithms in Chapter 2, we describe the abstract framework proposed by [Freitag et al., 1998] upon which a parallel optimisation application can be built. It is important that the parallel algorithm is *correctly executed*, i.e. parallel execution yields the same results as a serial one. In order to ensure correct parallel execution, we use the concepts of *elemental operations*, one for each mesh improving technique, and the *task graph*, a graph with respect to which these elemental operations have to be synchronised.

4.1 Important graph-structures

In order to define and study the parallel framework we need to define some auxiliary structures/graphs. These include the *vertex graph* G_V, the *element graph* G_T, the *edge graph* G_E and the *2-neighbourhood edge graph* G_E^2. Using the definitions of mesh vertices, elements, edges and edge pairs, these graphs are defined as follows:

- Vertex Graph G_V: The vertex graph G_V is defined as (V, E). In essence, the drawing of G_V coincides with the drawing of the mesh itself, if we ignore the exact shape of mesh elements. An example of G_V for some mesh is shown in Figure 4.1.
- Element Graph G_T: The element graph G_T is defined as (T, D). An example of G_T for the same mesh is shown in Figure 4.2.
- Edge Graph G_E: The vertex graph G_E is defined as (E, F), where F is the set of edge pairs (e_1, e_2) which belong to the same element. An example of G_V is shown in Figure 4.3.
- 2-Neighbourhood Edge Graph G_E^2: The 2-neighbourhood edge graph G_E^2 is defined as (E, F'), where F' is the set of edges f'_i which are either part of the same element as $e \in E$ or part of a directly-neighbouring element; in other words, $f'_i \in F'$ if there is a path in G_E from e to f'_i with length ≤ 2. Drawing of G_E^2 is too complicated to be depicted.

A two-dimensional mesh is then defined as the pair (G_V, G_T). As it was mentioned in an earlier chapter, meshes should be conforming during the solution process, but we allow for non-conformities during the refinement process, which means that G_T can temporarily contain non-conforming elements.

4.2 Correct Parallel Execution

Executing a mesh improving algorithm in a parallel fashion involves manipulation of a distributed data structure. Consistency of this structure has to be retained throughout the whole process and

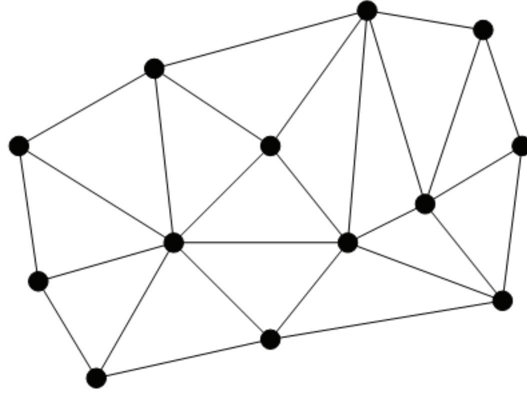


Figure 4.1: Example of a vertex graph G_V. If we ignore the exact shape of mesh elements, the same drawing could depict the mesh itself.

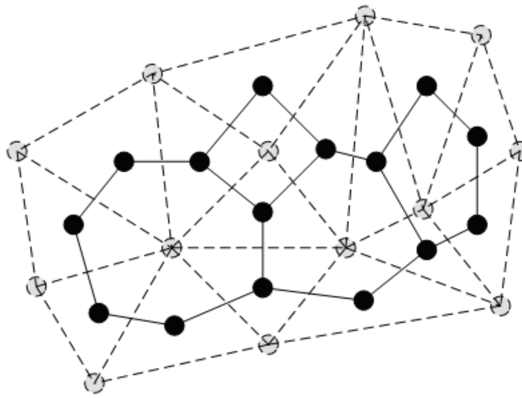


Figure 4.2: Example of the element graph G_T for the same mesh as above.

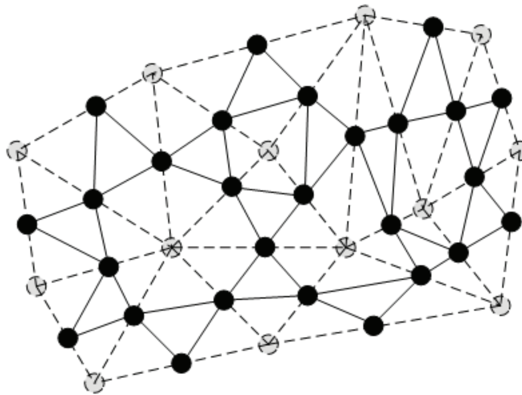


Figure 4.3: Example of the edge graph G_E for the mesh.

the parallel execution of component sub-problems has to yield the same result as some ordering of an equivalent sequential algorithm. If the above two conditions are met then the parallel execution is said to be *correct*.

