UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Multi scale preconditioner to solve elliptic problems with high-order methods on non conforming meshes with p4est

Dissertation presented by
David WEICKER

for obtaining the Master's degree in
Mathematical Engineering

Supervisor(s)
Jean-François REMACLE, Michael HANKE

Reader(s)
Philippe CHATELAIN

Academic year 2016-2017

Acknowledgments

Let me first thank professors Jean-François Remacle and Michael Hanke to have allowed me to work on the subject presented here. The guidance they provided throughout the definition of the algorithms, the implementation and the redaction was really appreciated. Their knowledge of the field also really helped me navigate the immense literature that is available about the subject at hand.

They say the devil is in the details and this thesis has had a lot of devils in it. Many thanks to Antoine David, Joachim L'Hoir and Robin Parez for their thoughtful recommendations about several aspects of the work that follows. The useful discussions we had about my work helped me to shed some light on a lot of devils.

I would also like to thank my close friends and family, who were always available for some engineering talk on both rainy and sunny days.

Finally, I would like to thanks Charlotte Jallet for her incommensurable and unwavering support until the completion of this thesis.

Contents

List of Abbreviations

AMG ..... Algebraic Multigrid

AMR ..... Adaptive Mesh Refinement

BDD ..... Balancing Domain Decomposition

CG ..... Conjugate Gradients

DD ..... Domain Decomposition

DOF ..... Degrees of Freedom

FEM ..... Finite Element Method

FFT ..... Fast Fourier Transform

FMM ..... Fast Multipole Method

GLL ..... Gauss-Lobatto-Legendre

GMG ..... Geometric Multigrid

MG ..... Multigrid

OAS ..... Overlapping Additive Schwarz

OMS ..... Overlapping Multiplicative Scharz

PCG ..... Preconditioned Conjugate Gradients

SEM ..... Spectral Element Method

Introduction

Elliptic problems are found in a wide range of different domains. We can find an application for them in astrophysics, chemistry, mechanics, electromagnetics, statistics, image processing, just to name a few. The most common elliptic problem is Poisson's equation with constant coefficients. It is usually stated as :

$$\nabla^2 u = f$$

Where u is a smooth function we want to solve for and f is a known function and is called the source term or the forcing term. Suitable boundary conditions also have to be provided to guarantee the unicity of the solution. Other formulations might include non constant coefficients. Poisson's equation encapsulates many of the problems we can encounter while solving elliptic partial differential equations and that is why it is often used to develop and test algorithms attempting to solve elliptic problems.

In addition to being present in a lot of domains, Poisson's equation can also be used as a building block for other numerical methods. An example of this is the numerical solution for incompressible flows. Indeed, most numerical methods solving Navier-Stoke equations for incompressible flows require to solve Poisson's equation at each time step (to project the velocity field onto the divergence-free space for example).

All those considerations lead to the need of fast and robust algorithms to compute the solution of Poisson's equation. Those algorithms are often called *Poisson solvers*. For simple geometries and regular grids, there exists well-known direct solvers, usually based on the Fast Fourier Transform, that are well suited to the task. However, if we relax one of those restrictions, it is no longer possible to use such solvers. Because most practical problems involve complex computational domains, or a function f that is very localized or has different scales of resolution (and therefore the use of a regular grid is no longer appropriate), or both, there has been a lot of work done into developing other approaches.

Let us also mention that the increase of the accuracy requirements leads to the growth of the typical problem sizes. The methods developed have to have an efficient memory management and a good scalability to a lot of degrees of freedom. Several research groups are working on algorithms that have to be able to scale well up to trillions of unknowns. That is why high-order methods are often used in practice.

Because the number of unknowns is so high, parallel computing has become a necessity. Some computations are performed on millions of CPU cores. That is why any algorithm has to be developed while keeping in mind that it must be able to scale well to such a number of processors.

There exists several algorithms that meet the requirement presented above. The work of this thesis is to present one of such algorithms and the theory that supports its choice. The chosen algorithm has then been implemented, from scratch, in order to test its performances and compare them with the theoretical results. That resulted in a code written in plain C of more than 8000 lines, leveraging the *p4est* library to handle the adaptive mesh and the *Lapack* library to perform some linear algebra (such as the diagonalization of a matrix). Some technical details of the implementation are also presented.

The chosen algorithm is the spectral element method on an non conforming mesh (that

arises from adaptive refinement) consisting of quadrilateral elements. To solve the linear system obtained with the discretization, the preconditioned conjugate gradients are used with a two-scale preconditioner. The first part of the preconditioner, called the coarse scale, consists of a geometric multigrid method. The second part, called the fine scale, is constituted by an overlapping additive Schwarz preconditioner. Currently, the implementation only supports two dimensional problems and has not been parallelized. However, let us stress the fact that the objective is to verify important properties of the algorithm used.

The work presented here targets specifically the following objectives :

- Investigate the properties of the chosen algorithm and their theoretical basis.
- Implement, from scratch, the algorithm to obtain an efficient two dimensional *Poisson solver* that handles non conforming meshes and uses a high-order method.
- Discuss the performances of the *Poisson solver* and compare them with the theoretical results.

For the remaining of this introduction chapter, let us present the different parts of the numerical scheme that has been chosen as well as the reasons for using those parts. We will start by the type of mesh used, then move on to the spectral element method. We will afterwards give a brief overview of the preconditioned conjugate gradients and the two preconditioners. The end of this chapter gives a general outline of this thesis.

Non conforming meshes and p4est

Let us start by saying that the meshes used in our applications contain only quadrilateral elements. Such meshes are usually considered to be better than triangular meshes. Examples of the reasons for that are given in [1] or in [2]. Meshes composed by quadrilaterals also contain fewer elements than those composed by triangles for the same number of vertices. Therefore, those meshes require less memory and the assembly procedure is faster. That is especially true for high-order spectral elements.

Another advantage of quadrilateral meshes is that we can define the basis functions (of the spectral element method) as tensor-products of the one dimensional shape functions. We will see that this property allows us to perform a matrix-vector product very efficiently.

We also want to be able to handle in practice both case when f is a superposition of oscillatory modes (where a regular grid is the best choice) and when f has very localized features (where the grid should be able to capture the local features while the number of degrees of freedom is not overwhelming). In practice, in order to have an accurate solution, a high resolution is needed in some parts of the computational domain while in large other parts, high levels of refinement is not necessary. That naturally leads to adaptive mesh refinement (AMR) and therefore to non conforming meshes. Examples of the different ways to define the criterion used to decide if a given quadrilateral must be refined are given in [3].

A non conforming mesh is one that contains hanging nodes. A hanging node can be defined by a node that is not shared by all its neighboring elements. They are inevitable when using tree-based AMR methods and must be treated carefully. Examples of meshes containing hanging nodes can be found later in this thesis.

The library used here to generate and handle the mesh is *p4est*. Its founding paper is [4]. It bases its AMR process on a forest of quadtrees (it is therefore a tree-basd AMR). Each quadrant can be refined in an adaptive manner using a user-supplied criterion. The inherent structure of the meshes produced using this forest of quadtrees allows us to easily define an hierarchy of grids for the geometric multigrid method.

The spectral element method

Let us now turn to the discretization method. The spectral element method consists of approximating the solution u with continuous piece-wise polynomials of degree p. It is the fact that the approximation is continuous that forces us to take special care of hanging nodes.

More precisely, on each quadrant, the interpolation bases consist of a tensor product of one dimensional shape functions (as explained in [5]). Those shape functions are the Lagrangian polynomials associated with the Gauss-Lobatto-Legendre (GLL) quadrature points.

As stated above, we will use a high-order method, i.e. a large degree p. This is because high-order interpolations can better approximate a smooth function than piece-wise (bi-)linear ones. Let us think about a one dimensional example to grasp this intuitively. With three points, an interpolation of half a period of a sine wave using a quadratic function will be better than a continuous piece-wise linear interpolation.

This intuition has been formalized into a really important result. If the forcing term is infinitely smooth, we have the *spectral convergence* property : the discretization error decays faster than any power law for sufficiently large p. That means that we have an exponential convergence of our solution. Our course, that requires that f is smooth enough. This result is for example given in [6] or in [7].

That is why it is often a good idea to use high-order methods. It allows us to obtain a very high accuracy while keeping the number of degrees of freedom reasonable. In practice, high-order methods almost always perform better than their low-order counterparts when the accuracy requirements are high.

Our implementation supports any degree of interpolation p. However, in the results chapter, we only present cases for $p = 2, 4, 6, 8$.

The preconditioned conjugate gradients

From the spectral element method arises a large system of linear equations. Direct methods are impossible due to its size and therefore we favor an iterative method, namely the conjugate gradients. Since it is rather slow because the matrix defining the linear system (let us call it A) is badly conditioned, we use a preconditioner.

This conditioner is composed by two parts and we want one part to act on a large scale and the other to act on a fine scale. Our hope is that the sum of the two parts of the preconditioner is a good approximation of A^{-1} so that the preconditioned conjugate gradients converge in only a few iterations. That is why we call it a two-scale preconditioner.

The first part of the preconditioner, called the coarse preconditioner, consists of solving a low-order problem on the mesh. To solve this coarse problem, we will use a geometric multigrid solver. It is one of the fastest method available. This algorithm has multiple advantages but the main one is called the h-independent convergence. The number of iterations of the geometric multigrid method is independent of the number of quadrant. As we will see in the results chapter, it is this part of the preconditioner that allows us to have a constant number of iterations whatever the level of refinement in our mesh. The hierarchy of meshes needed for the geometric multigrid method derives naturally from the tree-based AMR process : the mesh at a given level k contains only quadrants that have a level of refinement inferior or equal to k.

The second part of the preconditioner, called the fine preconditioner, is constituted by an ovarlapping additive Schwarz preconditioner. It is a domain decomposition method. The principle is to solve the problem on several subdomains with Dirichlet boundary conditions and then to sum the contributions of the different subdomains. We will see that our quadrilateral mesh provides us with an easy way to define those subdomains. We will also make some assumptions that will allow us to compute the fine preconditioner efficiently.

Outline of the thesis

Let us end this introduction chapter with an outline of the contents of this thesis.

Poisson solvers are intensively studied and the algorithm for this thesis is not the only one available. The first chapter therefore presents several other algorithms to solve elliptic problems and tries to give a snapshot of the different methods used in practice.

The second chapter refocuses on the chosen algorithm. In it, we present in detail the different methods that constitute our *Poisson solver*. The theoretical reasons behind the choices made are also laid out.

As a large part of this thesis was the implementation, with *p4est*, of the method presented in chapter 2, the third chapter spends some time explaining the different design choices that were made and how the algorithm is implemented in practice.

The fourth chapter puts the implementation to the test. We test the algorithm against several examples and quantify its performances. The number of iterations needed to reach a given tolerance is compared for the different cases and we compare the observations with the theoretical results.

We end this thesis by a conclusion that summarizes all the work done and the important results. We also check the fulfillment of the objectives presented earlier in this chapter.

Chapter 1

State of the art

Since the resolution of elliptic problems is a building block for a lot of different numerical methods, the amount of work put into efficient solvers has been huge, both in the academical and the industrial domains. This means that the available literature on the subject is equally massive. This chapter attempts to give the reader a snapshot of the different methods used in practice to solve Poisson's equation with trillions of unknowns.

Let us first note that it is impossible to define the most efficient method since it depends a lot on the geometry of the computational domain and on the source term (i.e. the right-hand side in Poisson's equation). Therefore, we will show different algorithms and explain what their strengths and weaknesses are.

Let us also note that, for all the methods presented here, the high-order version of the method is almost always a better choice than the low-order counterpart if we want to reach a given accuracy. As explained in the introduction chapter, this is because high order methods have a better accuracy for an equivalent number of unknowns. The only exception to this rule is if we only need a rather low accuracy in the solution.

We can divide the solvers into two large categories : basic solvers that resulted from earlier work and high-end solvers that are the implementation of more modern, more efficient methods. We will see that in some particular cases, the basic solvers still outperform the second category of solvers.

Let us start with the earlier methods. Those include the direct sparse solvers, which are especially effective in one dimension since then the stiffness matrix is banded and the linear system resulting from the discretization can be solved in a time complexity that is linear in the number of degrees of freedom (as showed in [8]). In two (or more) dimensions, this method is less effective since the band increases and depends a lot on the numbering of the unknowns. This is particularly true for an unstructured mesh.

Another early method that has to be mentioned is the Fast Fourier Transform (FFT). Direct solvers based on the FFT are extremely fast even in two or three dimensions. The drawback is that the grid often needs to be regular (even though if some work has shown that it is possible to compute efficiently the FFT for non uniform spacing, for more information see [9]). In any case, this method does not work for general unstructured meshes. Let us nonetheless note that if the source term is smooth and we have need of an uniform resolution, solvers based on the FFT often outperform by a very large margin any other method.

The last group of methods in the basic solvers consists of the iterative methods, the most famous of which are the conjugate gradients. This method is very general since it can handle unstructured, non conforming meshes in any dimension but, used on its own, it is not very efficient. This problem can be overcome by the addition of a preconditioner. A lot of different preconditioning techniques have been developed with more or less success. The idea is always to reduce the condition number of the stiffness matrix. The performances of the preconditioned conjugate gradients (PCG) depend a lot on the choice of the preconditioner.

The second category of solvers were developed because the size of the problems became too large for classical iterative methods. It was now needed to have an accurate solution even for systems with trillions of unknowns on very general geometries. We therefore look at methods that can scale to that number of unknowns.

The second reason is that the solutions of the problems often have very different behavior in different parts of the computational domain. Using an uniform resolution is therefore a lot less efficient that using an adaptive mesh. The second category methods therefore also need to handle non conforming meshes.

A first example of those methods is the Domain Decomposition (DD), where one split the computational domain into several subdomains. The subdomains can be overlapping, such as in the overlapping additive Schwarz method (OAS) or its multiplicative counterpart (OMS). The subdomains can also be non-overlapping, such as in the balancing domain decomposition method (BDD). The domain decomposition methods can be used by themselves (for example if we can identify two regions where we can use regular grids and the FFT) but are more typically used as preconditioners for the PCG, as it is the case in our application. We can also mention that a coarse problem where we have only a few unknowns by subdomains is very often used to coordinate the solution globally.

Another very important method is the Geometric Multigrid Method (GMG), where we define an hierarchy of grids and where we solve recursively the problem on coarser and coarser meshes. Those methods have proven very efficient and have been generalized to high-order methods on general unstructured, non conforming meshes. The major difficulty is to define the hierarchy. That is why it is often implemented with an AMR that uses a tree-based method in its refinement process. Even if the geometric multigrid method was initially built for low order discretizations, it has successfully been generalized to higher-order methods (see for example [10]). We can then use a GMG solver with high order on its own or use a GMG solver with a low order as a solver for the coarse problem used in domain decomposition. The latter approach is the one implemented in this thesis.

A close cousin to GMG is the Algebraic Multigrid Method (AMG) which use the same principles but directly applied to the fine scale matrix. The principal advantage over GMG is that there is no need of a hierarchy of meshes and therefore the method is more flexible (as exemplified in [11] or [12]). The downside is that it often requires more memory and it is generally less efficient than the geometric multigrid method.

A last method we can mention is Fast Multipole Method. It was initially developed to solve the particle N-body problem but it has been successfully adapted to the Poisson's equation (see [13] and [14] for information). In practice, this method has proven as efficient as GMG.

In addition of the different restrictions presented above, the algorithms also need to be highly parallelizable. Indeed, we want to be able to use millions (or even billions) of CPU cores. Thus, only methods that have a good scalability performs well on such systems.

A comparative study of the efficiency of the different methods presented here has been realized in paper [15]. They compare state of the art solvers implementing FFT, FMM, GMG and AMG on different problems. Several conclusions are worth mentioning here. For uniform grids, FFT outperforms all other methods by a very large margin. However, for solutions that have very local features, FFT loses to methods that can use adaptive refinement such as GMG and FMM. That is because those methods can use several orders of magnitude fewer unknowns than FFT. Let us also mention that low order variants of any method are always significantly slower than their high order counterparts when the accuracy requirements are high.

We can end this chapter by saying that no method rules the others. Each method has a particular case for which they are better suited than everybody else. A robust solver should be a hybrid method to handle both cases when the solution is a superposition of oscillatory modes and when we have a highly localized forcing term.

Chapter 2

Theory

In this chapter, we present the theory behind the different algorithms used. As mentioned in the introduction, the objective is to solve Poisson's equation as quickly as possible. The particularities of hanging nodes are also an important part of this chapter.

The first section formalizes the mathematical problem to solve and derives the weak formulation of Poisson's equation that will be used for the discretization.

The second section presents the spectral element method and how it applies on non conforming meshes. Both the mesh and the discretization of the weak formulation are explained.

We will then turn to the iterative method used to solve the linear system of equations arising from the discretization : the preconditioned conjugate gradients.

The last two sections explain in some details the two parts of our preconditioner. The coarse part is constituted by a geometric multigrid solver while the fine scale preconditioner consists of an overlapping additive Schwarz method.

2.1 Presentation of the problem

In this section, we will present the problem we will solve. Let us consider a domain $\Omega \in \mathbb{R}^2$ with its boundary $\Gamma = \partial\Omega$. Then, we want to find a function $u : \Omega \to \mathbb{R}$ that satisfies :

$$\nabla^2 u = f \qquad \text{on } \Omega \tag{2.1}$$

$$u = u_0 \qquad \text{on } \Gamma \tag{2.2}$$

Where $f : \Omega \to \mathbb{R}$ and $u_0 : \Gamma \to \mathbb{R}$ are two given functions. We also assume that f and u_0 satisfy the standard regularity assumptions. Let us denote $H^1(\Omega)$, the Sobolev space containing functions whose partial derivatives up to order 1 are in $L^2(\Omega)$. Let us also define $H_0^1(\Omega)$, a subspace of $H^1(\Omega)$ where the functions are equal to zero on Γ.

$$H_0^1(\Omega) = \{v \in H^1(\Omega), v|_\Gamma = 0\}$$

Let us then multiply equation 2.1 by any function v in $H_0^1(\Omega)$ and integrate it over the domain. And since the laplacian is equivalent to the divergence of the gradient, we have that :

$$\int_\Omega \nabla \cdot (\nabla u) v \, dxdy = \int_\Omega f v \, dxdy \tag{2.3}$$

Any function u that is a solution of 2.1 is obviously also a solution of 2.3 for every function $v \in H_0^1(\Omega)$. Let us now recall that for any functions u and v regular enough :

$$\nabla \cdot (v\nabla u) = \nabla u \cdot \nabla v + v\nabla \cdot (\nabla u)$$

Using this relation on equation 2.3, we get :

$$\int_\Omega \nabla u \cdot \nabla v \, dxdy = -\int_\Omega \nabla \cdot (v\nabla u) dxdy - \int_\Omega fv \, dxdy \tag{2.4}$$

Finally, using the divergence theorem, we have :

$$\int_\Omega \nabla u \cdot \nabla v \, dxdy = -\int_\Gamma (v\nabla u) \cdot \mathbf{n} \, ds - \int_\Omega fv \, dxdy \tag{2.5}$$

Since $v \in H_0^1(\Omega)$, it is equal to zero on the boundary and therefore :

$$\int_\Omega \nabla u \cdot \nabla v \, dxdy = -\int_\Omega fv \, dxdy \tag{2.6}$$

We can now state the weak formulation of problem 2.1. Let us assume, without loss of generality that $u_0 = 0$. Then, we want to find a function $u \in H_0^1(\Omega)$ that satisfies :

$$\int_\Omega \nabla u \cdot \nabla v \, dxdy = -\int_\Omega fv \, dxdy \qquad \forall v \in H_0^1(\Omega) \tag{2.7}$$

2.2 Spectral element method on non conforming meshes

Equation 2.7 is the problem we will be attempting to solve. However, the equation must still hold for every function $v \in H_0^1(\Omega)$. This section presents the method used for the discretization of the problem, namely the spectral element method on a non conforming mesh.