4.2.1 Elemental Operations

As it is the case with almost every parallel algorithm solving any problem, it is not possible to maintain coherency of data structures at every single point in the parallel execution process. If we were strict on this requirement then the parallel algorithm would be degenerated into a sequential one. A more realistic approach is using the concept of *elemental operations* on distributed data structures and requiring that these structures are coherent only between completions of parallel elemental operations. An *elemental operation* on distributed data consists of the following steps:

1. Execution of a well defined algorithm which may modify the local portion of the distributed data structure in every processing unit

2. A global reduction between all processing units in order to update data that are affected by the above modifications

4.2.2 Data Consistency

The type of data structures depends on the problem under investigation, the algorithm used and the specific implementation of this algorithm. The research group that proposed this parallel framework ([Freitag et al., 1998]), however, believe that following properties must be fulfilled for every type of distributed data structures used:

- Every piece of mesh data (vertices, edges and elements) is owned by a unique processing unit – no two processors can share ownership of the same data.

- Vertex data have to retain their consistency. After any elemental operation, every vertex v has to know which its neighbours are, i.e. $adj(v)$. This knowledge has to be consistent, i.e. a vertex u is neighbour of v if and only if $v \in adj(u)$ in the processing unit owning vertex u. In other words, G_V has to be consistent across all processors. Knowing a neighbour means knowing which vertex it is (for example its global index number) and its position on the mesh.

- Element neighbour data have to retain their consistency. After any elemental operation, every element t has to know which are its neighbours, i.e. $adv(t)$. In other words, G_T has to be consistent across all processors. This knowledge can be used to perform some important operations, for example calculating a quality metric of two neighbouring elements in order to decide whether an edge-flip will improve local quality.

4.3 General Parallel Framework

The mesh refinement process consists of a number of elemental operations. In general, there is no specific ordering of them. It should be noted that each operation acts on a geometric object (vertex, edge, element) in a one-to-one fashion. In order to execute a parallel mesh improving algorithm we need to utilise an important structure called *Task Graph*. In essence, this graph represents all tasks that have to be accomplished throughout the refinement process along with the dependencies between them, therefore helping us extract independent sets of tasks that can be executed in parallel.

4.3.1 Task Graph

Let V be a set of possible operations and E a set of edges. Then, an elemental operation is represented by an element $v \in V$. The set E is constructed as follows: if an operation v_1 affects the input of another operation v_2 then we add an edge (v_1, v_2) to E. We expect that this dependence is bi-directional, i.e. if $(v_1, v_2) \in E$ then $(v_2, v_1) \in E$ as well. Then, the operation task graph is the graph $G = (V, E)$.

4.3.2 Parallel Algorithm

The first step of the parallel algorithm is to find a set $S \subseteq V$ of elemental operations that have to be executed. In order to parallelise execution, we have to find a set of operations that are independent from each other, i.e. we have to colour the task graph and extract an independent set I. After all tasks $v_i \in I$ have been executed, the distributed data structure remains consistent, since these tasks were independent from each other. The algorithm proceeds by choosing new independent sets of elemental operations until S is empty. The algorithm definition be seen in Algorithm 7.

Algorithm 7 General parallel algorithm for the mesh refinement process.

```
TaskGraph G = Problem.createTaskGraph();
Set S = new Set(G.getTasksToBeAccomplished());
while(!S.empty())
{
    Set R = new Set();
    while(!S.empty())
    {
        Set I = new Set(S.getIndependentSet());
        I.executeElementalOperationsOnAllElements();
        I.updateElements(I.getAdjacentElements());
        R.add(I.getAdjacentElements().
            getNonConformingElements().spawnElementalOperations());
    }
    S.setEqualTo(R);
}
```

Every elemental operation may spawn other tasks that have to be accomplished, i.e. it may leave behind non-conformities in neighbouring elements that have to be “fixed”. For example, bisecting an element may create a non-conforming vertex on an adjacent element, which has to be eliminated by subsequent passes through the outer loop. Because of this property, we refer to the outer loop as *propagation loop*, as it propagates tasks to other, non-local parts of the mesh. As far as the inner loop is concerned, the number of iterations performed depends on the task graph and the way independent sets are extracted from it. The nature of the optimisation techniques we have studied implies that these techniques can run asynchronously and mostly require only one-to-one communication between processing units, with only a few global reductions. This characteristic is very important for the efficiency and scalability of a parallel application.

4.4 Elemental Operations

This section describes the elemental operations for each of the three optimisation techniques. The elemental operation for every technique involves manipulation and modification of different data;

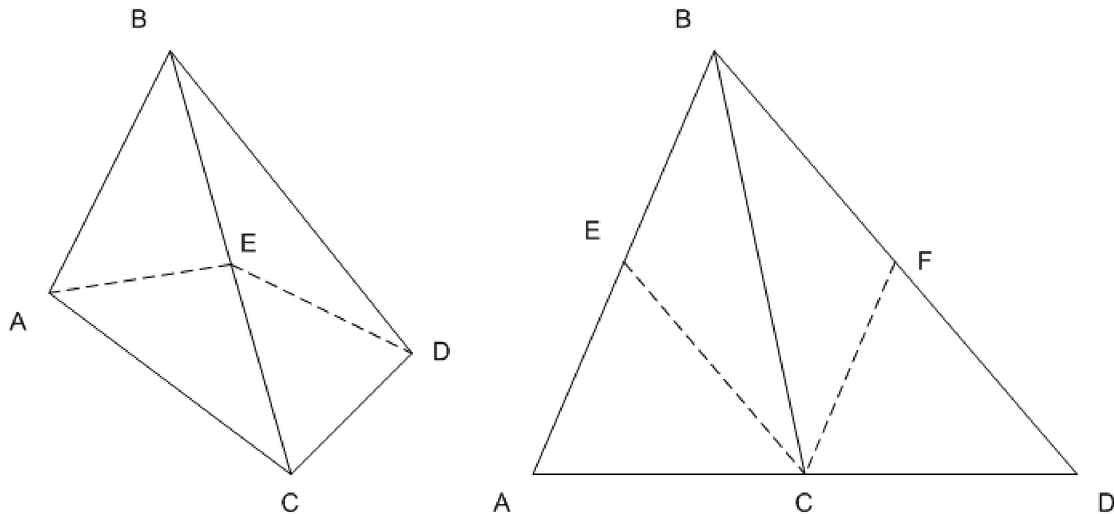


Figure 4.4: Example of problems that can arise if elements are bisected in parallel. In the left figure, bisecting triangles ABC and BCD simultaneously results in the creation of a new vertex E. This vertex, however, is seen as two discrete vertices by triangles ABE-ACE and BDE-CDE, which is wrong and corrupts the distributed data structure. On the right, bisecting trangles ABC and BCD results in the creation of triangles ACE, BCE, BCF and CDF. If these bisections are performed simultaneously, element BCE erroneously knows that its neighbouring element is triangle BCD and element BCF erroneously knows that its neighbouring element is triangle ABC.

therefore, different techniques are executed using different task graphs.

4.4.1 Element Bisection Elemental Operation

When bisection is performed, candidate elements for bisection are probably a set S_B of low-quality triangles. It is not possible to perform bisection to arbitrarily any elements simultaneously. As it can be shown in Figure 4.4, bisecting an element results in the creation of new vertices and elements. This can lead to data corruption. In the first example, bisection of triangle ABC creates a new vertex E in the middle of edge BC. Respectively, bisection of triangle BCD also creates a new vertex E' in the middle of BC. These two new vertices are essentially the same vertex, but they are seen as distinct ones by different elements. In the second example, bisecting triangles ABC and BCD at the same time leads to data corruption in neighbour lists. The new triangle BCE is not aware of BCD bisection, so it does not know that the correct neighbour from now on is the new triangle BCF. Respectively, BCF does not know that its new neighbour is triangle BCE.

In order to overcome these dangers, we have to “lock” neighbouring elements once we have decided to bisect an element. Figure 4.5 depicts what has to be done. Let’s suppose that we decide to bisect triangle ABC, having previously (but at the same refinement level) bisected triangle ACF. The left figure shows the information we need for this bisection, i.e. the shaded element ABC and the black-circled vertices. The right figure shows that the shaded elements must have their element-adjacency lists updated, since mesh topology has changed, and vertices A, B and C must also receive information about their new neighbour (the vertex that was created as a result of bisection).