The first part of the task is the discretization of the domain Ω. As explained in the introduction, we will use non conforming meshes. So we will present such meshes, present their particularities and explain how we will handle them.

The second part of the task is the discretization of equation 2.7. The method here presented is the spectral element method, which consists of using high degree piece-wise polynomials as basis functions. We will first show how to obtain the linear system of equation ($Au = b$) that arises with this method. We will afterwards present the one dimensional basis functions used and the location of the nodes on the 1D reference element. We will then explain how to construct the global basis functions in the presence of hanging nodes. We will finally show how to compute the right hand side of the linear system and how to perform an efficient matrix-vector product without having to explicitly construct the matrix A.

2.2.1 Discretization of the domain

Let us first tackle the discretization of the domain. In classical AMR, the mesh depends on the problem to solve and the refinement is performed to obtain the fact that the error committed on each quadrant is roughly the same (see [16]). But that is not the focus of the work done here. Instead, let us assume that we already have a mesh G consisting of unstructured quadrilaterals (denoted Ω_e) and that might be non conforming, i.e. where we have the presence of hanging nodes. As always, we have :

$$G = \bigcup_e \Omega_e$$

$$\Omega_i \cap \Omega_j = \emptyset \qquad \text{if } i \neq j$$

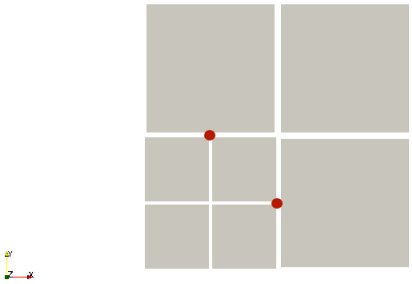


Figure 2.1: Example of a legal non conforming mesh (it is 1-irregular). We can see that we have two hanging nodes in red since they are not vertices of the bigger quadrants neighboring them.

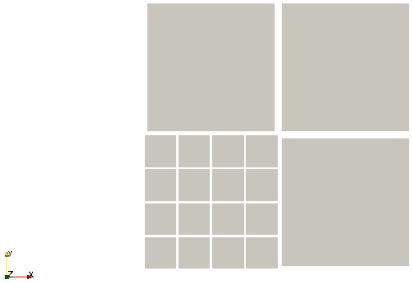


Figure 2.2: Example of an illegal non conforming mesh (it is 2-irregular). We can see that the top left quadrant has four neighbors through its south edge while we imposed a maximum of two.

We can define a hanging node as being a node that is not shared between all its neighboring quadrants. We can note that hanging nodes can only be found on edges. Figure 2.1 shows an example of a mesh containing hanging nodes. We can see that the vertices in red are hanging nodes since they are vertices of the smaller quadrants neighboring them but not of the bigger quadrants neighboring them. Those nodes will require a special treatment in the spectral element method.

We have however one restriction to make on the mesh. We will allow two neighboring quadrants to have only one level of difference in the refinement. This means that the number of neighbors of a quadrant through a given edge is maximum two. We call such meshes 1-irregular. Figure 2.2 shows an example of an illegal non conforming mesh. We can see that on that mesh the top left quadrant has four neighbors through its south edge. Since the maximum difference of the levels of refinement between two neighboring quadrants is two, such a mesh is called 2-irregular. Handling all possibilities of hanging faces for a general k-irregular mesh (with $k > 1$) is impossible and that is why we (and the vast majority of others) consider as illegal meshes such as the one presented in figure 2.2.

For the rest of this chapter, it is important to note that, in a 1-irregular mesh, a hanging edge (an edge containing hanging nodes) is always half of a non hanging edge for its neighboring quadrant.

Once we have our mesh, it is clear that we can compute the integral of any scalar function g as the sum of integrals on the quadrants. Formally, we have :

$$\int_\Omega g\, dxdy = \sum_e \int_{\Omega_e} g\, dxdy \tag{2.8}$$

All the integrals we compute will be performed quadrant by quadrant and then summed.

2.2.2 Discretization of equation 2.7

Since equation 2.7 must hold for every function $v \in H_0^1(\Omega)$, it is not very practical. We will therefore restrict ourselves to $V^h \subset H_0^1(\Omega)$, where V^h has a finite dimension. Let us denote the functions that form the basis of V^h by $\phi_1, ..., \phi_N$ (therefore, the space V^h is of dimension N). We will define explicitly those basis functions later.

Using Galerkin approximation, we will also restrict our search of the solution to V^h. Let us denote this solution u^h. Therefore, we have that :

$$u^h = \sum_{j=1}^{N} u_j \phi_j$$

Asking equation 2.7 to be satisfied for every function in V^h is equivalent to ask it to be satisfied for every function of its basis. As a result, the discretized version of equation 2.7 is given by :

$$\sum_{j=1}^{N} \int_\Omega \nabla\phi_i \cdot \nabla\phi_j\, dxdy\, u_j = -\int_\Omega f\phi_i\, dxdy \qquad \text{for } i = 1, ..., N \tag{2.9}$$

Solving this to get the nodal values u_j is actually solving a linear system of equations $\mathbf{Au} = \mathbf{b}$ where we define :

$$A_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j\, dxdy \tag{2.10}$$

$$b_i = -\int_\Omega f\phi_i\, dxdy \tag{2.11}$$

Those integral will be performed using equation 2.8.

2.2.3 Basis functions on the reference element and Gauss-Lobatto-Legendre nodes

As shows equations 2.10 and 2.11, we need to compute integrals to solve the problem. Here, we will perform the integration using Gauss-Lobatto-Legendre (GLL) quadrature. The particularity of 1D GLL nodes is that they include the end points of the interval on which we compute the integral. For example, on the one dimensional reference element, it means that the GLL nodes include $\xi = -1$ and $\xi = 1$. This particularity enables us to use the GLL nodes both as quadrature points and as global nodes where we compute our numerical solution. We will see later that this allows us to compute the matrix-vector product very efficiently.

For $p + 1$ GLL nodes, the quadrature integrate exactly polynomials of degree at most $2p - 1$. It is less than for the Gauss-Legendre quadrature (which achieve to integrate exactly polynomials up to order $2p + 1$). The way to compute the GLL nodes and their weights is for example detailled in [17]. Table 2.1 shows the values for different degrees. We can see that the nodes are not uniformly distributed but they tend to cluster near the end points $\xi = -1$ and $\xi = 1$.

Let us denote the $p + 1$ GLL nodes on the reference 1D element by ξ_0, ξ_1, ..., ξ_p. Since the GLL nodes contain the end points of the interval, we can also use them as global nodes. We will

Number of nodes $p+1$	Points ξ_i	Weights w_i
2	± 1	1
3	± 1	$\frac{1}{3}$
	0	$\frac{4}{3}$
4	± 1	$\frac{1}{6}$
	$\pm\sqrt{\frac{1}{5}}$	$\frac{5}{6}$
5	± 1	$\frac{1}{10}$
	$\pm\sqrt{\frac{3}{7}}$	$\frac{49}{90}$
	0	$\frac{32}{45}$

Table 2.1: Values of the GLL nodes and the associated weights on the reference one dimensional element for the first few degrees.

use Lagrangian polynomials for the interpolation. Let us denote them $l_0(\xi)$, $l_1(\xi)$, ... , $l_p(\xi)$. Let us recall that :

$$l_i(\xi) = \prod_{j=0, j\neq i}^{p} \frac{\xi - \xi_j}{\xi_i - \xi_j}$$

For the following parts of this chapter, it is very important to note that :

$$l_i(\xi_j) = \delta_{ij}$$

Where δ_{ij} is to be understood as the Kronecker delta. This fact will allow us later to compute the matrix-vector product in a very efficient way.

Let us also use this opportunity to define the derivation matrix H as :

$$H_{ij} = l_i'(\xi_j) \tag{2.12}$$

Let us now define the basis functions on the 2D reference element. In 2D, we will use $(p+1)^2$ nodes, tensor product of the 1D GLL nodes. If we denote the 1D GLL nodes by ξ_i^1, the 2D nodes are thus located at :

$$(\xi_i, \eta_j) = (\xi_i^1, \xi_j^1) \qquad \text{for } i, j = 0, 1, ..., p$$

On the reference quadrant $\xi, \eta \in [-1; 1]$, we have in the same way that the basis function associated with node I located at (ξ_i, η_j) is given the tensor product of the 1D basis functions :

$$\phi_I(\xi, \eta) = l_i(\xi) l_j(\eta)$$

Thus, if we want to obtain the value of a field u in the reference quadrant, whose value at node I is given by u_{ij}, we interpolate as :

$$u(\xi, \eta) = \sum_{i=0}^{p} \sum_{j=0}^{p} u_{ij} l_i(\xi) l_j(\eta)$$

2.2.4 Global basis function

Let us first mention the distinction between global and local nodes. Each quadrant has $(p+1)^2$ local GLL nodes but that does not mean that all of them are global nodes, since some might be hanging. As explained in [18], one solution consists to treat hanging nodes as global nodes and then add equations in the linear system $Au = b$ to enforce the continuity of the numerical solution. Another solution is to immediately enforce the continuity of the numerical solution by having continuous global basis function. This is what is done here. We will consider N global nodes (that are the GLL nodes for at least one quadrant) and give them a global unique index $I = 1, ..., N$.

We want our global basis function ϕ_I to be a piece-wise polynomial of order p and to have a value of exactly 1 at global node I and to be equal to 0 on any other global node.

In order to define such a function, we first need a mapping that goes from the 2D reference element to the actual quadrant e, i.e. $x^e(\xi, \eta)$ and $y^e(\xi, \eta)$ that for quadrant e give the actual coordinates of any point in the reference element. Let us also define the inverse mappings $\xi^e(x, y)$ and $\eta^e(x, y)$.

The last thing we need is a global to local operator that gives for local node (ξ_i, η_j) on quadrant e the weight of global node I. Let us denote this operator $R^e_{ij}(I)$. In a conforming mesh, since each local node correspond to a global node, $R^e_{ij}(I)$ can only be equal to 0 or to 1. In a non conforming mesh, however, the value of $R^e_{ij}(I)$ must be computed to ensure the continuity of the global basis function. For example, if we have hanging nodes on the east edge of quadrant e (where $i = p$), then $R^e_{pj}(I)$ is given by :

$$R^e_{pj}(I) = l_k\left(\frac{\xi_j}{2} \pm \frac{1}{2}\right) \qquad \text{if } I \text{ is the } k\text{-th global node on the non hanging east edge}$$

More explanation about how to compute the operator $R^e_{ij}(I)$ can be found in [19] or in [20]. The plus or minus sign in the above formula comes from the fact that a hanging edge can be the first or the second part of the larger non hanging edge. This is why we accept only 1-irregular meshes : because we only have to handle two cases when we compute $R^e_{ij}(I)$. If we accepted k-irregular meshes with $k > 1$, we would have too many different cases to look at.

With this operator and our mapping, it is then possible to compute the value of the basis function associated with node I inside any element e. It is given by :

$$\phi_I(x, y) = \sum_{i=0}^{p}\sum_{j=0}^{p} R^e_{ij}(I)\, l_i(\xi^e(x, y))\, l_j(\eta^e(x, y)) \qquad \text{if } (x, y) \in \Omega_e \tag{2.13}$$

Figure 2.3 shows a global basis function on the mesh presented in figure 2.1 for $p = 1$. We can see that it is not entirely trivial since the global node associated with the basis function has a influence in quadrants where we have hanging nodes. We can also see that it is indeed a continuous function that is a piece-wise bilinear polynomial.

Let us also note that there is no global basis function associated with a hanging node.

In practice, we do not build the operator $R^e_{ij}(I)$ explicitly since only nodes located on edges can be hanging and the p4est library helps us to treat hanging nodes (see the implementation chapter for more details).

2.2.5 Computing the right-hand side

Now that we have defined our global basis functions, it is possible to perform the integration needed to compute b_j (equation 2.11). As explained before, we will perform the integration quadrant by quadrant. Let us denote :

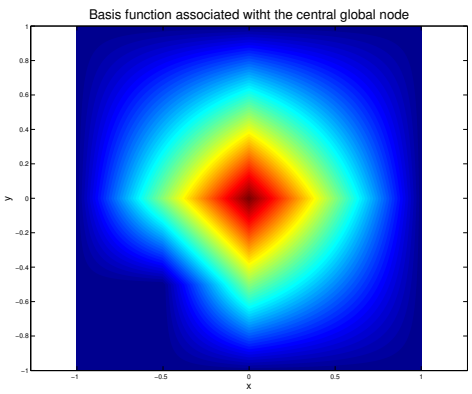


Figure 2.3: Global basis function associated with the global GLL node located in the center of the mesh presented in figure 2.1 for $p = 1$.

$$b_I^e = -\int_{\Omega_e} f\phi_I \, dxdy$$

We obviously have that $b_I = \sum_e b_I^e$. Let us also define the reference element associated with quadrant e by $\hat{\Omega}_e$ and the jacobian of the mapping as :

$$J^e(\xi,\eta) = \begin{pmatrix} \frac{\partial x^e}{\partial \xi} & \frac{\partial x^e}{\partial \eta} \\ \frac{\partial y^e}{\partial \xi} & \frac{\partial y^e}{\partial \eta} \end{pmatrix}$$

Then the integration on quadrant e becomes :

$$b_I^e = -\int_{\hat{\Omega}_e} f(x^e(\xi,\eta), y^e(\xi,\eta))\phi_I(x^e(\xi,\eta), y^e(\xi,\eta))|J^e(\xi,\eta)| \, d\xi d\eta$$

Using equation 2.13, we get :

$$b_I^e = -\int_{\hat{\Omega}_e} f(x^e(\xi,\eta), y^e(\xi,\eta)) \left(\sum_{i=0}^{p}\sum_{j=0}^{p} R_{ij}^e(I) l_i(\xi) l_j(\eta) \right) |J^e(\xi,\eta)| \, d\xi d\eta$$

This is where the fact that our global nodes are also GLL nodes becomes important. To alleviate a little the formulas, let us define two new notations : $J^e(\xi_m, \eta_n) = J_{mn}^e$ and $f(x^e(\xi_m,\eta_n), y^e(\xi_m,\eta_n)) = f_{mn}^e$. Using the GLL quadrature rule, we have :

$$b_I^e = -\sum_{m=0}^{p}\sum_{n=0}^{p} w_m w_n f_{mn}^e \left(\sum_{i=0}^{p}\sum_{j=0}^{p} R_{ij}^e(I) l_i(\xi_m) l_j(\eta_n) \right) |J_{mn}^e|$$

Since $l_i(\xi_j) = \delta_{ij}$, this becomes :

$$b_I^e = -\sum_{m=0}^{p}\sum_{n=0}^{p} w_m w_n f_{mn}^e R_{mn}^e(I) |J_{mn}^e|$$

This expression looks more complicated than it actually is. Indeed, the operator $R_{mn}^e(I)$ is often equal to zero. It is non zero only if in quadrant e, the local node (m,n) is the global node I (in which case it is equal to 1) or if the local node (m,n) is located on a hanging edge where the global node I has an influence. Therefore, in a quadrant where we do not have hanging nodes, and if the local node (m,n) has I as global number, we simply have :

$$b_I^e = -w_m w_n f_{mn}^e |J_{mn}^e|$$

2.2.6 Computing a matrix-vector product

As said in the introduction, we will use an iterative method to solve the system presented by equations 2.10 and 2.11, namely the preconditioned conjugate gradients. As a result, we do not need to build the matrix A explicitly but rather we need to be able to perform a matrix-vector product Au where u is a vector containing the values of our numerical solution at the global nodes, i.e. u_I is the value of the solution at global node I. By linearity of the integral and the derivative, and the definition of u^h, we have :

$$\begin{aligned}(Au)_I &= \sum_{J=1}^{N} \int_\Omega \nabla\phi_I \cdot \nabla\phi_J \, dxdy \, u_J \\ &= \int_\Omega \nabla\phi_I \cdot \nabla \left(\sum_{J=1}^{N} \phi_J u_J \right) dxdy \\ &= \int_\Omega \nabla\phi_I \cdot \nabla u^h \, dxdy\end{aligned}$$

Just as in the previous subsection, we will compute this integral quadrant by quadrant. The first thing we want to do is to compute the value of u^h at the local GLL nodes on quadrant e. Let us define, u^e_{mn} as this local value :

$$u^e_{mn} = \sum_{I=1}^{N} R^e_{mn}(I) u_I$$

Let us then define u^e as the restriction of u^h on quadrant e. Since we know that, on quadrant e, u^h must be a polynomial of degree p going through the $(p+1)^2$ GLL local nodes whose values are given by u^e_{mn} for $m, n = 0, ..., p$, and since we know that such a polynomial is unique, we have :

$$u^e(x,y) = \sum_{m=0}^{p} \sum_{n=0}^{p} u^e_{mn} l_m(\xi^e(x,y)) l_n(\eta^e(x,y)) \qquad \text{if } (x,y) \in \Omega_e$$

Using our restriction, the linearity of the different operators and the equation 2.13, we have :

$$\begin{aligned}(Au)^e_I &= \int_{\Omega_e} \nabla\phi_I \cdot \nabla u^h \, dxdy \\ &= \int_{\Omega_e} \nabla\phi_I \cdot \nabla u^e \, dxdy \\ &= \sum_{m=0}^{p} \sum_{n=0}^{p} R^e_{mn}(I) \int_{\Omega_e} \nabla \left(l_m(\xi^e(x,y)) l_n(\eta^e(x,y)) \cdot \nabla u^e \, dxdy \right. \qquad (2.14)\end{aligned}$$