Algorithm 8 describes the whole process. This algorithm is the modification to Rivara’s original one, proposed by [Freitag et al., 1998]. The original algorithm always chooses to bisect an element across its longest edge. According to the new proposal, we can bisect an element only across an

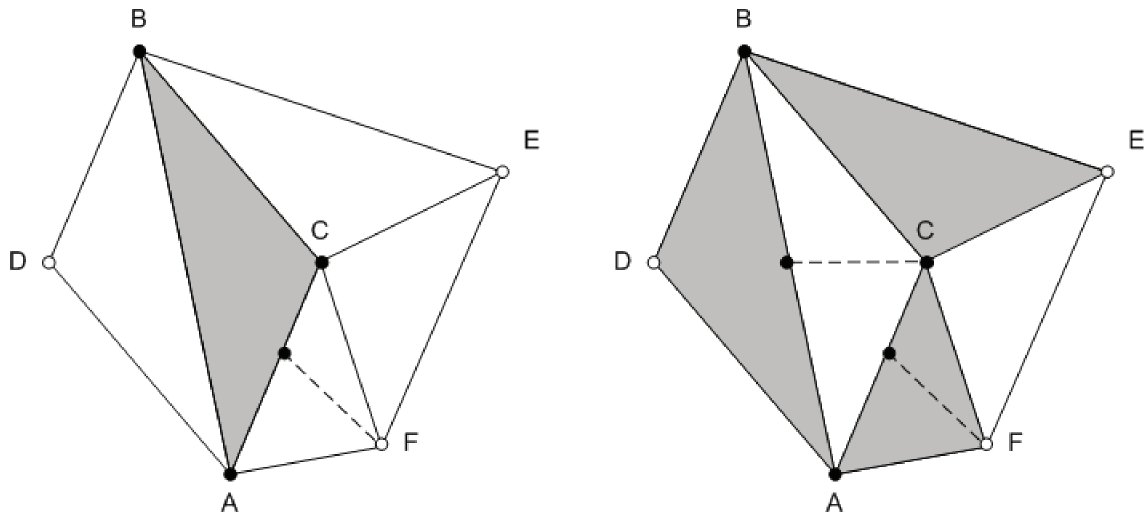


Figure 4.5: Example of how element bisection should be performed. On the left, in order to bisect triangle ABC, neighbours ABD and BCE have to be "locked", i.e. no other operation can be performed on these elements at the same time. On the right, after bisection has taken place, shaded elements must have their neighbour lists updated with the correct data and vertices A, B and C must be informed that the newly created vertex in the middle of edge AB is one of their neighbours.

edge that has not been modified at this refinement level. For example, in Figure 4.5, edge AC had been modified during bisection of triangle ACF and contains a non-conforming vertex. Therefore, it is not a candidate for bisection at this refinement level. This modification has the advantage that an edge is bisected at most once per refinement level, i.e. only one non-conforming vertex can lie on an edge at a time. This property simplifies the algorithm without affecting quality results, according to [Jones and Plassmann, 1997].

Algorithm 8 Algorithm describing bisection elemental operation.

```
Element e = element under consideration;
if(e.hasNonConformingVertex()) //created by adjacent element bisection
    e.bisect(along the longest of the remaining conforming edges);
else
    e.bisect(along longest edge);
e.getAdjacentElements().updateAdjacencyLists();
R.add(e.getNonConformingNeighbour());
```

As it was stated earlier, when an element is bisected all elements surrounding it have to be locked. Therefore, it becomes clear that the operation task graph for element bisection is the element graph G_T; independent sets extracted from this graph correspond to non-neighbouring elements.

4.4.2 Edge Flipping Elemental Operation

Parallel execution of edge flipping has its own restrictions, as it was the case with element bisection. Figure 4.6 shows a potential danger of parallel edge flipping. If edges BC and BD are flipped simultaneously, the flipped edges will cross each other and we will end up with an invalid mesh.

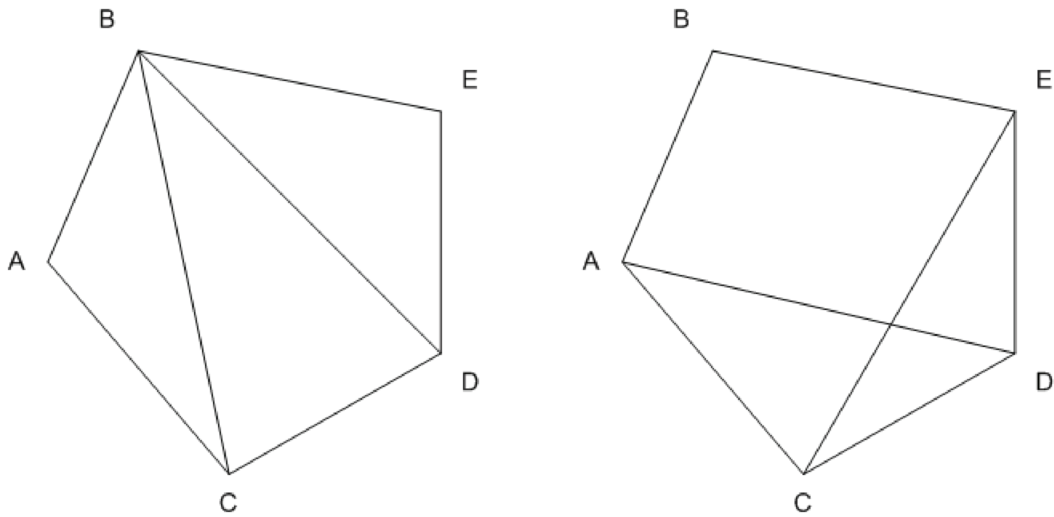


Figure 4.6: Example of how a mesh can be invalidated by simultaneously trying to flip adjacent edges.

At a first glance, an approach to this problem could be “locking” neighbouring edges, i.e. if we decide to flip an edge, all other edges adjacent to it must remain intact. Although this solution avoids invalidating the mesh, it is still dangerous. As it is depicted in Figure 4.7, the neighbouring triangles BCD and CDE have their edges flipped. The resulting mesh is valid; however, there is a corruption of element adjacency data. As it was in the case of bisection, triangle ACD erroneously knows that its neighbour is triangle CDE and triangle CDF erroneously knows that its neighbour is triangle BCD.

The next thought is to lock not only adjacent edges but also adjacent elements. Figure 4.8 depicts what is affected by flipping edge AC. The left figure shows the information we need for this operation, i.e. the shaded elements ABC and ACD and the positions of all black-circled vertices. The right figure shows that the shaded elements and vertices A, B, C, D must have their element-adjacency lists updated, since mesh topology has changed by the edge flip.

Algorithm 9 describes the whole process. It is clear that the operation task graph we need in order to ensure correct parallel execution of edge flipping is the 2-neighbourhood edge graph G_E^2.

Algorithm 9 Algorithm describing edge flipping elemental operation.

```
Edge e = edge under consideration;
ElementPair p = e.getElementsBelongingTo();
Graph GE = edge graph;
Graph GE2 = 2-neighbourhood edge graph;
if(p.qualityIsImproved(e.testFlip()))
{
    e.flip();
    GE2.updateNeighboursOf(e);
    R.add(GE.getAdjacentEdges(e));
}
```