Let us now focus on the last part of equation 2.14. Since we want to integrate on the reference element, we first have to take care of the derivatives. To simplify the notations, let us define : $L_{mn}(\xi,\eta) = l_m(\xi) l_n(\eta)$. Then, using the chain rule, we have :

$$\begin{aligned}\int_{\Omega_e} \nabla \left(l_m(\xi^e(x,y)) l_n(\eta^e(x,y)) \cdot \nabla u^e \, dxdy = \int_{\hat{\Omega}_e} &\left(\frac{\partial L_{mn}}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial L_{mn}}{\partial \eta}\frac{\partial \eta}{\partial x} \right) \left(\frac{\partial u^e}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{u^e}{\partial \eta}\frac{\partial \eta}{\partial x} \right) |J^e| \\ &+ \left(\frac{\partial L_{mn}}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial L_{mn}}{\partial \eta}\frac{\partial \eta}{\partial y} \right) \left(\frac{\partial u^e}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{u^e}{\partial \eta}\frac{\partial \eta}{\partial y} \right) |J^e| \, d\xi d\eta\end{aligned}$$

It is then a matter of rearranging the different terms.

$$\int_{\Omega_e} \nabla \left(l_m(\xi^e(x,y)) l_n(\eta^e(x,y)) \cdot \nabla u^e \, dxdy = F^{\xi\xi}_{mn;e} + F^{\xi\eta}_{mn;e} + F^{\eta\xi}_{mn;e} + F^{\eta\eta}_{mn;e} \right.$$

$$F^{\xi\xi}_{mn;e} = \int_{\hat{\Omega}_e} \frac{\partial L_{mn}}{\partial \xi} \frac{\partial u^e}{\partial \xi} \left(\frac{\partial \xi}{\partial x}\frac{\partial \xi}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \xi}{\partial y} \right) |J^e| d\xi d\eta$$

$$F^{\xi\eta}_{mn;e} = \int_{\hat{\Omega}_e} \frac{\partial L_{mn}}{\partial \xi} \frac{\partial u^e}{\partial \eta} \left(\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y} \right) |J^e| d\xi d\eta$$

$$F^{\eta\xi}_{mn;e} = \int_{\hat{\Omega}_e} \frac{\partial L_{mn}}{\partial \eta} \frac{\partial u^e}{\partial \xi} \left(\frac{\partial \eta}{\partial x}\frac{\partial \xi}{\partial x} + \frac{\partial \eta}{\partial y}\frac{\partial \xi}{\partial y} \right) |J^e| d\xi d\eta$$

$$F^{\eta\eta}_{mn;e} = \int_{\hat{\Omega}_e} \frac{\partial L_{mn}}{\partial \eta} \frac{\partial u^e}{\partial \eta} \left(\frac{\partial \eta}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \eta}{\partial y}\frac{\partial \eta}{\partial y} \right) |J^e| d\xi d\eta$$

We will again use the GLL quadrature to compute those integrals. Once again, the fact that the quadrature nodes are the same as our local nodes will allow us to compute the different terms efficiently. Let us introduce a few definitions :

$$W^{\xi\xi}_{ij;e} = w_i w_j |J^e(\xi_i,\eta_j)| \left(\frac{\partial \xi}{\partial x}\frac{\partial \xi}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \xi}{\partial y} \right)(\xi_i,\eta_j)$$

$$W^{\xi\eta}_{ij;e} = w_i w_j |J^e(\xi_i,\eta_j)| \left(\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y} \right)(\xi_i,\eta_j)$$

$$W^{\eta\eta}_{ij;e} = w_i w_j |J^e(\xi_i,\eta_j)| \left(\frac{\partial \eta}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \eta}{\partial y}\frac{\partial \eta}{\partial y} \right)(\xi_i,\eta_j)$$

The derivatives that appear in the expression above do not need an analytic expression for the inverse mapping to be available. We can compute them using the inverse of the jacobian matrix to obtain the values of $\frac{\partial xi}{\partial x}(\xi_i,\eta_j)$,...

Let us also remind the definition of the derivation matrix H given by equation 2.12. Using the quadrature, we have :

$$\begin{aligned} F^{\xi\xi}_{mn;e} &= \sum_{a=0}^{p}\sum_{b=0}^{p} W^{\xi\xi}_{ab;e} l'_m(\xi_a) l_n(\eta_b) \left(\sum_{c=0}^{p}\sum_{d=0}^{p} u^e_{cd} l'_c(\xi_a) l_d(\eta_b) \right) \\ &= \sum_{a=0}^{p}\sum_{b=0}^{p} W^{\xi\xi}_{ab;e} H_{ma}\delta_{nb} \left(\sum_{c=0}^{p}\sum_{d=0}^{p} u^e_{cd} H_{ca}\delta_{db} \right) \\ &= \sum_{a=0}^{p} W^{\xi\xi}_{an;e} H_{ma} \left(\sum_{c=0}^{p} u^e_{cn} H_{ca} \right) \end{aligned}$$

If we denote U^e a matrix such that $(U^e)_{ij} = u^e_{ij}$ and $W^{\xi\xi}_e$ a matrix such that $(W^{\xi\xi}_e)_{ij} = W^{\xi\xi}_{ij;e}$ and if we denote the Hadamard product by $\circ$, then we can write :

$$F^{\xi\xi}_{mn;e} = \left(H(W^{\xi\xi}_e \circ H^T U^e) \right)_{mn}$$

For similar definitions, it can be showed that the order terms are given by :

$$F^{\xi\eta}_{mn;e} = \left(H(W^{\xi\eta}_e \circ U^e H) \right)_{mn}$$

$$F^{\eta\xi}_{mn;e} = \left((W^{\eta\xi}_e \circ H^T U^e) H^T \right)_{mn}$$

$$F^{\eta\eta}_{mn;e} = \left((W^{\eta\eta}_e \circ U^e H) H^T \right)_{mn}$$

We can see that the problem reduce to computing a few matrix products. The only thing left to do is to take hanging nodes into account when we gather the results :

$$(Au)_I^e = \sum_{m=0}^{p} \sum_{n=0}^{p} R_{mn}^e(I) \left(F_{mn;e}^{\xi\xi} + F_{mn;e}^{\xi\eta} + F_{mn;e}^{\eta\xi} + F_{mn;e}^{\eta\eta} \right)$$

And, of course, we need to take the influence of every quadrant into account :

$$(Au)_I = \sum_e (Au)_I^e$$

2.3 Preconditioned conjugate gradients

Now that we have our right hand side vector b and that we are able to compute a matrix-vector product Au, we can explain how to solve the linear system.

Since the number of degrees of freedom is huge, it is impossible to use a direct method to solve the linear system. Instead, we will turn ourselves to iterative methods.

The matrix A arises from the discretization of an Poisson equation using finite elements and therefore it is sparse, symmetric and positive-definite. The conjugate gradients method is therefore well suited to solve the linear system. We have to note however that the convergence speed of the conjugate gradients depends on the condition number of the matrix A, $\kappa(A)$. Indeed, as given in [21], if we denote $e_i = u - u_i$ where u is the solution of the linear system and u_i is our approximation after the i-th iteration, then :

$$||e_i||_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i ||e_0||_A \tag{2.15}$$

Where $||e_i||_A = \left(e_i^T A e_i \right)^{\frac{1}{2}}$. We can see in equation 2.15 that the higher the condition number of A, the slower the convergence.

Unfortunately, the condition number of the matrix A is often very large and that is why we need to use a preconditioner.

Formally, we want to solve :

$$Au = b$$

Which is equivalent to solving :

$$M^{-1}Au = M^{-1}b$$

If M is easy to invert and $\kappa(M^{-1}A) \ll \kappa(A)$, then we should have a faster convergence at not too great a cost. The problem is that it is not possible to guarantee that $M^{-1}A$ is either symmetric nor positive definite, which is required for the CG.

However, we know that if M is symmetric and positive-definite, then there exists a matrix E such that $M = EE^T$. We can also note that $M^{-1}A$ and $E^{-1}AE^{-T}$ have the same eigenvalues. Indeed, let us assume that v if an eigenvector of $M^{-1}A$ associated with eigenvalue λ. Then :

$$E^{-1}AE^{-T}(E^T v) = (E^T E^{-T})E^{-1}Av = E^T M^{-1}Av = \lambda E^T v$$

And we can conclude that $E^T v$ is an eigenvector of $E^{-1}AE^T$ with eigenvalue λ. If the two matrices have the same eigenvalues, they also have the same condition number. If A is symmetric

and positive-definite, it is clear that $E^{-1}AE^{-T}$ is also symmetric and positive-definite. We can thus use the CG to solve the following system (which is equivalent to $Au = b$) :

$$E^{-1}AE^{-T}\hat{u} = E^{-1}b \qquad\qquad \hat{u} = E^T u$$

This formulation has the unwanted property of making explicit use of the matrix E. We would like to remove it entirely and use M only. It is indeed possible (see [21] for details).

Let us define u_i the approximation after the i-th approximation and $r_i = b - Au_i$. Then, the PCG procedure is given by algorithm 1. We can note that we indeed do not mention E but use M^{-1} instead. We can also see that we start with a zero initial guess. It does not have to be the case and if we had a better initial guess, we should use it.

Algorithm 1 Preconditioned Conjugate Gradients

$u_0 = 0$
$r_0 = b - Au_0$
$z_0 = M^{-1}r_0$
$p_0 = z_0$
$i = 0$
while $||r_i||_2 > \epsilon||r_0||_2$ **do**
  $d_i = Ap_i$
  $\alpha_i = \frac{r_i^T z_i}{p_i^T d_i}$
  $u_{i+1} = u_i + \alpha_i p_i$
  $r_{i+1} = r_i - \alpha_i d_i$
  $z_{i+1} = M^{-1}r_{i+1}$
  $\beta_{i+1} = \frac{z_{i+1}^T r_{i+1}}{z_i^T r_i}$
  $p_{i+1} = z_{i+1} + \beta_{i+1} p_i$
  $i = i + 1$

The stopping criterion is defined using the norm of the residual. We want it to be less than a given fraction of the norm of the initial residual.

The most important part in the algorithm presented is the preconditioner M^{-1}. Indeed, the convergence speed is a function of $\kappa(M^{-1}A)$. Therefore, we want a preconditioner that is easy to compute and such that the condition number of $M^{-1}A$ is a lot smaller than the one of A. Oftentimes, those two goals are contradictory.

As explained in the introduction, our preconditioner here consists of two parts. We have one coarse grid correction based on solving the problem on the mesh with an interpolation of degree $p = 1$. The resolution of this coarse problem will be done using a multigrid method. Let us call this part P^c. The second part is the fine preconditioner where we solve the problem exactly on overlapping subdomains (this is often called an additive Schwarz preconditioner). Let us call this part P^f. We then add the two part of the preconditioner :

$$M^{-1} = P^c + P^f$$

The hope is that the two parts of the preconditioner act on different parts of the problem and that the sum of the two makes a good preconditioner. The rest of this chapter is entirely about how to compute $M^{-1}r = P^c r + P^f r$.

2.4 Coarse preconditioner : the multigrid solver

Let us first present the coarse part of the preconditioner. It consists of solving the problem on the mesh using an interpolation of degree $p = 1$ with a geometric multigrid method.

Since in general we use an interpolation degree p, the first thing to do is to restrict this residual on the same mesh but for a degree $p = 1$. The first part of this section presents the restriction used. Similarly, once we have computed the solution for $p = 1$, we have to prolong it to a higher degree. To preserve the symmetry of the problem, we will define the prolongation operator to be the transpose of the restriction operator. The procedure explained here is an application with hanging nodes of the one described in [22]. Formally, if we denote by S the restriction operator and by G the geometric multigrid solver, we have :

$$P^c r = S^T G S r$$

The second part explains in more details what are the key components of the geometric multigrid solver. The practical implementation and the way it works with p4est is however described in the next chapter.

We have to note that the coarse preconditioner cannot be use as the only preconditioner for the CG. Indeed, because of the restriction, it has a kernel, i.e. it is possible to have a high degree residual $r \neq 0$ such that :

$$Sr = 0$$

As a result, the PCG will stop even tough we have not reached the solution. That is why, in the results chapter, we never test the multigrid alone with a degree superior to $p = 1$.

2.4.1 Restriction of r from a high degree to $p = 1$

Let us now define the operator S that will restrict the high degree residual r to a low degree $p = 1$. Authors of [23] have shown that a good choice for the restriction is the L^2 projection.

Let us call $V_p^h \subset H^1(\Omega)$ the space whose basis consists of the high degree global basis functions (later denoted ϕ_i for $i = 1, ..., N$) and $V_1^h \subset H^1(\Omega)$ the one whose basis is constituted by the bilinear global basis function (later denoted Φ_i for $i = 1, ..., M$). We want to find $R \in V_1^h(\Omega)$ such that R is the L^2 projection of $r \in V^p(\Omega)$ onto $V_1^h(\Omega)$. As shown in [22], the L^2 projection is then given by :

$$R = Cm^{-1}r$$

Where m is the high degree mass matrix and C is called the correlation matrix. The mass matrix m is easy to invert since in the case of spectral elements, it is diagonal, i.e. we have :

$$m_{ij} = \int_\Omega \phi_i \phi_j \, dxdy \approx 0 \qquad \text{if } i \neq j$$

The other term is the correlation matrix. It is defined by :

$$C_{I,i} = \int_\Omega \Phi_I \phi_i \, dxdy$$

We can also define its local counterpart using local shape functions. Let us denote l_i the one dimensional shape function of degree p associated with the i-th GLL node and L_i the one dimensional linear shape function associated with the i-th GLL node. Then we have :

$$C_{IJ,ij;e} = \int_{\Omega_e} L_I L_J l_i l_j \, dxdy$$

If we use the GLL quadrature using the $(p+1)^2$ GLL nodes on quadrant e, we get :

$$C_{IJ,ij;e} \approx w_i w_j |J_{ij}^e| \Phi_I(\xi_i) \Phi_J(\xi_j)$$

For later purposes, let us define :

$$m_{ij;e} = w_i w_j |J^e ij|$$
$$B_{IJ,ij} = \Phi_I(\xi_i)\Phi_J(\xi_j)$$

We can then compute the restriction of r as :

$$y = m^{-1} r$$
$$R = Cy$$

Applying m^{-1} to r is rather easy : we only scale the residual by the inverted mass matrix. We now have to transfer y_i for $i = 1, ..., N$ onto each local quadrant e. For this, let us use the operator R^e_{ij} to handle hanging nodes :

$$y_{ij;e} = \sum_{K=1}^{N} R^e_{ij}(K) y_K$$

Let us then compute the local coarse grid residual using the correlation matrix in its local form. This yields :

$$R_{IJ;e} = \sum_{i=0}^{p} \sum_{j=0}^{p} B_{IJ,ij} m_{ij;e} y_{ij;e}$$

The last remaining thing to do is to gather the local coarse grid residual to obtain the global coarse grid residual. Let us do not forget to handle the hanging nodes and therefore to use the operator R^e_{IJ} (not to confuse with $R_{IJ;e}$ the local coarse grid residual on quadrant e). Let us note that it is not the same as R^e_{ij} since it works on the global nodes for bilinear interpolation where R^e_{ij} is used for the interpolation of degree p.

$$R_K = \sum_{e} \sum_{I=0}^{1} \sum_{J=0}^{1} R^e_{IJ}(K) R_{IJ;e}$$

This residual R is then used as a right-hand side for the multigrid solver.

2.4.2 Prolongation of the solution from $p = 1$ to a high degree

Let us assume that the solution given by the multigrid solver with R as right-hand side is given by Z, i.e. :

$$Z = GR$$

We now need to prolong this coarse scale correction onto the fine grid. Since the prolongation operator is defined as the transpose of the restriction operator, we have :

$$P^c r = m^{-1} C^T Z$$

The global coarse correction Z is first scattered to each element to have the local coarse scale correction. Let us not forget to handle hanging nodes :

$$Z_{IJ;e} = \sum_{K=1}^{M} R^e_{IJ}(K) Z_K$$

Then, let us prolong the correction to obtain :

$$z_{ij;e} = \sum_{I=0}^{1}\sum_{J=0}^{1} B_{IJ,ij} m_{ij;e} Z_{IJ;e}$$

We then have to gather $z_{ij;e}$ to compute its global counterpart. Once again, this is where we handle hanging nodes.

$$z_K = \sum_{e}\sum_{i=0}^{p}\sum_{j=0}^{p} R_{ij}^{e}(K) z_{ij;e}$$

The last thing left to do is to scale z_K by the inverted mass matrix. This yields :

$$(P^c r)_K = \frac{z_K}{m_K}$$

2.4.3 The geometric multigrid solver

This subsection generally describe how the geometric multigrid method works and why it is so effective. However, a lot of details have been omitted and more information can be obtained in the relevant literature (for example in [24]). It has to be noted that here the degree of interpolation is always $p = 1$.

Motivation

The first observation we have to make is that for any known function v such that $v(x, y) = u_0$ if $(x, y) \in \Gamma$, we can rewrite problem 2.1. Indeed, if we define a function $e = u - v$, then we have

$$\nabla^2 e = f - \nabla^2 v = r \qquad \text{on } \Omega \tag{2.16}$$

$$e = u - v = 0 \qquad \text{on } \Gamma \tag{2.17}$$

Which is qualitatively the same problem. But if we are able to solve it for e more easily than we would have been able to do it for u, then we can simply recover u from e :

$$u = v + e$$

The second observation we have to make is that the discretization of problem 2.1 or equivalently the one above yields a linear system of equations to solve (as the one derived in the first section of this chapter). As already mentioned, the matrix A defining this linear system has very often a very large condition number $\kappa(A)$. As a result, using iterative methods can be very slow. Typically, the condition number of A for $p = 1$ is $\mathcal{O}(h^2)$ where h is the size of the smallest quadrant (see [25]). Thus, if we use a coarser grid (where the smallest quadrant has a larger size than before), the problem is easier to solve.

The idea behind multigrid is therefore a two-level idea. It works as follow :

1. Given an approximate solution to the problem, compute the residual $f - \nabla^2 v$.
2. Solve problem 2.16 to obtain e. It should be easier to solve than the initial problem by using a coarser grid.
3. Add e to v to get the solution u.

The next step in the reasoning is to use this idea recursively. Indeed, in step 2, to solve problem 2.16, one can reuse the same process. A more precise definition of the algorithm is given at the end of this section, in algorithms 2 and 3.

We still have not explained two things in the procedure above. The first is how to choose an approximate solution to the problem. In theory, it can be any but we have to remember that, in step 2, we will use a coarser grid to solve for e. As a result, we want an approximation such that $e = u - v$ is a smooth function that a coarse grid can capture well. One way to obtain this approximation is by using an iterative method. Indeed, it has been shown (for example, in [26]) that even tough iterative methods like Jacobi or Gauss-Seidel have a tendency to stall, just a few iterations allows them to capture the high-frequency modes in the solution. That means that if we compute v by doing some iterations of the Jacobi method, then $e = u - v$ will contain only low frequency modes. And a solution containing low frequencies can be computed on a coarser grid. This property that the high frequency errors converge a lot quicker than the low frequency error is called the smoothing property and is hold by a lot of different smoother (for example, the Jacobi method).

The second thing we have to consider is how to transfer one function onto another grid. Indeed, in step 1, we compute the residual $f - \nabla^2 v$ but then we have to solve problem 2.16 on a different, coarser grid. We therefore need an operator to compute the right-hand side of the linear system on the coarser grid from the right-hand side of the one on the initial grid. This is called the restriction operator. Similarly, when we have computed the solution on the coarser grid in step 2, we need to update our approximation v with e on the initial grid. This operator is called the prolongation operator. We will see that there exists a link between those two operators.

The smoother : damped Jacobi method

Let us now present the smoother used : the damped Jacobi method. Let us assume that we want to solve :

$$Av = b$$

Let us then split the matrix A in the form :

$$(D - L - U)v = b$$

Where D is a diagonal matrix containing the diagonal of A, $-L$ is the strictly lower triangular part and $-U$ is the strictly upper triangular part. Let us also denote as v^i the approximation of the solution of $Av = b$ after the i-th iteration. Then, the damped Jacobi method with damping factor ω is given by :

$$v^{i+1} = \omega \left(D^{-1}(L + U)v^i + D^{-1}b \right) + (1 - \omega)v^i$$

It is to be noted that since D is diagonal, it is very cheap to compute D^{-1}. It can be shown (see [24]) that if $0 < \omega \leq 1$ then the damped Jacobi method is stable. The parameter ω can then be tuned to obtain different properties in the convergence of high frequency errors. In [24], it is shown that a good choice for ω is :

$$\omega = \frac{2}{3}$$

We now have to explain how to compute A for the different grids. Since it results from the discretization of problem 2.16, we can use the same method as the one explained in the section concerning the spectral element method. It is here even easier since we have piece-wise bilinear global basis function. We however still have to be careful of hanging nodes. The next chapter explains in more details the implementation.

(a) Mesh at level 1

(b) Mesh at level 2

Figure 2.4: Example of two meshes used in the geometric multigrid method. The one on the left is the coarser mesh, where all quadrants have a refinement level of 1. The one on the right has some quadrants that have a level of refinement of 2 and is therefore the level 2 mesh. We can also see the influence of increasing by one level. We have the apparition of three new global nodes (in red) and two new hanging nodes (in blue).

Prolongation operator

Let us now define the prolongation operator, i.e. the operator that takes the solution to problem 2.16 on the coarse grid and prolong it on the fine grid.

In order to do this, we have to precise the nature of the different meshes that will be used. As explained earlier, we will solve the problem on different meshes. Let us organize those meshes on different levels, where the lowest level has the coarsest mesh and the highest level has the finest mesh. The AMR provides us with a natural choice for the definition of those meshes. Indeed, since each quadrant is dynamically refined, we will consider the mesh at level lev to be the original mesh where the maximum level of refinement is lev (see for example [27]). An example is given on figure 2.4. We can see that the mesh on the left has all its quadrants with a refinement level of 1, it is thus the mesh at level 1. On the right, the mesh has some (but not all) quadrants that have a refinement level of 2. It is therefore the mesh at level 2.

Now that we have some sense of how the meshes are constructed, it is possible to define the prolongation operator. We can see on figure 2.4b the influence of increasing by one level. There is the apparition of three new global nodes (in red) and two new hanging nodes (in blue).

Since we use a bilinear interpolation (thus linear along each edge), the prolongation operator is quite natural : the new node created in the middle of a refined quadrant (and that can never be hanging) is equal to a fourth of the sum of the values at the corners of the larger quadrant. Concerning the nodes added on the edges, if they are not hanging, they are equal to half of the sum of the values at the ends of the edge. Of course, we do not need to compute the prolongation for the hanging edges since they are not global.

Restriction operator

As before, to conserve the symmetry of the problem, the restriction operator is defined as the transpose of the prolongation operator.

It means that if a group of four quadrants has to be coarsened to move to the mesh a level below, the value of the residual at the middle node (once again, it can never be hanging) contributes a fourth of its value to the residual at the four corners of the larger quadrant. Similarly, the residual at a node on an edge that is not hanging contributes half of its value to both nodes at the ends of the edge.

Formalization of the algorithm

In this last part, we will give the pseudo-code for the geometric multigrid solver. Let us denote A^{lev} the matrix arising form the discretization on the mesh at level lev, I_{lev}^{lev+1} the prolongation operator, I_{lev+1}^{lev} the restriction operator, v^{lev} the value of the solution at level lev and r^{lev} the value of the right-hand side at level lev.

Algorithm 2 μ-cycle scheme

function Mμ(v^{lev},r^{lev})
  if $lev = 0$ **then**
    Solve $A^0 v^0 = r^0$ using a direct method
  else
    Do ν_1 iterations of the damped Jacobi method on $A^{lev}u = r^{lev}$
    $r^{lev-1} = I_{lev}^{lev-1} r^{lev}$
    $v^{lev-1} = 0$
    $v^{lev-1} = M\mu(v^{lev-1}, r^{lev-1})$ μ times
    $v^{lev} = v^{lev} + I_{lev-1}^{lev} v^{lev-1}$
    Do ν_2 iterations of the damped Jacobi method on $A^{lev}u = r^{lev}$
  return v^{lev}

Algorithm 2 describes what is called a μ-cycle scheme. We can see several parameters in this algorithm. First, the number of times we recursively use the function, μ. In practice, only $\mu = 1$ (called a V-cycle) and $\mu = 2$ (called a W-cycle) are used. We also have the number of smoothing iterations we do before restricting the residual, ν_1, called the pre-smoothing parameter and the number of smoothing iterations we do after the correction, ν_2, called the post-smoothing parameter.

The last thing to say is that we can use the μ-cycle scheme as an iterative process, each iteration being one μ-cycle. Algorithm 3 describe how the procedure works to solve the problem defined by $Au = b$.

Algorithm 3 Geometric multigrid solver

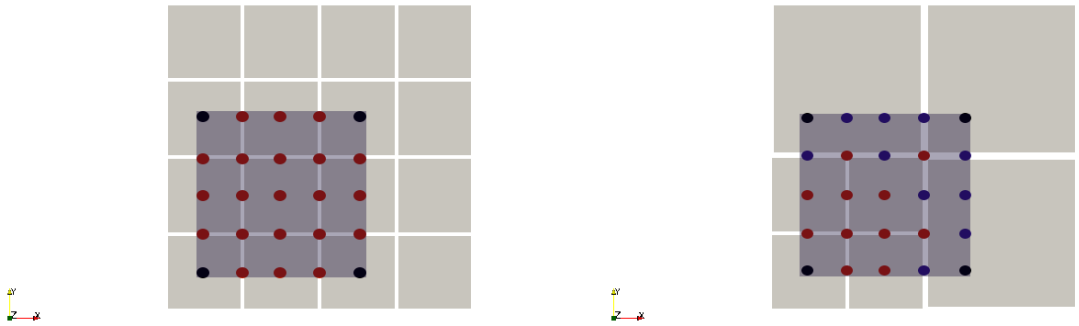
$v = 0$
$r = b - Av$
for $i = 0, 1, ...$ **do**
  $e = 0$
  $e = M\mu(e, r)$
  $v = v + e$
  $r = b - Av$

An important characteristic of geometric multigrid methods is the h-independent convergence. To obtain a given tolerance of the norm of the residual r, the number of iterations needed in algorithm 3 remains constant for any number of levels. Of course, the time needed to perform an iteration increases if we have more levels but the number of iterations remains identical.

2.5 Fine preconditioner : overlapping additive Schwarz

Let us now move on the fine part of the preconditioner, P^f. As explained before, it consists of an OAS preconditioner. The general idea is to solve problem 2.1 with homogeneous Dirichlet boundary conditions on several subdomains that overlap and with the residual as the right-hand side. The results are subsequently added to form the fine scale preconditioner.

The procedure explained here is to a large extend inspired from [22], we only adapt it to handle hanging nodes.



(a) Subdomain for $p = 2$ on a conforming mesh (b) Subdomain for $p = 2$ on a non conforming mesh

Figure 2.5: Example of two subdomains for $p = 2$ used in the overlapping Schwarz preconditioner. On the left, we have a conforming mesh and all the local nodes are also global nodes (red). On the right, we have a non conforming mesh. Some nodes coincide with global GLL nodes (in red) but for others (in blue), we must use an interpolation to get their value. For both, the nodes at the corners of the subdomain (in black) will not be considered.

2.5.1 Overlapping subdomains

We need to divide the mesh into overlapping subdomains. A natural choice is to use the quadrants and add a layer of GLL nodes to each side from neighboring quadrants. That yields $(p+3)^2$ local nodes on each subdomain.

An example of the subdomains for $p = 2$ is given in figure 2.5. On the left, the mesh is conforming. That means that all the local nodes on subdomain s coincide with global nodes. However, in a non conforming mesh, in addition to hanging nodes on some edges, we also need to interpolate the residual when, as presented in figure 2.5b, we are hanging with respect to our neighbor. There is also need for interpolation when at least one of our neighbor has hanging nodes. We can see here that adding hanging nodes brings a lot of new complexities. We will explain in more detail how they are treated in the implementation chapter.

For now, let us assume that we have an operator $T_{ij;s}$ that gives the value of the local residual at local node (i, j) on subdomain s. If we denote the global residual by r and its restriction on subdomain s by $r_{ij;s}$, then we have :

$$r_{ij;s} = T_{ij;s} r \qquad \text{for } i, j = -1, 0, ..., p, p+1$$

Let us also note that, even in the conforming case, it is not always possible to compute the residual at the four corners of the subdomain (nodes in black on figure 2.5a). That is because in an unstructured mesh, a quadrant can have several other quadrants as neighbors through a corner or none at all. Therefore, we will impose that :

$$r_{ij;s} = 0 \qquad \text{at the four corners of the subdomain}$$

2.5.2 Additive Schwarz method

A more detailled explanation of the additive Schwarz method and the domain decomposition can be found in [28]. As said earlier, we now want to solve problem 2.1 to get the local fine scale correction, denoted by $z_{ij;s}$. In order to do this, we will impose homogeneous Dirichlet boundary conditions on nodes located at $i, j = -2$ or $i, j = p + 2$.

As explained in [29], since we are using the additive Schwarz method as a preconditioner, we do not need to be extremely accurate in the solution of problem 2.1. We will also make an assumption that will allow us to compute the fine scale correction efficiently : we will assume that every quadrant is aligned with the axes. That means that a lot of geometric factors cancel and we can actually solve the problem analytically. We will see in the results chapter that, in practice, we can indeed make this assumption.

Let us first consider the equivalent 1D problem for an element of length h. The stiffness matrix is then given by :

$$A_{ij}^e = \int_0^h \frac{dl_i}{dx}\frac{dl_j}{dx}dx \approx \frac{2}{h}\sum_{m=0}^{p} w_m H_{im} H_{jm} = \frac{2}{h} d_{ij}$$

The influence of the overlaps (points at -1 and $p+1$) is computed using the standard finite element procedure : by adding their contribution to the stiffness matrix. We also assume that the neighboring elements have the same size so that the factor h does not change. As a result, the linear system we obtain with the discretization is given by :

$$A = \begin{pmatrix} d_{11} & d_{01} & 0 & \dots & \dots & \dots & 0 \\ d_{10} & 2d_{00} & d_{01} & \dots & \dots & d_{0p} & 0 \\ 0 & d_{10} & d_{11} & \dots & \dots & d_{1p} & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & d_{p-1,0} & \dots & \dots & d_{p-1,p-1} & d_{p-1,p} & 0 \\ 0 & d_{p0} & \dots & \dots & d_{p,p-1} & 2d_{pp} & d_{p-1,p} \\ 0 & \dots & \dots & \dots & 0 & d_{p,p-1} & d_{p-1,p-1} \end{pmatrix}$$

$$M = \begin{pmatrix} w_1 & & & & & & \\ & 2w_0 & & & & & \\ & & w_1 & & & & \\ & & & \ddots & & & \\ & & & & w_{p-1} & & \\ & & & & & 2w_p & \\ & & & & & & w_{p-1} \end{pmatrix}$$

$$\frac{4}{h^2}M^{-1}Az = \frac{2}{h}M^{-1}r$$

As mentioned in [22], the matrix $L = M^{-1}A$ is diagonalizable and it has real and positive eigenvalues λ_i. So let us define :

$$L = V^{-1}\Lambda V$$

That leads to an analytic solution for z. Indeed :

$$z = \frac{h}{2}V^{-1}\Lambda^{-1}VM^{-1}r$$

Let us now try to generalize this procedure to the two dimensional case. As explained before, let us assume that the subdomain s is aligned with the axes x and y and has dimensions h_x and h_y. Then the system of equations we have to solve to find the local fine scale correction on subdomain s, $z_{ij;s}$, is given by :

$$\sum_{m=-1}^{p+1} \frac{4}{h_x^2}L_{im}z_{mj;s} + \frac{4}{h_y^2}L_{jm}z_{im} = \frac{4}{h_x h_y}\frac{1}{M_{ii}M_{jj}}r_{ij;s} \qquad \text{for } i,j = -1,...,p+1 \qquad (2.18)$$

The reader can compare equation 2.18 with the one derived in the section about the spectral element method to convince himself that it is indeed the set of equations obtained when the quadrant is aligned with the axes.

Let us define the matrices Z_s and R'_s such that $(Z_s)_{ij} = z_{ij;s}$ and $(R'_s)_{ij} = \frac{4}{h_x h_y}\frac{1}{M_{ii}M_{jj}} r_{ij;s}$. Then, equation 2.18 can be written as :

$$\frac{4}{h_x^2} L Z_s + \frac{4}{h_y^2} Z_s L^T = R'_s$$

Using the diagonalization of L, we have :

$$\begin{aligned} \frac{4}{h_x^2} V^{-1}\Lambda V Z_s + \frac{4}{h_y^2} Z_s V^T \Lambda V^{-T} &= R'_s \\ \frac{4}{h_x^2}\Lambda V Z_s V^T + \frac{4}{h_y^2} V Z_s V^T \lambda &= V R'_s V^T \end{aligned} \tag{2.19}$$

Let us define $W = V Z_s V^T$ and $D = V R'_s V^T$. Then, it is obvious that the solution to equation 2.19 is given by :

$$\begin{aligned} W_{ij} &= \frac{1}{\frac{4}{h_x^2}\lambda_i + \frac{4}{h_y^2}\lambda_j} D_{ij} \qquad \text{for } i,j = -1, ..., p+1 \\ Z_s &= V^{-1} W V^{-T} \end{aligned}$$

We have reduced the problem to a few matrix products (where the matrices have $(p+3)$ rows and columns). If we had taken the geometric factors into account, we would have had a system with $(p+3)^2$ unknowns to solve for each subdomain s. That is $\mathcal{O}((p+3)^6)$ operations using a direct method. Here, we only need to perform matrix products and therefore we need $\mathcal{O}((p+3)^3)$ operations. The gain would be even greater in three dimensions. Let us note that the diagonalization of L can be done once at the beginning since it does not depend on the subdomain.

The last thing we have to do is to gather the local fine preconditioner to the global GLL nodes to form the global fine preconditioner. As before, the gather operator is the transpose of the restriction operator. That yields :

$$P^f r = \sum_s \sum_{i=-1}^{p+1} \sum_{j=-1}^{p+1} T^T_{ij;s} z_{ij;s}$$

2.5.3 Managing the actual boundary

Up until now, we have assumed that the subdomain was included in Ω. But some quadrants have at least an edge on Γ. In order to be able to use the analytic solution described above, we however need to give a value to the local fine residual.

The method used here is to interpolate the value of the residual from inside the domain. For example, if the south edge is on the boundary Γ, then the local fine residual is given by :

$$r_{i,-1;s} = 2r_{i,0} - r_{i,1} \qquad \text{for } i = 0, ..., p$$

Of course, we do not gather the value of the fine scale correction that is outside of Ω.

Chapter 3

Implementation

The most important part of the work done in realizing this thesis was the implementation, from scratch, of the different methods presented in the theory chapter. It resulted in a code written in plain C and consisting of more than 8000 lines.

Although it is impossible to present every part of the code here due to its shear volume, some parts are worth explaining. In this chapter, we will look into different the aspects of the implementation that are not immediately obvious.

The first section of this chapter presents the *p4est* library. It is an essential part of the implementation and it interacts with our application at all times. This is why we spend some time showing the components and the philosophy behind *p4est*. It is also important to understand the library to understand how we handle the mesh and its hanging nodes throughout the application.

The implementation of the spectral element method is quite straightforward. The only thing we present here is how to compute the R_{ij}^e operator.

The second section looks at the geometric multigrid method. The essential ingredient is a structure that handles all levels and the information needed on each. The building of such structure is presented here.

The last section focuses on the implementation of the fine preconditioner. We show the different possible configurations as far as neighbors are concerned and we present how to compute the local residual in every case.

3.1 The p4est library

Managing the mesh, especially when we have the presence of hanging nodes, is an important part of the work in any SEM implementation. This section thus present the library used, namely *p4est* (the three dimensional equivalent is called *p8est*).

The founding principles of *p4est* are given in [4]. In that article, they generate up to $5.13\,10^{11}$ octants (the three dimensional equivalent of the quadrant) on as many as $2.2\,10^5$ CPU cores. Some additional information about the high-order node numbering can also be found in [30]. We present here only the features relevant to our work.

We first present the approach to AMR used by *p4est* and how it manages the dynamic refinement process, namely the refine and coarsen operators. We also explain the different structures used to build the mesh and the inherent terminology.

In the second part, we will describe how, in our program, we handle hanging nodes leveraging *p4est* and its structures. The way to compute the operator R_{ij}^e (defined in the theory chapter) is also laid out.

3.1.1 AMR for p4est

Different approaches exists for handling non conforming meshes. One strategy consists of splitting the computational domain into several unstructured macro-elements and then to refine them uniformly. The adaptivity is therefore only present at the coarse level. On the other hand, completely unstructured AMR yields more flexibility in the geometry but at a greater cost since there is no structure to rely on. Between the two are three-based methods, where each node is a quadrant that can have four children (which in turn can have children,...) or none at all. That recursive definition provides both simplicity and efficiency.

One downside is that the domain represented by a quadtree (a tree where a node is a leaf or has four children, as presented above) is always cube shaped. We can circumvent this problem by using a forest of quadtrees to represent a large range of geometrical shapes.

The approach favored by *p4est* is the latter : it handles the dynamic management of a collection of adaptive quadtrees, later called a forest of quadtrees.

We can split the process used by *p4est* in a two-level decomposition of the domain : first the definition of the forest, which we later call the macro-mesh, and then the adaptive recursive definition of each individual quadtree, later called the micro-mesh.