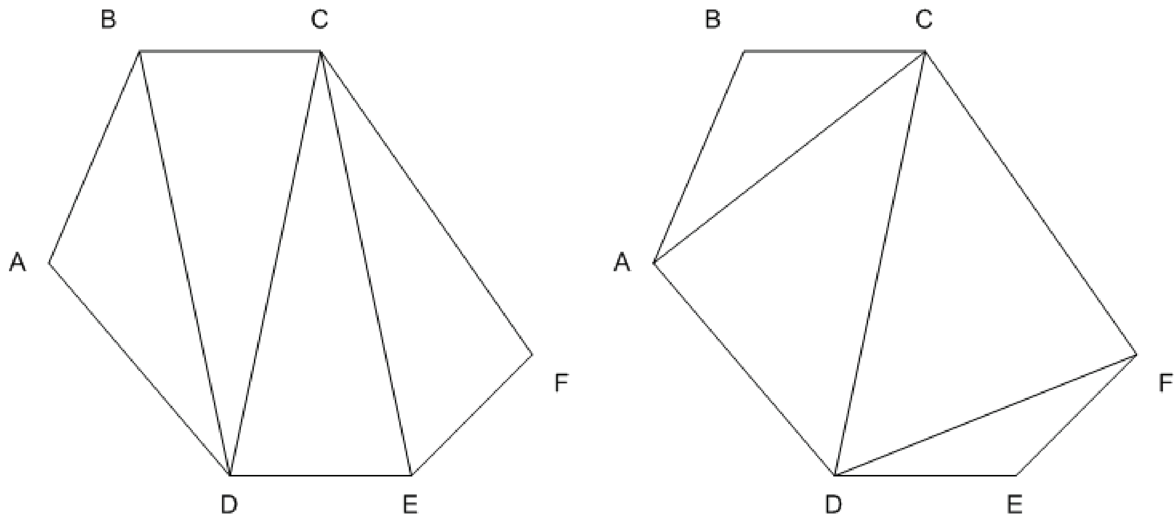


Figure 4.7: Example of adjacency data corruption by simultaneously flipping edges belonging to neighbouring elements. The new triangles ACD and CDF have outdated information about their neighbouring elements.

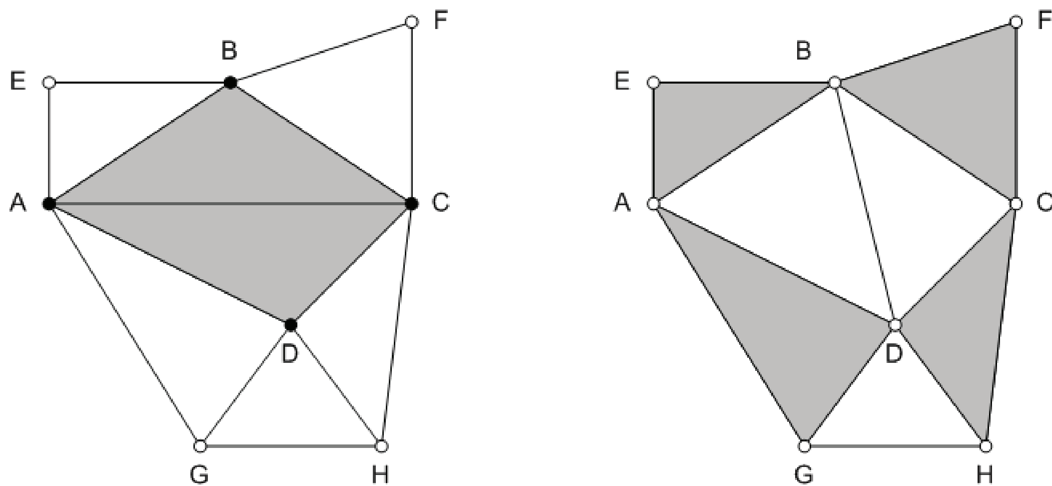


Figure 4.8: Example of how edge flipping should be performed. On the left, in order to flip the edge AC, neighbours ABE, BCF, ADG and CDH have to be "locked", i.e. no other operation can be performed on these elements at the same time. On the right, after flipping the edge, shaded elements must have their neighbour lists updated with the correct data; vertices B and D must be informed that they are now adjacent and vertices A and C must be informed that they are not neighbours any longer.

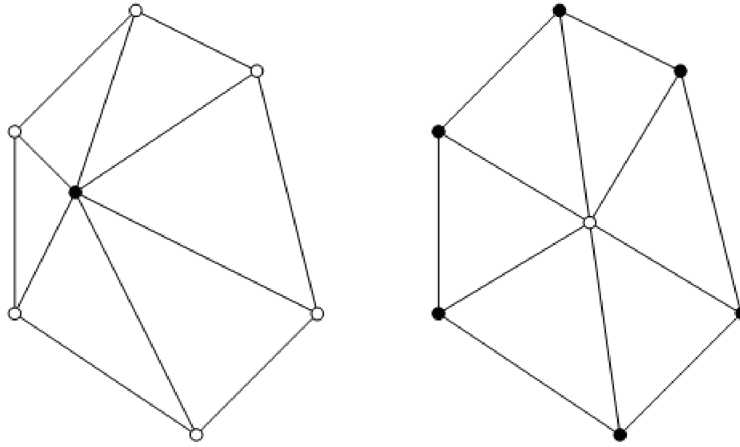


Figure 4.9: Example of how vertex smoothing should be performed. The left figure shows that in order to smooth the vertex denoted by a black circle we need to know the positions of all adjacent vertices (white circles). In the right, figure we see that all adjacent vertices must be updated with the new position of the smoothed one.