For the macro-mesh, the domain $\Omega \subset \mathbb{R}^2$ is split into K conforming elements. Using K conforming elements allows us to map more general domains than with a single quadtree since each corner of the macro-elements can be shared by more than 4 elements. There exists a mapping to each element from a reference element by a smooth function $\phi_k : [0; 2^b]^2 \to \mathbb{R}^2$. Formally, we have that the domain Ω is split as :

$$\Omega = \bigcup_k \phi_k([0; 2^b]^2) \qquad\qquad 0 \leq k < K$$

We have an important thing to note here. It is an important feature of *p4est* to perform all computations related to the connectivity and neighborhood relations discretely (i.e. it uses an integer-based approach). This is why the reference element is $[0; 2^b]^2$. The main objective is to avoid floating-point arithmetic and the roundoff errors that might lead to errors in the topology. In their algorithms, the mappings ϕ_k are never used, except for the visualization and to encode the geometry that might be used for an external numerical application (such as ours). The consequence of having an integer-based system is that we cannot refine a quadtree indefinitely. Indeed, a quadrant cannot have a smaller length than 1 on the reference element. Thus, with $[0; 2^b]^2$, it is possible to have at most 2^{2b} quadrants per element of the macro-mesh. Equivalently, each quadtree cannot have a depth superior to b, and a quadrant a refinement level superior to b. In the *p4est* library, $b = 29$, which yields a maximum of $2.9\,10^{17}$ quadrants per quadtree (the number would even be higher in three dimensions). We can see that it is amply sufficient for our applications.

Let us also note that the macro-mesh cannot be changed dynamically and is shared among all processes. In practice, the number of quadtrees K is thus limited by local memory. The experiments in [4] go up to several million quadtrees.

Let us now turn to the micro-mesh. Based on a user-defined criterion, the refinement of the quadtree is often defined recursively. This creates a quadtree with the quadrants as leaves. Each quadrant of a given quadtree is then uniquely defined by the integer coordinates of its lower left corner $x, y \in \{0, 1, 2, ..., 2^b - 1\}$ on the referent element and its level of refinement, i.e. the level of the leaf corresponding to that quadrant in the quadtree. In the reference element, a quadrant of level l is a square of length 2^{b-l}.

The details of the refine algorithm used in *p4est* are not given here but can be found in [4]. Here, non conforming edges are allowed. As mentioned in the theory chapter, our application only allows 1-irregular meshes, i.e. two neighboring quadrants differ by at most one level of refinement. However, the refine function in *p4est* does not constraint the size relations for

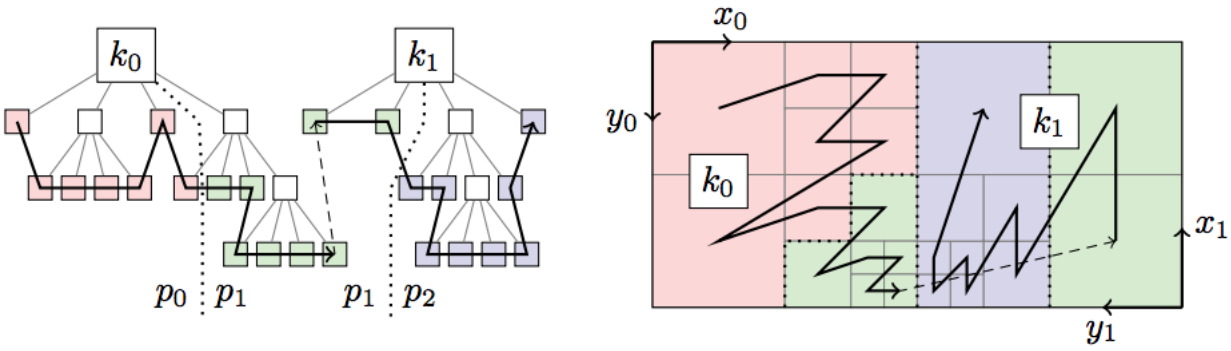


Figure 3.1: One-to-one correspondence between a forest of quadtrees (left) and a computational domain (right). There are two quadtrees k_0 and k_1 (corresponding to two elements in the macro-mesh). Going through the leaves of the forest from left to right creates a space filling curve (in black) and thus a total ordering of the quadrants. Source : [4].

neighboring quadrants (leading possibly to k-irregular meshes with $k > 1$). Fortunately, the library also contains a balance function that we can use to guarantee a 1-irregular mesh. Of course, the balance function acts both on quadrants in the same quadtree and on quadrants in different quadtrees that connect through an edge of the macro-mesh.

The micro-mesh, i.e. the collection of quadrants, can be dynamically changed (with the refine and coarsen functions) and is shared between all processes.

Tree-based AMR methods naturally lead to space filling curves. Indeed, if the global index of a quadrant is given by its order of appearance when looking at the forest of quatrees from left to right, then going quadrant by quadrant creates a curves that tends to fill the domain. An example of this is given in figure 3.1. On the left, we can see the forest which contains two quadtree (k_0 and k_1), distributed between three processes. We can also see that the refinement process has lead to quadrants with level 1, 2 and 3. On the right, the corresponding mesh is shown. Because we have different levels of refinement, the mesh is non conforming but it is still acceptable for our application since it is 1-irregular. Inspecting the leaves in the forest of quadtrees from left to right creates a space filling curve that can be used to define a global index for the quadrants. We can see that, within one quadtree, the space filling curve follows the orientation of its coordinate axes.

There are still three things we want to point out in this part : how to tell if a quadrant is touching the boundary of the computational domain, how to obtain the physical coordinates of the corners of any quadrant and give some explanation about the global numbering of nodes used in the spectral elements method.

Quadrants on the boundary

When we solve the system arising from SEM, it is fundamental to be able to tell whether a quadrant has one of its face on the boundary of the computational domain.

Let us first note that if a quadrant has a face on the boundary, then the corresponding face of the macro-mesh element is on the boundary.

In order to be able to tell about the neighboring quadtrees, there exists a function $k' = NO(k, f)$ that tells for a quadtree k that its neighbor through face f is k', as well as a function $f' = NF(k, f)$ that tells that k' is connected to k through face f'.

The convention used by *p4est* to say that quadtree k has face f on the boundary is :

$$NO(k, f) = k$$
$$NF(k, f) = f$$

This convention only prevents the (pathological!) configuration where a quadtree connects to itself periodically through a face that is rotated against itself and that never happens in our application. That enables us to tell which quadrants have a face on the boundary.

Physical coordinates of the corners

We are now looking at how to compute the physical coordinates of a quadrant. Let us first define the one dimensional linear mapping :

$$\phi_0(x) = 1 - x$$
$$\phi_1(x) = x$$

Let us then assume that the quadrant is located in the macro-mesh element k. Let us also denote the physical coordinates of the corners of element k by x_i^k, y_i^k for $i = 0, 1, 2, 3$. Let us finally assume that the integer-based coordinate of the lower left corner of the quadrant are x, y and it has a level of refinement l. In the reference element, the quadrant has therefore a length of :

$$h = 2^{b-l}$$

Let us then use our linear mapping above to compute the coordinates of the corners of the quadrant X_i, Y_i for $i = 0, 1, 2, 3$ as :

$$X_{i+2j} = \sum_{i=0}^{1}\sum_{j=0}^{1} x_{i+2j}^k \phi_i(\frac{x+ih}{2^b})\phi_j(\frac{x+jh}{2^b}) \qquad \text{for } i, j = 0, 1$$
$$Y_{i+2j} = \sum_{i=0}^{1}\sum_{j=0}^{1} y_{i+2j}^k \phi_i(\frac{x+ih}{2^b})\phi_j(\frac{x+jh}{2^b}) \qquad \text{for } i, j = 0, 1$$

Once we have the physical coordinates of any quadrants of our mesh, we can compute easily quantities such as the physical coordinates of the different local GLL nodes, the value of the jacobian,...

Global unique node numbering

Let us finally give some explanation about the globally unique numbering of the unknowns used in the spectral elements method. It is created in *p4est* by a structure named *lnodes*. The principle is to number in order the global GLL nodes encountered while going through the quadrants by the global index. If there are no hanging nodes, it is simple : when we get to a quadrant, we number the nodes that have still not been numbered in lexicographic order then go to the next quadrant. When we have hanging nodes, it is more difficult since they are not independent and therefore cannot get a global numbering. The principle is then to number the global nodes that have an influence of the hanging nodes in the quadrant.

All this information is contained in a large array called *element_nodes* in *lnodes*. Let us give an example for the sake of clarity. Figure 3.2 shows a mesh for a degree of interpolation $p = 2$ where we see in red the global nodes having an influence in the small quadrant close to the

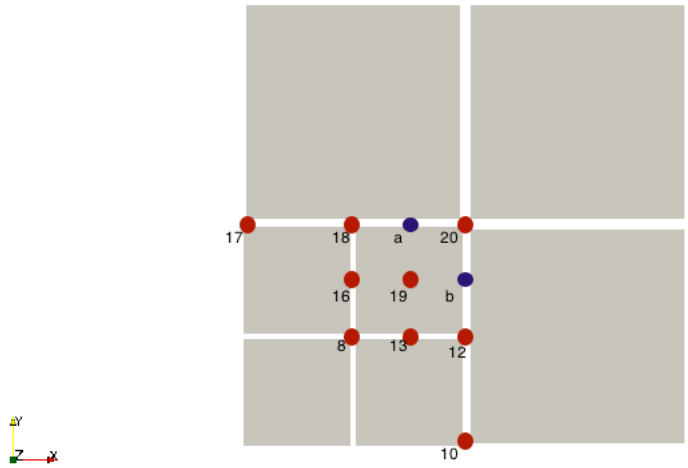


Figure 3.2: Global numbering of the nodes (in red) that have an influence in the small quadrant close to the center as well as the hanging nodes (in blue) for an interpolation of degree $p = 2$.

center. We can see that this quadrant has two hanging edges. The *element_nodes* array for this example is given by :

$$element\_nodes = \begin{bmatrix} 8 & 13 & 10 & 16 & 19 & 12 & 17 & 18 & 20 \end{bmatrix}$$

We can see that it numbers the global GLL nodes first in the x direction, then in the y direction and uses the global numbering to refer to hanging nodes. It is possible to do this since hanging nodes are always located on edges.

3.1.2 Handling the hanging nodes

Let us now look at how to practically handle hanging nodes. As presented before, the array *element_nodes* always uses global numbering to tell the nodes that have an influence in a quadrant. Therefore, we have to know which, if any, edges are hanging. This information, for each quadrant, is encoded in $face\_code$ in *lnodes*.

Decoding the face code

The function that we built to decode $face\_code$ returns $false$ if no edge is hanging while it returns *true* and fill an array of length 4 called *hanging_corner* if there are at least one hanging edge where :

$$\begin{aligned} hanging\_corner_i &= -1 && \text{if corner } i \text{ is not hanging} \\ &= a && \text{if } a \text{ is the non hanging corner corresponding to } i \end{aligned}$$

The $face\_code$ is a bitcode where the last two bits contain the position of the quadrant relative to its parent and the next two bits contain the status of the potentially hanging edges. Indeed, it is obvious that for any quadrant, at most two edges can be hanging.

For example, in figure 3.2, the $face\_code$ of the quadrant is 1111 since it is the fourth child of its parent and the two potentially hanging nodes are indeed hanging. The *hanging_corner* array resulting from decoding that face code is :

$$hanging\_corner = \begin{bmatrix} -1 & 3 & 3 & -1 \end{bmatrix}$$

Once we have the hanging corners, we can know which edges are hanging and if those edges constitute the first or the second part of the corresponding non hanging edge. For example in figure 3.2, the east edge is hanging and constitute the second part of the corresponding non hanging edge defined by global nodes 10 and 20.

Computing R^e_{ij}

All the information about the mesh computed above has two purposes : for us to be able to interpolate the different functions in each quadrant (scatter operation) and then to gather the result of the integration. As mentioned in the theory chapter, we never build the operator R^e_{ij} explicitly. Let us first present the scatter operation. There are three positions possible for a local GLL node :

1. The node is one of the four corners
2. The node is on an edge
3. The node is in the "interior", i.e. neither on an edge nor on a corner

Let us stress the fact that only the first two types of nodes can be hanging. Therefore, the first step every time we need to perform an interpolation is to load the "interior", since we are sure those nodes are not hanging. In figure 3.2, this corresponds to only one node (19).

The second step is to look at the edges. If an edge is not hanging, then we can load all nodes on this edge but the corners (since even on a non hanging edge, those can be hanging). In figure 3.2, this corresponds to node 13 for the south edge and node 16 for the west edge. For the hanging edges, we need to perform an interpolation of the values at the global nodes to obtain the values at the hanging nodes. We can do a one dimensional interpolation since hanging nodes and the global nodes that influence them are located along an edge which is a straight line. In figure 3.2, hanging nodes b is interpolated using global nodes 10, 12 and 20. Whether the hanging edge is the first or the second part of the non hanging edge is important since the interpolation will be different. Let us assume that the values at the global nodes are given by u_i^{glob} for $i = 0, ..., p$. Then the value at hanging node j, u_j^{loc}, is given by :

$$u_j^{loc} = \sum_{k=0}^{p} l_k \left(\frac{\xi_j}{2} - \frac{1}{2} \right) u_k^{glob} \qquad \text{if the hanging edge constitutes the first part}$$

$$u_j^{loc} = \sum_{k=0}^{p} l_k \left(\frac{\xi_j}{2} + \frac{1}{2} \right) u_k^{glob} \qquad \text{if the hanging edge constitutes the second part}$$

Where l_k denotes the Lagrangian polynomial associated with the k-th 1D GLL node. Since it is always the same interpolation (i.e. the coefficients of the linear combination do not depend on the quadrant) for all hanging edges, we built two matrices that perform the interpolation (one when the edge is the first part, D^1, and the other when the edge is the second part, D^2) and we use them every time we have a hanging edge.

$$D^1_{ij} = l_j \left(\frac{\xi_i}{2} - \frac{1}{2} \right)$$
$$D^2_{ij} = l_j \left(\frac{\xi_i}{2} + \frac{1}{2} \right)$$

For the example given in figure 3.2, we have for the east edge :

$$\begin{pmatrix} u_{12} \\ u_b \\ u_{20} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -0.125 & 0.75 & 0.375 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_{10} \\ u_{12} \\ u_{20} \end{pmatrix}$$

Here we can also do the corners and mark them as visited.

The third step is to look at all four corners and check whether we have visited them or not. Those we have not yet visited are not hanging and we can therefore load them. In figure 3.2, this corresponds to node 8.

The gather operation works in much the same way. We first deal with the interior where we are sure no nodes are hanging. Then, we look at the edges. If the edge is hanging, we use the transpose of the matrices D^1 and D^2 to go from local nodes to global ones. We end with the corners that have not yet been visited.

3.2 Multigrid structure

Let us now move on the structure used to handle the multigrid method. Is it the most important structure to build. If we define our structure in the right way, smoothing, restricting and prolonging are fairly straightforward.

Let us first note that throughout the program, we have two different global node numbering. One for the high degree interpolation, $lnodesP$, and one for the interpolation with $p = 1$ used in the multigrid, $lnodes1$. As expalined in the theory chapter, we therefore need functions to transfer the high degree residual from $lnodesP$ to $lnodes1$ and then to transfer the coarse grid correction from $lnodes1$ to $lnodesP$. The details of that transfer are not given here but the implementation from the theory is rather simple.

We will focus here on the structure we build to handle the multigrid using the global numbering used by $lnodes1$. For each level, we need to know :

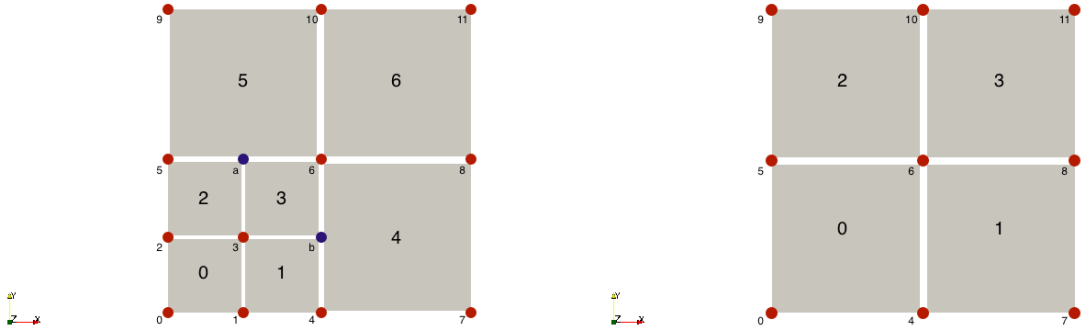
1. The number of nodes on that level
2. The number of quadrants on that level
3. Which nodes form the k-th quadrant
4. If a given quadrant has children on the upper level and if so, which quadrants it is
5. What quadrants are hanging on that level
6. The hanging information of the hanging quadrants

Several other arrays are also contained in the structure to store the value of the solution at that level, the right-hand side, some geometric factors, ...

Building the structure is a two-step process : we first fill the highest level (corresponding to the finest grid) and then we recursively fill the levels by the information contained in the upper level (we use level $lev + 1$ to fill level lev).

3.2.1 Filling the highest level

At the highest level, called $maxlev$, it is obvious that the number of nodes, N^{maxlev}, is the number of nodes in $lnodes1$, the number of quadrants, Q^{maxlev}, is the number of quadrants in the forest and the nodes for the k-th quadrant, $nodes_{i;k}^{maxlev}$, are given by $element\_nodes$ in $lnodes1$. There is no upper level so we do not need to care about the children (the array up^{maxlev} is set to -1).



(a) Mesh at level 2

(b) Mesh at level 1

Figure 3.3: Example of two meshes at level 1 and 2. We can see that when we go from level 2 to level 1, all quadrants that had a refinement level of 2 are replaced by their parent. Moreover, the parent of the quadrants with level of refinement 2 that were hanging at level 2 is not hanging at level 1. The global nodes are in red (with their global number) and the hanging nodes in blue. The global number of the quadrants is also given.

We then have an array containing boolean flags to see if quadrant k has hanging nodes. This is easy to see from the $face\_code$ (see above).

$$\begin{aligned} hang_k^{maxlev} &= 1 && \text{if quadrant } k \text{ contains hanging nodes} \\ &= 0 && \text{if quadrant } k \text{ does not contain hanging nodes} \end{aligned}$$

For quadrants where there is at least one hanging node, we decode the $face\_code$ using the function defined in the previous section and we store the information.

$$\begin{aligned} hang\_info_{i;k}^{maxlev} &= -1 && \text{if corner } i \text{ in quadrant } k \text{ is not hanging} \\ &= a && \text{if } a \text{ is the non hanging corner corresponding to } i \end{aligned}$$

3.2.2 Recursively filling the lower levels

Let us remember that the mesh at level lev is the mesh at level $lev+1$ where all quadrants with level of refinement $lev+1$ have been replaced by their parent. We can see an example in figure 3.3. At level 1, all quadrants with level 2 have been replaced by their parent.

An important thing to note here is that if a quadrant with level of refinement $lev+1$ is hanging at level $lev+1$, then the corresponding parent at level lev is not hanging. This is a direct consequence of the fact that two neighboring quadrants differ by at most one level in their refinement levels. In the example in figure 3.3, we can indeed see that the parent of the quadrants that were hanging at level 2 is not hanging at level 1.