4.4.3 Vertex Smoothing Elemental Operation

In order to see how vertex smoothing can be correctly executed in parallel, we have to observe what piece of data is involved in this operation. As it is shown in Figure 4.9, when we smooth a vertex we need to know the exact positions of all its adjacent vertices. Therefore, these vertices have to be "locked". After smoothing, they have to be updated with the new position of the vertex under consideration.

Algorithm 10 describes this process. It is clear that the operation task graph for vertex smoothing is the vertex graph G_V.

Algorithm 10 Algorithm describing vertex smoothing elemental operation.

```
Vertex v = vertex under consideration;
v.smooth();
if(v.positionChanged())
{
    v.getAdjacentVertices().updateWithNewPosition(v.getPosition());
    R.add(v.getAdjacentVertices());
}
```

Recapitulating, this chapter presented a framework upon which a parallel mesh improving application can be built. It was shown that the process of applying a mesh improving algorithm to the whole mesh can be broken down into elemental operations on distributed data. These operations are used to construct the operation task graph, a graph with respect to which the elemental operations have to be consistent. Extracting independent sets of operations out of the graph helps process parallelisation, giving us the ability to assign elemental tasks, independent from each other, to different processing units and performing the optimisation while retaining the coherency of distributed data structures.

Chapter 5

Conclusions and Future Work

This research was conducted with the aim of accumulating the necessary background knowledge for building a parallel mesh improving application in the future. Apart from the literature survey part, study of various algorithms and concepts led us to some interesting conclusions.

Mesh improving algorithms can be grouped in two categories: algorithms which modify mesh topology comprise the h-adaptivity group and algorithms which leave topology intact comprise the r-adaptivity group. Two common techniques of the first category are adaptive refinement and edge flipping. Adaptive refinement has the advantage of increasing mesh density in erroneous areas, a process which leads to a decrease in the solution error. On the other hand, increasing the discretisation degree means that the PDE has to be solved using more elements, which leads directly to an increase in computation time. Edge flipping leaves the number of elements constant and only tries to maximise the minimum element angle. It is a relatively fast technique; however, it only improves element angle and not element density and this fact renders edge flipping less powerful than adaptive refinement.

Vertex smoothing is member of the second category of algorithms. It is a versatile idea in the sense that it can be implemented using a great variety of algorithms, ranging from naïve Laplacian smoothing to optimisation-based smoothing, which tries to improve quality by maximising a relative function. Optimisation-based smoothing is a heavier task but yields better results, as it is performed having a meaningful objective in mind.

Our orientation-adjusting proposal is based on an error-driven objective and not just a quality-driven one, trying to take advantage of anisotropic characteristics. Although we have not evaluated the actual capabilities of this proposal, our intuition is that applying orientation smoothing would be useful during the early iterations of the refinement process, giving an initial, better suited orientation to mesh elements, but the method's potential for solution improvement may fade out quickly thereafter. Edge flipping seems to be a good technique to be combined with orientation smoothing, if we are in pursuit of satisfying the maxmin angle criterion. Additionally, having increased element density in erroneous areas, subsequent adaptive refinement attempts will have to be executed fewer times, keeping the total number of mesh elements as low as possible, therefore saving time during the PDE solution process.

Regarding parallel implementations, the need to ensure correct execution leads to the introduction of elemental operations and task graphs. Task graphs are essentially the representation of these operations along with the way they depend on each other. Extracting independent sets out of a task graph helps us find operations that are independent from each other and can therefore be executed in parallel by different processing units, without distorting coherency of distributed data. In a parallel version of mesh adaptivity, vertex smoothing is simpler to implement as it does not modify the task graph during an adaptation iteration.

Scalability of parallel implementations relies almost exclusively on the ability to extract inde-

pendent sets that are as large as possible. This, in turn, depends on the connectivity Δ of each task graph. [Freitag et al., 1998] have analysed graph connectivity as follows:

- Graph G_V: Denoting the smallest angle in the initial mesh as θ_{min}, adaptive refinement creates elements with minimum angle no smaller than half this value. Edge flipping and maxmin-based vertex smoothing can only increase the minimum angle, so the latter will never fall below $\theta_{min}/2$. This means that the maximum number of triangles sharing a common vertex will always be less than $4\pi/\theta_{min}$.
- Graph G_T: A triangle with no non-conforming vertices has 3 neighbouring triangles. If we allow one non-conforming vertex to lie on each edge, the maximum number of neighbours of a triangle is 6.
- Graph G_E^2: An edge in a conforming mesh can be shared by at most 2 elements. Each of these elements has at most two other edges, so connectivity of G_E is ≤ 4. Each one of these other edges is in turn adjacent to at most two other edges, so connectivity of G_E^2 is ≤ 12.