The first thing to do is to compute the number of quadrants at level lev. If, at level $lev+1$, there are q^{lev+1} quadrants with a level of refinement of $lev+1$, then it is given by :

$$Q^{lev} = Q^{lev+1} - \frac{3}{4} q^{lev+1}$$

Then, we go quadrant by quadrant following their global ordering (given by the space-filling curve). If the current quadrant has a refinement level inferior or equal to lev, then we copy the information from level $lev+1$ to level lev. However, if we reach a quadrant with a refinement

level of $lev + 1$, then we have to coarsen. An important thing to note is that, if we visit the quadrants in their global order given by the space filling curve, then the four quadrants we have to replace by their parent when we coarsen are always positioned consecutively. Therefore, we can fill the *up* array easily. For the example in figure 3.3, we have :

$$up_0^1 = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

The *up* array for the other quadrants are filled with -1 since they do not have children. We can also use the fact that the children are placed consecutively and the fact that when we coarsen we are sure that no node is hanging to easily fill the *nodes* array. In our example this yields :

$$\begin{aligned} nodes_0^1 &= \begin{bmatrix} nodes_{0;0}^2 & nodes_{1;1}^2 & nodes_{2;2}^2 & nodes_{3;3}^2 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 4 & 5 & 6 \end{bmatrix} \end{aligned}$$

The last thing to handle for that set of quadrants is the hanging information. Since we coarsen, the parent is not hanging. In our example, that just means setting $hang_0^1 = 0$.

After having gone through all the quadrants, we still need to compute the number of global nodes at level *lev*. The idea is to fill an array with all the nodes we see when we go through the quadrants (which is therefore of length $4Q^{lev}$), then to sort this array and count the number of different global nodes we have. That also allows us to set up a mapping between the numbering of the nodes at level *lev* and the global numbering given in $lnodes1$. That mapping is very useful when we have to perform an operation on the nodes at a given level (for example, during the smoothing).

Let us finally say that the minimum level ($lev = 0$) is the macro-mesh. It is a conforming mesh and on it, we solve the linear system of equation exactly using the Lapack library (see [31] for information).

3.3 Fine preconditioner

Let us finally give a few explanations about the fine preconditioner and how it is implemented. Practically, in addition to the information we already have about a quadrant (the physical coordinate of its corners, the global number of the local nodes in it, if some nodes are hanging,...), we need three things to be able to compute the fine preconditioner :

1. The matrices V, V^{-1} and Λ such that $L = V^{-1}\Lambda V$
2. For each quadrant, the global number of each neighbor
3. For each quadrant e, the orientation in the neighboring quadrant d of the face that connects d to e.

The matrices V, V^{-1} and Λ are computed using the Lapack library (see [31] for information) once we have built L, which is fairly simple from the theory.

The additional information needed (items 2 and 3 in the list above) is contained in *neighbors*, an array we build just for this purpose.

In this section, we will first explain how to build *neighbors* and then how to compute the residual in the overlaps (i.e. the parts of the subdomains that are not the quadrant) in the four different cases we can encounter.

3.3.1 Building *neighbors*

The easiest way to know which quadrant is connected to what quadrant is to list every (oriented) edge in our mesh and sort that list so that pairs of edges are consecutive. Because we have non conforming meshes, we have to be careful though. Thanks to the fact that we only handle 1-irregular meshes, we can say that a global edge consists of one, two or three oriented edges. One if we are on the boundary, two if the edge connects two quadrants and there are no hanging nodes on this edge, and three if we have hanging nodes on this edge.

Thus, the first thing to do is to go through the list of quadrants and for each quadrant, build four edge structures. The structure contains the end points of the edge (oriented counter-clockwise for the quadrant), whether the edge contains hanging nodes and if so, if it is the first or the second part of the larger edge. It also contains the global number of the quadrant it is in and its disposition (0 for a west edge, 1 for an east edge, 2 for a south edge, and 3 for a north edge).

This method has still one problem that arises from hanging nodes. The best way to show is with an example. Let us assume we have the same mesh as the one in figure 3.3a. When we are in quadrant 0, the north edge has end points 6 and 3. And when we are in quadrant 2, the east edge has also endpoints 3 and 6. Therefore, when we will sort the list of edges, these two will pair and we will wrongly thing that there is a connecting edge between quadrants 1 and 2. To circumvent this problem, we change the end points when we have hanging nodes. We chose negative numbers $(-1, -2, -3, -4)$ so that they do not collide with the global numbering. For example, in quadrant 1, we say that the end points of the north edge has end points -2 (because hanging node b is on the east edge of the parent) and 3. In quadrant 2, we say that the east edge has end points 3 and -4 (since hanging node a is on the north edge of the parent). That will allow those edges to be listed next to their right pair after sorting. However, it is important to note that we do not change the end points of the hanging edges themselves (for example the east edge in quadrant 1 and the north edge in quadrant 2). Indeed, we want them to pair with the edge that form the other part of the larger edge and for that we need them to have the same end points.

Once we have built the list of edges, we can sort them. We will sort them first by the end points with the lowest value, then by the endpoints with the highest value and finally by their hanging information (0 if the edge is not hanging, 1 if the edge is the first part of the larger edge and 2 if it is the second). Once sorted, we can go through the list to form *neighbors*. We know that if three consecutive edges have the same end points, then the first is the larger (non hanging) edge, the second is the first part of the hanging edge and the third is the second part. If two consecutive edges have the same end points, then they connect two quadrants through a non hanging edge. If we have a lone edge, then it is part of the boundary.

Let us define $neighbors_{i;k}$ for $i = west, east, north, south$ to be the part of *neighbors* related to quadrant k and edge i. Then, it is given by (with $j \in \{west, east, north, south\}$):

$$
\begin{aligned}
neighbors_{i;k} &= \begin{bmatrix} -1 & -1 & -1 \end{bmatrix} && \text{if edge } i \text{ in quadrant } k \text{ lies on the boundary} \\
&= \begin{bmatrix} a & -1 & j \end{bmatrix} && \text{if non hanging edge } i \text{ in quadrant } k \text{ connects } k \text{ to } a \\
& && \quad a \text{ is connected to } k \text{ by non hanging edge } j \\
&= \begin{bmatrix} a & b & j \end{bmatrix} && \text{if non hanging edge } i \text{ in quadrant } k \text{ connects } k \text{ to } a \text{ and } b \\
& && \quad a \text{ and } b \text{ are connected to } k \text{ by hanging edge } j \\
&= \begin{bmatrix} a & a & j \end{bmatrix} && \text{if hanging edge } i \text{ in quadrant } k \text{ connects } k \text{ to } a \\
& && \quad a \text{ is connected to } k \text{ by non hanging edge } j
\end{aligned}
$$

We can see that any neighbor can be of four types : the boundary, a regular neighbor through a non hanging edge, two neighbors through a non hanging edge, a neighbor through a hanging

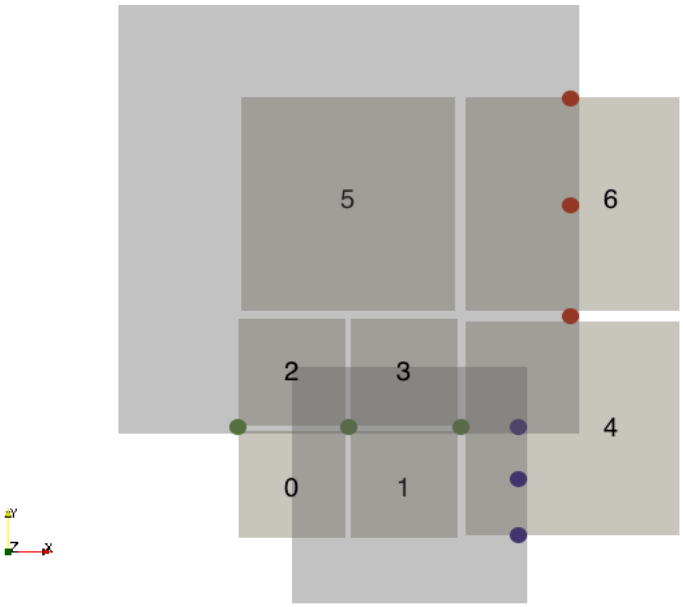


Figure 3.4: Example of a mesh for $p = 2$ and the subdomains (in light grey) associated with quadrants 1 and 5. It presents the latter three kinds of neighbors quadrants can have : one neighbor through a non hanging edge (in red for quadrant 5), two neighbors through a non hanging edge (in green for quadrant 5) and one neighbor through a hanging edge (in blue for quadrant 1).

edge. All those types have to be treated separately when we compute the local residual.

Let us finally give two examples. In figure 3.3a, *neighbors* associated with the east edge in quadrant 1 is :

$$neighbors_{east;1} = \begin{bmatrix} 4 & 4 & west \end{bmatrix}$$

The one associated with the south edge in quadrant 5 is :

$$neighbors_{south;5} = \begin{bmatrix} 2 & 3 & north \end{bmatrix}$$

3.3.2 Computing the local residual

Let us now look at how to compute the local residual on a given subdomain. The part that is constituted by the quadrant is easy enough : we just look at the nodes in the quadrant and use, if necessary, the R^e_{ij} operator (see the first section of this chapter for more information). It is trickier for the overlaps (i.e. the parts in the subdomain that is not a full quadrant but the layer of GLL nodes in the neighbors). As explained above, there are four different types of neighbors and all of them have to be treated individually.

Figure 3.4 shows the latter three kinds of neighbors a quadrant can have. We will use this example to illustrate the different cases.

The neighbor is the boundary

We know that edge i of quadrant k lies on the boundary if the first entry of $neighbors_{i;k}$ is -1. This is the easiest case to treat. As said in the theory chapter, we just fill the overlap using a linear interpolation from inside the quadrant. When we gather the fine scale correction, we of course do not take this overlap into account.

One neighbor through a non hanging edge

That is another fairly easy case. Because of our assumptions, we can copy the values of the residual of the neighboring quadrant in the overlap. Indeed, in that case, the nodes in the overlap coincide with the local nodes of the neighboring quadrant. An example of such a case is given for quadrant 5 in figure 3.4 (nodes in red). We can see that quadrant 6 is its neighbor through its (non hanging) east edge. We can also see that the red nodes in the overlap coincide with local GLL nodes of quadrant 6.

There is however one problem with this approach. It does not take into account the fact that edges in quadrant 6 can be hanging. Indeed, in general, the edge between quadrants 6 and 4 could be a hanging edge. Then, it is not possible to just copy the values. Since it grows the complexity a lot, our application does not handle this case. We will see in the results chapter that the fine preconditioner works anyway.

Two neighbors through a non hanging edge

Here, even with our assumptions, we have to interpolate the value of the residual since nodes do not coincide. An example of this situation is given in figure 3.4 for quadrant 5 (nodes in green). For this special case of $p = 2$, we can see that the nodes in the overlap do coincide with nodes in quadrant 2 and 3 but for higher degrees it is not true anymore. We therefore have to interpolate the value of the residual at the right place. It is here a two dimensional interpolation (where the interpolation for hanging nodes is one dimensional). We can note that it is always the same interpolation and we can therefore build a matrix beforehand, called *two_to_one*, and apply it every time we encounter this case.

The same remark as before applies here : we do not treat every case perfectly. For example, we do not take into account the fact that the edge between quadrant 3 and 4 is hanging for the two dimensional interpolation. This yields a small error when we compute the residual for such an overlap.

When we have to perform the gather operation, we use the transpose of *two_to_one* to distribute the fine scale correction to nodes in quadrants 2 and 3.

One neighbor through a hanging edge

This last case also needs us to interpolate the value of the residual because, here again, nodes do not coincide with local nodes in the neighboring quadrant. An example of this case is given in figure 3.4 for quadrant 1 (nodes in blue). It is here clear that the nodes in the overlap do not coincide with local nodes in quadrant 4. Once again, we have to perform a two dimensional interpolation to get the values of the residual. It is always the same interpolation and we build the matrix *one_to_two* to perform it. The transpose of this matrix is used when we have to gather the fine scale correction.

As before, we do not handle the extra cases : for this example, that the edge between quadrant 4 and 6 could be hanging. We will see in the next chapter the consequences of this approximation on the number of iterations of PCG.

Chapter 4

Results and discussion

In this chapter, we will put the algorithms presented in the theory chapter to the test and present the results. The implementation of the algorithms consists of a code written in C of more than 8000 lines which leverages the p4est library (see the implementation chapter for information) for the mesh generation and refinement as well as the Lapack library (a widely used linear algebra library, see [31]) to solve linear systems and diagonalize matrices. We will look at the experimental results and compare them to the theory developed. The code written can in principle handle any order of interpolation and any geometry but in practice most tests have been performed for $p = 2, 4, 6, 8$ and on $\Omega = [-1; 1]^2$ which allowed us to have analytic solutions to the problems we investigated.

This chapter is divided into several sections. The first looks at the geometric multigrid preconditioner. As explained in the theory chapter, it cannot be used as the only preconditioner since it has a kernel but we can use it as an iterative solver for $p = 1$. This is done to verify an important property of the geometric multigrid methods : the h-independent convergence. In this section, we also look at the influence of hanging nodes on the solver.

The second section focuses on the preconditioned conjugate gradients with only the fine preconditioner. This aims at looking at the properties of the overlapping Schwarz preconditioner presented in the theory chapter. We will first test it on regular elements, where the preconditioner is optimal, then look at what happens when we have a mesh with distorted elements. We will afterwards move on to non conforming meshes and look at the influence of having hanging nodes on the fine preconditioner. Finally, we will increase the degree of the interpolation and observe how the number of iterations of PCG evolves.

The third section looks at the PCG with the two scale preconditioner : both the coarse grid correction computed by the multigrid solver and the fine scale correction computed by the overlapping Schwarz method. The addition of the coarse scale preconditioner should provide a convergence independent of the number of quadrants. As before, we will first test the algorithms on a regular mesh, where it should perform very well. Then, we will see how it fares on meshes with distorted elements and what happens when we increase the distortion. We will thereafter consider non conforming meshes obtained through adaptive mesh refinement. We will check that here also we have a convergence that is independent of the number of quadrants and how the number of iterations of PCG compares to the case of conforming meshes. Then, we will increase the degree of the interpolation on conforming meshes and look at what happens to the number of iterations. Finally, we will emphasize the importance of being able to handle higher orders of interpolation. Given a wanted accuracy on the L^2-norm of the error, we will present the best polynomial order to use for a given problem and a given mesh.

The chapter ends on a partial conclusion which summarizes the results presented before.

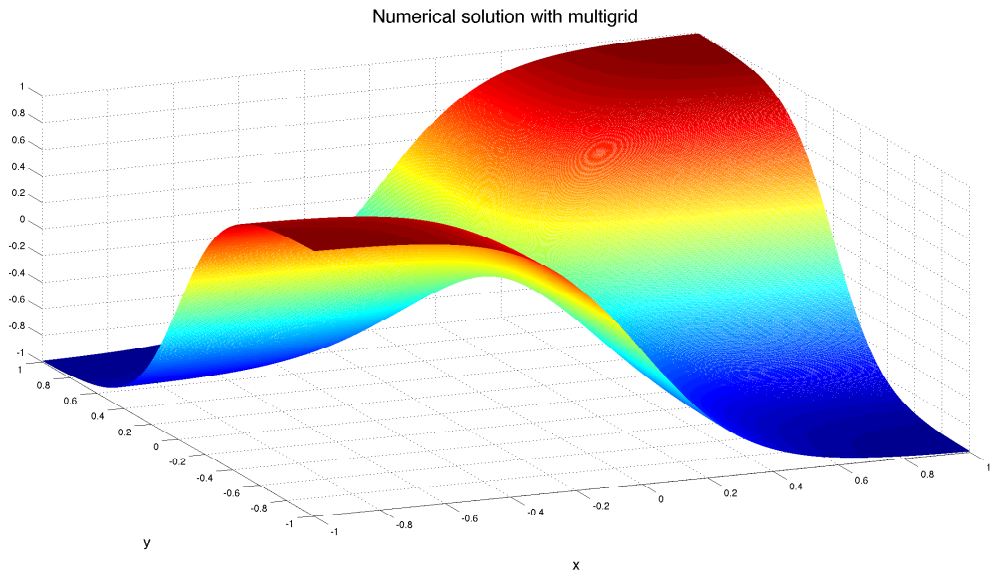


Figure 4.1: Numerical solution using the multigrid solver of $\nabla^2 u = f$ for $f = -2\tanh(3x)\tanh(3y)(18 - 9\tanh^2(3x) - 9\tanh^2(3y))$

4.1 Multigrid

In this section, we will test the coarse part of the preconditioner : the multigrid solver. This will be done in two steps. First, we will verify a well known property of the multigrid solvers : the h-independent convergence. We will also compare the number of iterations needed while varying key parameters of the model. Those tests will be performed on various meshes. The second part will focus on the influence of hanging nodes on the numerical solution.

Let us before all present a type of numerical solution that can be obtained using the multigrid solver. Figure 4.1 shows an example of the numerical solution computed. We can see that even with $p = 1$, we have a good approximation.

4.1.1 h-independent convergence

Let us first verify that our geometric multigrid solver has the required property and that the same number of iterations is needed to obtain a given accuracy, however small the elements. We will use the model problem throughout this section with the same right hand side. For all the tests below, the domain will be : $\Omega = [-1; 1]^2$. We will solve :

$$\nabla^2 u = -\frac{\pi^2}{2}\cos(\frac{\pi}{2}x)\cos(\frac{\pi}{2}y) \qquad \text{on } \Omega \tag{4.1}$$

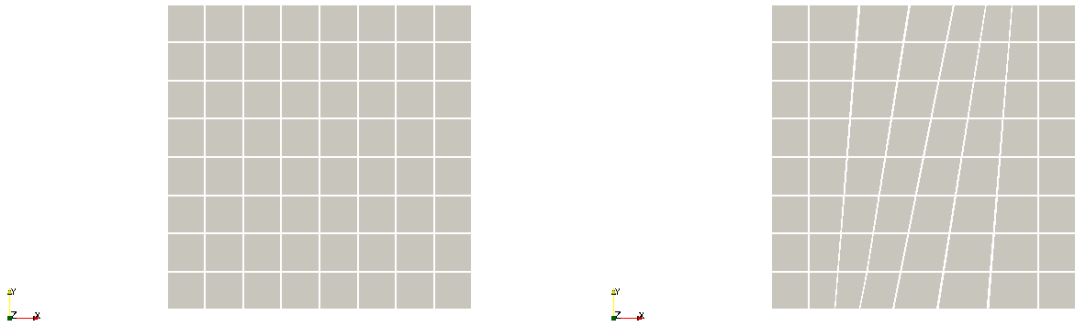
$$u = 0 \qquad \text{on } \Gamma \tag{4.2}$$

It is easy to see that for the given domain, we have an analytic solution :

$$u(x, y) = \cos(\frac{\pi}{2}x)\cos(\frac{\pi}{2}y))$$

Let us now explain how we define the error. We will look at the absolute difference between the value of the approximation and the value of the analytic solution at the global nodes and take the maximum. Formally, we have that the error after iteration k, e_k is :

$$e_k = \max_i |u(x_i, y_i) - u_i^k|$$



(a) Regular mesh

(b) Crooked mesh

Figure 4.2: The two macro-meshes that will be refined during the tests for the multigrid solver. We have one regular mesh (left) where the elements are squares and one crooked mesh (right) where the elements are slightly distorted.