There are various ways of colouring a graph. There has been some recent research by [Barenboim and Elkin, 2009] who have proposed a distributed graph colouring algorithm that uses $\Delta+1$ colours and has an impressive running time of $O(\Delta) + \frac{1}{2}\log^* n$. For sufficiently large meshes we can get independent sets large enough to fully utilise modern multi-core platforms running thousands of threads, like nVIDIA's CUDA.

There are a few topics which remain open to further investigation. First of all, our new proposal has to be implemented and evaluated, experimenting with parameters such as the user-defined area ratio threshold or the number of FMM aggregation points and determining for how long (for how many iterations) this method can retain its improving capabilities. As far as the CUDA implementation is concerned, knowing that there are limitations regarding data alignment and accessing, it would be necessary to investigate representations of data structures which are "compatible" with this architecture's special characteristics and maximise performance. Finally, an interesting topic would be the three-dimensional version of this research, dealing with tetrahedral mesh elements and appropriately modified mesh improving algorithms.

Bibliography

[AMCG, 2006] AMCG (2006). Simple adapt example. http://amcg.ese.ic.ac.uk/index.php?title=Simple_Adapt_Example.

[AMCG, 2009] AMCG (2009). Fluidity. http://amcg.ese.ic.ac.uk/index.php?title=Fluidity.

[Babuška and Aziz, 1976] Babuška, I. and Aziz, A. K. (1976). On the angle condition in the finite element method. *SIAM Journal on Numerical Analysis*, 13(2):214–226.

[Barenboim and Elkin, 2009] Barenboim, L. and Elkin, M. (2009). Distributed (Δ+1)-coloring in linear (in Δ) time. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 111–120, New York, NY, USA. ACM.

[Farrell, 2009] Farrell, P. E. (2009). *Galerkin projection of discrete fields via supermesh construction*. PhD thesis, Imperial College London.

[Formaggia et al., 2004] Formaggia, L., Micheletti, S., and Perotto, S. (2004). Anisotropic mesh adaptation in computational fluid dynamics: application to the advection-diffusion-reaction and the stokes problems. *Appl. Numer. Math.*, 51(4):511–533.

[Freitag et al., 1995] Freitag, L., Jones, M., and Plassmann, P. (1995). An efficient parallel algorithm for mesh smoothing. In *INTERNATIONAL MESHING ROUNDTABLE*, pages 47–58.

[Freitag, 1997] Freitag, L. A. (1997). On combining laplacian and optimization-based mesh smoothing techniques. In *TRENDS IN UNSTRUCTURED MESH GENERATION*, pages 37–43.

[Freitag et al., 1998] Freitag, L. F., Jones, M. T., and Plassmann, P. E. (1998). The scalability of mesh improvement algorithms. In *IMA VOLUMES IN MATHEMATICS AND ITS APPLICATIONS*, pages 185–212. Springer-Verlag.

[Jones and Plassmann, 1997] Jones, M. T. and Plassmann, P. E. (1997). Parallel algorithms for adaptive mesh refinement. *SIAM J. Sci. Comput.*, 18(3):686–708.

[Kulkarni et al., 2009] Kulkarni, M., Burtscher, M., Cascaval, C., and Pingali, K. (2009). Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76. IEEE.

[Labelle, 1999] Labelle, F. (1999). Anisotropic triangular mesh generation based on refinement. http://www.eecs.berkeley.edu/ flab/cs294-5/project2/mesh.html.

[Piggott et al., 2009] Piggott, M. D., Farrell, P. E., Wilson, C. R., Gorman, G. J., and Pain, C. C. (2009). Anisotropic mesh adaptivity for multi-scale ocean modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4591–4611.

[Raykar, 2006] Raykar, V. C. (2006). A short primer on the fast multipole method.

[Rivara, 1984] Rivara, M.-C. (1984). Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613.