Where u_i^k is the value of our approximation at the global node i after iteration k. Since $u_i^0 = 0$ for all i, it is clear that $e_0 = 1$.

Figure 4.2 shows the two macro-meshes that we will refine during the tests. Some refinements will be uniform and some will be so that we have the presence of hanging nodes. We can note that even for the crooked mesh, the elements are not really distorted. It does not matter since we are only testing the h-independent convergence. Having a mesh with elements that are more distorted will only influence the accuracy of the approximation and not how the algorithm solve the linear system we want to solve.

Let us start with a simple V-cycle on the regular mesh (figure 4.2a) that we will refine uniformly. We will compare the errors when we increase the number of degrees of freedom and for different ν_1 and ν_2. The sum of the two smoothing parameters is chosen to be constant so that we have the same number of Jacobi iterations for all pairs ν_1 and ν_2. Here, we chose the sum to be equal to four.

The results are shown in table 4.1. We can see that we indeed have an h-independent convergence of the solver. For every pair of the smoothing parameters, at least for the first few iterations, the error e_k is identical for all values of N. Except for the pair $\nu_1 = 4$ and $\nu_2 = 0$, each iteration roughly decrease the error by one decimal point.

We can also note that the values of the parameters influence the convergence. For this particular problem and this particular mesh, the values $\nu_1 = 0$ and $\nu_2 = 4$ seem to be the best as the error is smaller after the same number of iterations than for the other pairs. The fewer post smoothing iterations (ν_2) we do, the slower the convergence. This is true for $\nu_2 = 1$ where the error on the finest mesh after six iterations is a hundred times larger than on the same mesh after the same number of iterations for $\nu_2 = 4$, and it is clearer still for $\nu_2 = 0$ where the error is a thousand times larger after six iterations on the finest mesh.

We have to note that even tough the errors are identical for all meshes at first, we have a difference after a few iterations. This can be explained by the fact that our geometric multigrid algorithm actually solves a linear system whereas the error is measured as the difference between the analytic solution and the solution of the linear system. Thus, even if we solved the linear system exactly, we would still have an error and that error should decrease as the number of degrees of freedom increases. This is indeed what we observe here. For example, for the mesh with $N = 2.6\,10^5$, we can see that after six iterations, we have almost converged and that the error stays around $3.14\,10^{-6}$. Even if we did several more iterations, the error would not decrease significantly. That is because we have the solution of the linear system and the error is only due

N	$2.6\,10^5$	$1.1\,10^6$	$4.2\,10^6$	$1.7\,10^7$	$6.7\,10^7$
	$\nu_1 = 2$	$\nu_2 = 2$			
e_1	3.56e-02	3.56e-02	3.56e-02	3.56e-02	3.56e-02
e_2	1.34e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03
e_3	5.42e-05	5.66e-05	5.72e-05	5.73e-05	5.73e-05
e_4	3.11e-06	2.90e-06	3.45e-06	3.59e-06	3.62e-06
e_5	3.30e-06	9.48e-07	3.62e-07	3.50e-07	3.85e-07
e_6	3.17e-06	8.14e-07	2.26e-07	8.26e-08	5.23e-08
	$\nu_1 = 3$	$\nu_2 = 1$			
e_1	3.70e-02	3.71e-02	3.71e-02	3.71e-02	3.71e-02
e_2	1.62e-03	1.63e-03	1.63e-03	1.63e-03	1.63e-03
e_3	1.04e-04	1.06e-04	1.07e-04	1.07e-04	1.07e-04
e_4	1.10e-05	1.22e-05	1.27e-05	1.29e-05	1.29e-05
e_5	4.47e-06	2.20e-06	1.96e-06	2.10e-06	2.13e-06
e_6	3.34e-06	1.00e-06	4.38e-07	3.53e-07	3.88e-07
	$\nu_1 = 1$	$\nu_2 = 3$			
e_1	3.57e-02	3.57e-02	3.57e-02	3.57e-02	3.57e-02
e_2	1.29e-03	1.29e-03	1.29e-03	1.29e-03	1.29e-03
e_3	4.66e-05	4.89e-05	4.95e-05	4.96e-05	4.97e-05
e_4	1.53e-06	1.53e-06	2.10e-06	2.24e-06	2.28e-06
e_5	3.08e-06	7.31e-07	1.43e-07	1.17e-07	1.51e-07
e_6	3.14e-06	7.83e-07	1.95e-07	4.80e-08	1.13e-08
	$\nu_1 = 4$	$\nu_2 = 0$			
e_1	4.55e-02	4.55e-02	4.55e-02	4.55e-02	4.55e-02
e_2	3.26e-03	3.26e-03	3.26e-03	3.26e-03	3.26e-03
e_3	4.64e-04	4.71e-04	4.71e-04	4.71e-04	4.71e-04
e_4	9.24e-05	9.60e-05	9.65e-05	9.67e-05	9.67e-05
e_5	1.74e-05	2.04e-05	2.12e-05	2.14e-05	2.14e-05
e_6	6.17e-06	3.86e-06	4.55e-06	4.74e-06	4.78e-06
	$\nu_1 = 0$	$\nu_2 = 4$			
e_1	3.58e-02	3.58e-02	3.58e-02	3.58e-02	3.58e-02
e_2	1.29e-03	1.29e-03	1.29e-03	1.29e-03	1.29e-03
e_3	4.53e-05	4.77e-05	4.82e-05	4.84e-05	4.84e-05
e_4	1.17e-06	1.31e-06	1.89e-06	2.03e-06	2.07e-06
e_5	3.03e-06	6.80e-07	9.14e-08	7.66e-08	1.12e-07
e_6	3.13e-06	7.76e-07	1.87e-07	4.04e-08	3.62e-09

Table 4.1: Errors after k iterations of a V-cycle (e_k) for the regular mesh uniformly refined to have N degrees of freedom and for different values of the parameters ν_1 and ν_2

N	$2.6\,10^5$	$1.1\,10^6$	$4.2\,10^6$	$1.7\,10^7$	$6.7\,10^7$
	$\nu_1 = 2$	$\nu_2 = 2$			
e_1	3.64e-02	3.64e-02	3.64e-02	3.64e-02	3.64e-02
e_2	2.56e-03	2.49e-03	2.46e-03	2.44e-03	2.43e-03
e_3	7.41e-04	6.33e-04	5.80e-04	5.54e-04	5.41e-04
e_4	6.15e-04	3.15e-04	1.62e-04	1.35e-04	1.28e-04
e_5	5.96e-04	3.01e-04	1.52e-04	7.67e-05	4.76e-05
e_6	5.91e-04	2.98e-04	1.50e-04	7.50e-05	3.77e-05

Table 4.2: Errors after k iterations of a V-cycle (e_k) for the crooked mesh uniformly refined to have N degrees of freedom and for $\nu_1 = 2$ and $\nu_2 = 2$

Meshes	No hanging nodes	Figure 4.3a	Figure 4.3b
N	$4.2\,10^6$	$7.3\,10^6$	$7.0\,10^7$
e_1	3.56e-02	3.56e-02	3.56e-02
e_2	1.34e-03	1.34e-03	1.34e-03
e_3	5.72e-05	5.72e-05	5.74e-05
e_4	3.45e-06	3.47e-06	3.64e-06

Table 4.3: Errors after k iterations of a V-cycle (e_k) for a mesh without hanging nodes and the two meshes presented in figure 4.3. Each mesh has N degrees of freedom and the smoothing parameters were $\nu_1 = 2$ and $\nu_2 = 2$.

to the discretization. If we refine the mesh and go to $N = 6.7\,10^7$, then we can get smaller errors (of the order of 10^{-8}).

Let us now explore the results for the crooked mesh (figure 4.2b). Here also, we should expect an h-independent convergence. We only show the results for $\nu_1 = 2$ and $\nu_2 = 2$ but the same commentary applies for the other pairs. The results can be seen on table 4.2

We can see that the for the first few iterations, the error e_k is independent of the mesh. However, the note we made earlier is much clearer here. Because the mesh is not regular, the effect of discretization are more important and therefore we will not reach the same accuracy than we did before. That is why after six iterations, the less refined grid ($N = 2.6\,10^5$) still has an error of $5.91\,10^{-4}$. More iterations will not have a great impact on the solution since the error is mostly due to the discretization.

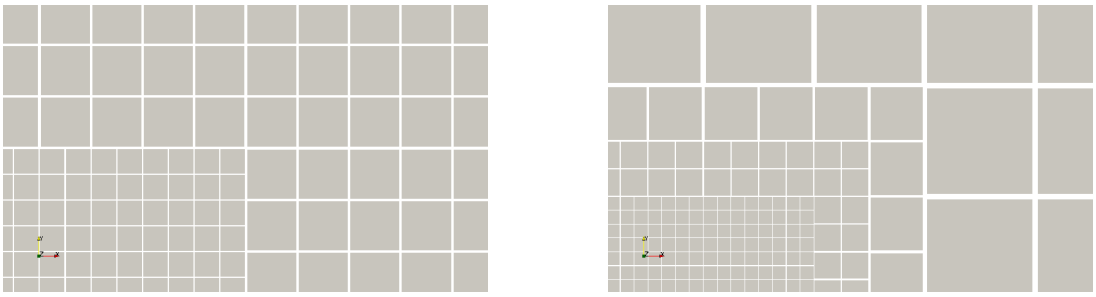
4.1.2 Influence of hanging nodes

We will now investigate the influence of hanging nodes on our solution. We will present the results only for the regular mesh but the tests have been performed on both and the same conclusions apply to the crooked mesh.

We will compare the values of the error between one mesh with no hanging nodes, one where we only have refined the lower left part of the domain once, and one where we have a rapid transition between two parts of different refinement level (which will occur often in AMR).

Figure 4.3 presents the latter two meshes. For the mesh in figure 4.3b, we have refined thrice more in a certain region than in the adjacent one. Since we do not allow adjacent quadrants to be more than one level apart, we obtain a "layer" where the levels of the quadrants is rapidly changing. In order to compare solutions, we made sure that the largest quadrants in all three meshes had the same size. This means that the meshes with hanging nodes have a lot more degrees of freedom.

Table 4.3 shows the results for the three different meshes using a V-cycle and with the smoothing parameters $\nu_1 = 2$ and $\nu_2 = 2$. Here again, we can see that the convergence is h-independent and that one iteration gives us roughly one more decimal. The presence of hanging



(a) Two levels (b) Levels rapidly changing

Figure 4.3: Zoom on a certain part of the two meshes containing hanging nodes that will be used to test the multigrid solver. We have one mesh where we only have refined a part of the domain once more than the rest (left) and a mesh where we have refined a part thrice more than the rest which results in levels changing rapidly (right).

nodes has no influence on the error we observe after a given number of iterations. The same result is observed with all pairs of the smoothing parameters and with other meshes.

This will be important in the next sections when we will use our multigrid solver as a preconditioner. Indeed, we will see that it is the coarse correction that allow for h-independent convergence.

4.2 Fine preconditioner

Let us now move on to the fine part of the preconditioner : the overlapping additive Schwarz preconditioner. We will test it by using the preconditioned conjugate gradients method described earlier but, for now, the preconditioner will only consist of the fine part (i.e. $P = P^f$).

As in the previous section, we will perform the tests in two parts : first, we will use meshes with elements that are distorted or not but with no hanging nodes. Then, we will see how the fine preconditioner performs in the presence of hanging nodes also for meshes that are distorted or not. Here, of course, we will use interpolations of higher degree. Typically, the tests will be performed for $p = 2, 4, 6, 8$.

4.2.1 No hanging nodes

Let us first present the problem we will use throughout this section. The forcing term will be chosen more oscillatory than in the previous part since we use interpolations of higher degree. As before, the domain is : $\Omega = [-1; 1]^2$ and Γ is the boundary. The problem is :

$$\nabla^2 u = -8\pi^2 \sin(2\pi x) \sin(2\pi y) \qquad \text{on } \Omega \tag{4.3}$$

$$u = 0 \qquad \text{on } \Gamma \tag{4.4}$$

This problem has an analytic solution and it is easy to convince oneself that this solution is given by :

$$u(x, y) = \sin(2\pi x) \sin(2\pi y)$$

Figure 4.4 shows the numerical solution to the problem above for $p = 2$ and $1.0\,10^6$ degrees of freedom for a regular mesh. We can note that it is exactly the same number of degrees of freedom as if we had refined uniformly once more and used an interpolation of degree $p = 1$. Let us then compare how the two approximations perform. We solved the problem for $p = 1$ with

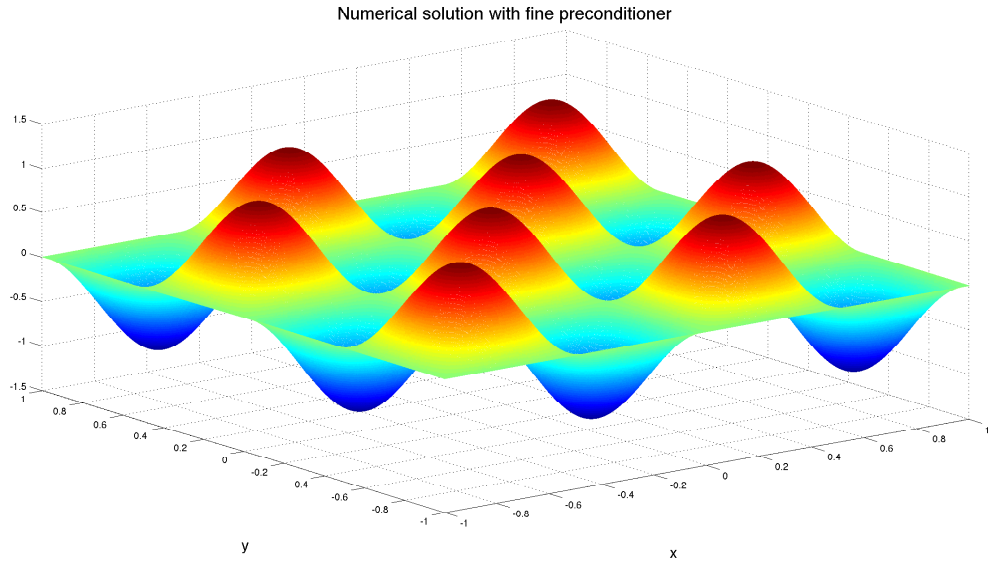


Figure 4.4: Numerical solution to problem 4.3 using an interpolation of order $p = 2$ and $1.0\,10^6$ degrees of freedom on a regular mesh with no hanging nodes.

our multigrid solver and the problem for $p = 2$ with the PCG and the fine preconditioner. Let us denote u_i^j as the value of the approximation for $p = j$ at node i. We have that :

$$e^1 = \max_i |u_i^1 - u(x_i, y_i)| = 5.02\,10^{-5}$$
$$e^2 = \max_i |u_i^2 - u(x_i, y_i)| = 1.01\,10^{-9}$$

We can see that with the same number of degrees of freedom, an approximation using $p = 2$ is much more accurate. This is because the solution is really smooth and is better approximated using a higher order interpolation than a bilinear interpolation on smaller quadrants. This is one example of why we want to use higher order interpolations.

Regular meshes

Let us now move on to the comparison for different degrees of the number of iteration needed to reach a given accuracy as a function of the number of quadrants. We will take our regular mesh and uniformly refine it. Then, for $p = 2, 4, 6, 8$, we will see how many iterations are needed to reach a given error on the norm of the residual. Let us denote r_k the residual after iteration k of the preconditioned conjugate gradients. Of course, since our initial guess is zero, we have that $r_0 = b$ (since we are solving the linear system $Au = b$). For the following tests, our stopping criterion is given by :

$$\frac{||r_k||_2}{||r_0||_2} < 10^{-3}$$

Figure 4.5 shows the results. To put the data in perspective, we also have to show the number of degrees of freedom. Indeed, for a given number of quadrants, the higher degree the interpolation is, the more nodes we have. Table 4.4 contains the number of nodes for each mesh and for each degree p.

We can see that, even without the coarse preconditioner, we are solving the system in a small number of iterations compared to the number of degrees of freedom. For example, we only do

Number of quadrants	16^2	32^2	64^2	128^2	256^2
$p = 2$	$1.1\ 10^3$	$4.2\ 10^3$	$1.7\ 10^4$	$6.6\ 10^4$	$2.6\ 10^5$
$p = 4$	$4.2\ 10^3$	$1.7\ 10^4$	$6.6\ 10^4$	$2.6\ 10^5$	$1.1\ 10^6$
$p = 6$	$9.4\ 10^3$	$3.7\ 10^4$	$1.5\ 10^5$	$5.9\ 10^5$	$2.4\ 10^6$
$p = 8$	$1.7\ 10^4$	$6.6\ 10^4$	$2.6\ 10^5$	$1.1\ 10^6$	$4.2\ 10^6$

Table 4.4: Number of degrees of freedom for a regular mesh with different number of quadrants and for different degrees of interpolation.

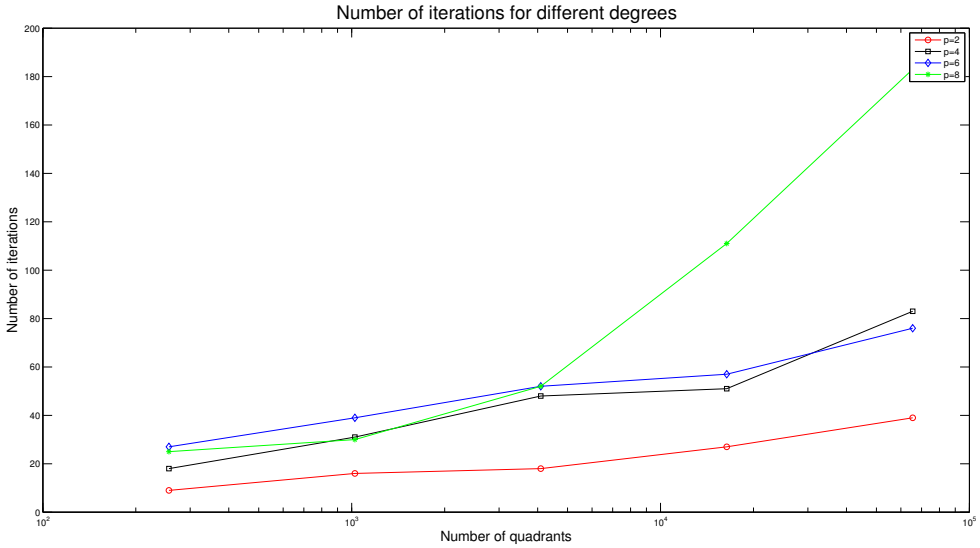
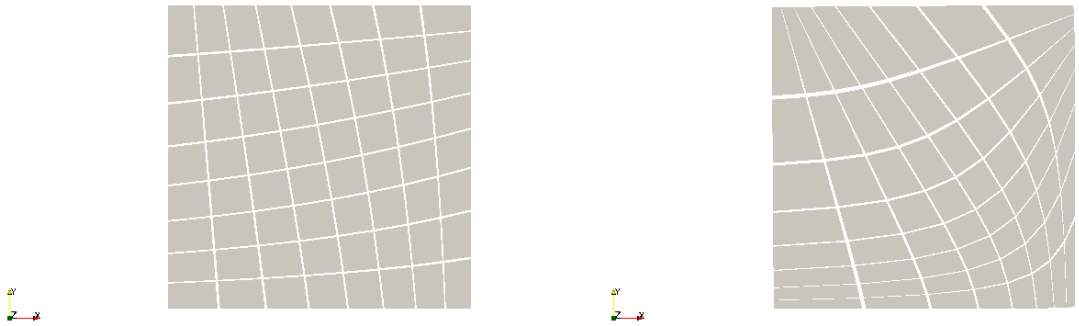


Figure 4.5: Number of iterations of PCG with only the fine preconditioner for different degrees p of interpolation as a function of the number of quadrants in a regular mesh.



(a) Mesh with $a = 1.1$ (b) Mesh with $a = 1.4$

Figure 4.6: Examples of the meshes used for the tests of the fine preconditioner with distorted elements. The meshes are created using GMSH with its progression tool. The key parameter is the common ratio a that we will be increasing progressively.

about 80 iterations to solve the system with $2.4\,10^6$ degrees of freedom and with interpolations of degree $p = 6$.

We can also see that for every degree, the number of iterations increases when we refine the mesh. This is to be expected since the information from the boundaries has to go through more quadrant before propagate to the entire domain. Asymptotically, the number of iterations is expected to double as the number of quadrants is multiplied by four (i.e. the mesh size is divided by two). We can see that it is not yet the case here.

A last remark we can make is that the number of iterations tends to increase when the degree of the interpolation increases. This is especially true for the finest mesh where we need 183 iterations for $p = 8$ where we only need 39 iterations for $p = 2$. This can be explained by the fact that the size of the overlap decreases when p grows. As mentioned in [32], this issue would be fixed if we imposed a constant overlap.

Meshes with distorted elements

Let us now move on to meshes that are not regular anymore. Let us remember that when we developed the fine preconditioner, we assumed that the elements were rectangular which allowed us to compute the analytic solution to the problem. This part explores the influence of having distorted elements on the number of iterations needed to obtain a given accuracy.

To control the deformation, we will continually deform the regular mesh using the progression tool of GMSH (information about GMSH can be found in [33], or in [34]). The deformation is performed using a geometric progression, with the common ratio a as the parameter. Obviously, the higher the parameter a, the more distorted the mesh is. It is clear that for $a = 1$, we have a regular mesh. Figure 4.6 presents two meshes of obtained with the progression with GMSH. On the left we used $a = 1.1$ and on the right we used $a = 1.4$.

Using the same tolerance as in the previous part, we ran the preconditioned conjugate gradients with the fine preconditioner for those new meshes. The order of the interpolation used is $p = 2$. The results are given in figure 4.7.

We can see, as it was expected, that the more we deform the mesh, the more iterations we need to do in order to obtain the wanted accuracy. The fact that, for distorted elements, we do not invert exactly the system but use an approximation allows us to compute the fine preconditioner efficiently even when we have a lot of elements but we can see here that it also costs more iterations of the preconditioned conjugate gradients. However, the gain is still huge.

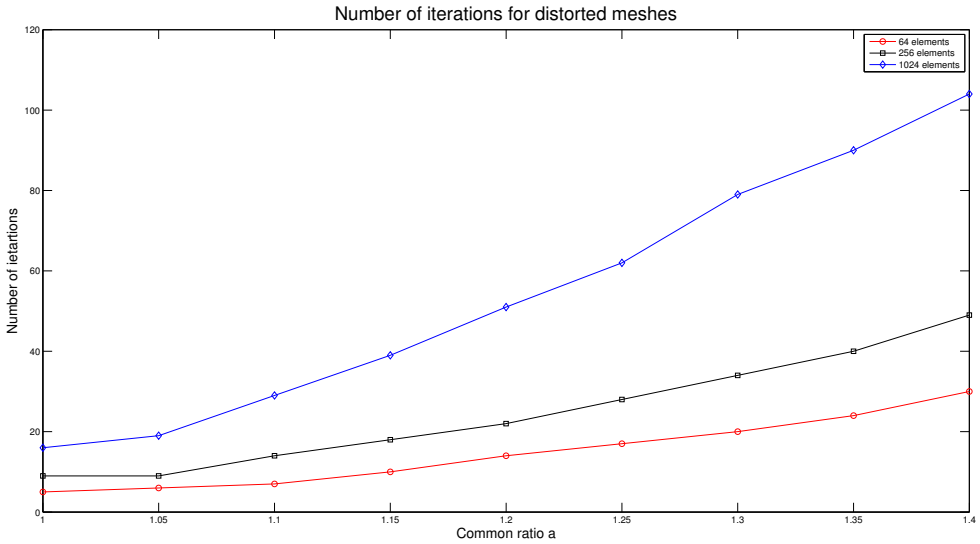


Figure 4.7: bla

Indeed, for a degree of interpolation p, we have $(p+3)^2$ nodes per overlapping subdomain. This means a complexity of $\mathcal{O}((p+3)^6)$ to solve the system exactly. Instead, with the method we use, we only need to do a few matrix multiplications where the matrices have $(p+3)$ rows and columns. This means a complexity of $\mathcal{O}((p+3)^3)$. Even with $p=2$, the results show that it is much faster to not take geometric factors into account.

We can also note that the effect of increasing the number of elements (i.e. reducing the mesh size) on the number of iterations is clearer here. For example, for the mesh with $a = 1.15$, we need 10 iterations for 8^2 elements, 18 iterations for 16^2 elements and 39 iterations for 32^2 elements. The same explanation applies here : when we multiply the number of elements by four, we roughly multiply the number of quadrants in each direction by two and therefore the information needs twice as many iterations to propagate to the domain.

4.2.2 Influence of hanging nodes

In this part, we will explore the influence of hanging nodes on the number of iterations needed by the preconditioned conjugate gradients with the fine preconditioner to converge. Because we want meshes that are not artificial and come from real AMR applications, we will change the forcing term in this part and we will also use a recursive refine function when we build the mesh with p4est.

Let us first define the problem and the forcing term. As before, the domain is : $\Omega = [-1;1]^2$ and Γ is the boundary. We will solve :

$$\nabla^2 u = -2\tanh(nx)\tanh(my)\left[n^2(1-\tanh(nx)^2) + m^2(1-\tanh(my)^2)\right] \quad \text{on } \Omega \qquad (4.5)$$

$$u = \tanh(nx)\tanh(my) \quad \text{on } \Gamma \qquad (4.6)$$

We can see that problem 4.5 has an analytic solution that is given by :

$$u(x,y) = \tanh(nx)\tanh(my)$$

The parameters n and m can be adjusted to make the jump in the hyperbolic tangent steeper. An example of a numerical solution with $p=1$ and obtained by our multigrid solver has already been shown on figure 4.1 for $n=3$ and $m=3$.

Tol	0.020	0.015	0.010
Number of quadrants	1276	1384	3064
hang	14.05%	16.61%	22.13%

Table 4.5: Number of quadrants obtained using the recursive refine function on the problem 4.5 for different tolerances as well as the ratio *hang* for each mesh with an interpolation of degree $p = 2$.

As explained before, we will use a recursive refine function when we build the forest. Since we know the analytic solution, we can cheat a little and use it for the refinement process. We will ask that the absolute value of the difference between the value of u in the center of the quadrant and the mean of the values of u at the four corners of the quadrant is less than a fixed tolerance multiplied by the maximum value of the function. So, if the four corners have coordinates (x_i, y_i) for $i = 0, 1, 2, 3$, and that $u_{max} = max_{x,y\in\Omega} |u|$, we impose the following rule :

$$\left| u(\frac{1}{4}\sum_{i=0}^{3} x_i, \frac{1}{4}\sum_{i=0}^{3} y_i) - \frac{1}{4}\sum_{i=0}^{3} u(x_i, y_i) \right| < u_{max} tol \tag{4.7}$$

Where *tol* is a fixed tolerance. Intuitively, the tighter the tolerance, the more refined the grid needs to be and the more hanging nodes we will have thanks to the jump in the hyperbolic tangent function.

Increasing the relative number of hanging nodes

To have a rather steep jump, we chose $n = m = 12$ for the different tests that follow. We varied the tolerance to have more or less hanging nodes. We also needed a way to quantify the presence of hanging nodes. Let us define *hang*, the ratio of the number of hanging nodes over the number of global nodes.

$$hang = \frac{\#\text{hanging nodes}}{\#\text{global nodes}} \tag{4.8}$$

Table 4.5 shows this number for different values of the tolerance and for $p = 2$. We can see, as expected, that the ratio *hang* increases while we make the tolerance tighter. Of course, the exact value of *hang* does not matter since even for a given mesh, it will change with the degree of the interpolation so it is rather how it evolves that interests us.

Let us now see how the number of iterations varies for the different meshes. Figure 4.8 shows the number of iterations needed to reach the same norm on the residual as before for the three different meshes obtained with the tolerances presented in table 4.5. We can see on the graph that the number of iterations increases when we have relatively more hanging nodes. We have to be careful since this phenomenon can also be explained by the fact that we have more quadrants when the tolerance gets tighter and that also causes an increase in the number of iterations as showed earlier in this section. However, it is observed that we need less iterations when we do not have hanging nodes and for a similar number of quadrants (for example, for 1024 quadrants without hanging nodes, we need 18 iterations whereas we need 38 for 1276 quadrants).

As explained in the implementation chapter, when we have hanging nodes, we do not treat all hanging possibilities and therefore we have an error when we compute the local residual. This explains why we need more iterations in presence of hanging nodes and the fact that we converge less quickly the more hanging nodes we have in the mesh. However, the number of iterations is still acceptable.

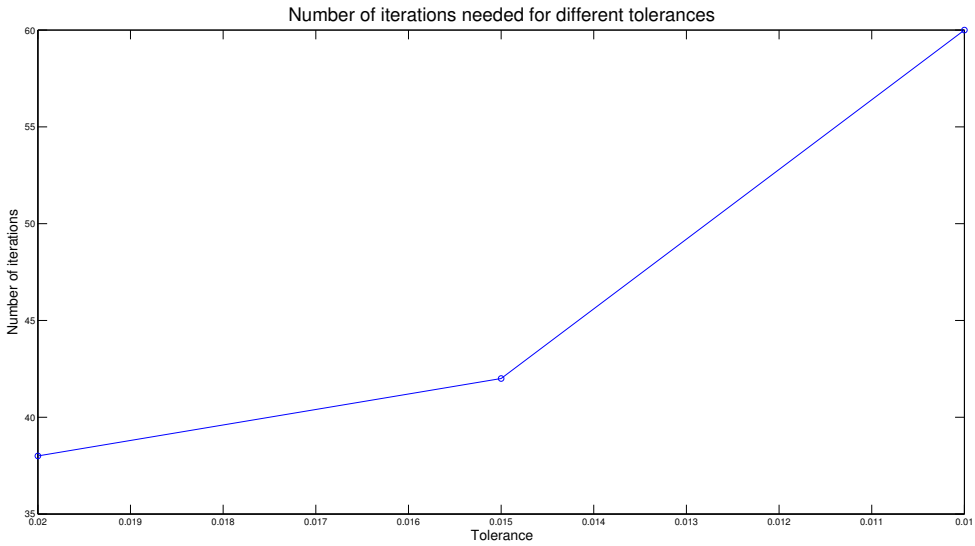


Figure 4.8: Number of iterations needed to reach a given norm on the residual for a degree of interpolation $p = 2$ and for meshes obtained with different tolerances.

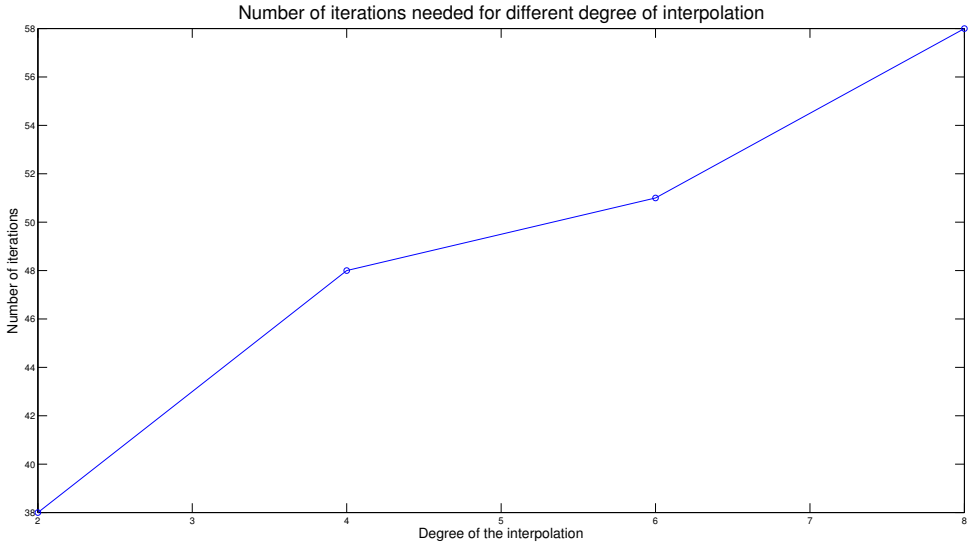


Figure 4.9: Number of iterations needed to reach the tolerance of the norm of the residual as a function of the degree of the interpolation used for a mesh obtained using $tol = 0.02$ in the recursive refine function.

Increasing the degree of the interpolation

Let us finally look at what happens when we increase the degree of the interpolation. We tried different degrees of interpolation ($p = 2, 4, 6, 8$) and looked at the number of iterations needed to obtain the numerical solution. Figure 4.9 shows the results.

We can see that the number of iterations increases with the degree. We need 38 iterations for $p = 2$ but 58 for $p = 8$. As already explained in the case with no hanging nodes, this is in part due to the fact that when we increase the degree of the interpolation, the size of the overlap decreases and therefore we need more iterations.

We can also mention the fact that since we do not treat all hanging possibilities when we compute the local residual at the overlaps, increasing the degree (and therefore the number of nodes in the overlaps) might have a effect on the number of iterations.

All the tests presented here were on meshes where the elements were not distorted and where we expect the fine preconditioner to behave optimally but the same tests have been performed with distorted quadrants and we observe the same qualitative results.

4.3 Two scale preconditioner

Let us finally move on to the two-scale preconditioner. We showed in the previous part that using only the fine scale preconditioner was not a good idea when the number of quadrants increased (the number of iterations roughly doubles when the mesh size is divided by two). As explained in the theory chapter, the idea is to add a coarse part in the preconditioner (i.e. $P = P^f + P^c$), consisting mainly of solving a problem with an interpolation degree $p = 1$ with a geometric multigrid method, so that the number of iterations stays constant when we increase the number of quadrants.

As in the previous section, the tests will be performed in two parts : first, we will analyze the performances of the two-scale preconditioner when there are no hanging nodes (but for both regular and distorted meshes) and then we will look at what happens when the forest of quadtrees is refined recursively and therefore the mesh is not conforming anymore.

In the last subsection, we will also motivate the choice of using higher order degrees of interpolation. We will indeed look at the best degree to obtain a given accuracy in the solution, i.e. the degree for which we have the solution to a given problem on a given mesh in the shortest amount of time and that is accurate enough.

4.3.1 No hanging nodes

Let us first present the problem we will solve in this part. We want our numerical solution to capture both low and high frequency modes so we will superpose a cosine with low frequency and a sine with a high frequency. As before, the domain is : $\Omega = [-1; 1]^2$ and Γ is the boundary. We will solve :

$$\nabla^2 u = -\frac{\pi^2}{2}\cos(\frac{\pi}{2}x)\cos(\frac{\pi}{2}y) - 5\pi^2 \sin(5\pi x)\sin(5\pi y) \quad \text{on } \Omega \tag{4.9}$$

$$u = 0 \quad \text{on } \Gamma \tag{4.10}$$

This problem has an analytic solution that is given by :

$$u(x, y) = \cos(\frac{\pi}{2}x)\cos(\frac{\pi}{2}y) + \frac{1}{10}\sin(5\pi x)\sin(5\pi y)$$

An example of a numerical solution obtained using the preconditioned conjugate gradients with the two scale preconditioner can be seen on figure 4.10. This solution has been obtained in only 8 iterations and with an interpolation of degree $p = 2$.

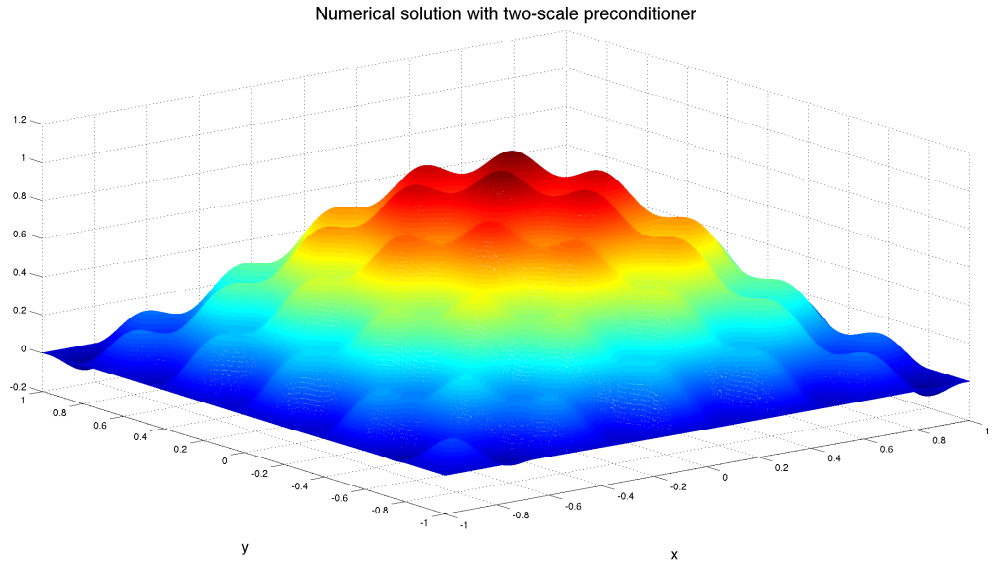


Figure 4.10: Numerical solution to problem 4.9 using an interpolation of order $p = 2$ and $1.0\,10^6$ degrees of freedom on a regular mesh with no hanging nodes. This numerical solution has been obtained by the PCG with the two scale preconditioner.

Number of quadrants	128^2	256^2	512^2	1024^2
$p = 2$	$6.6\,10^4$	$2.6\,10^5$	$1.1\,10^6$	$4.2\,10^6$
$p = 4$	$2.6\,10^5$	$1.1\,10^6$	$4.2\,10^6$	$1.7\,10^7$
$p = 6$	$5.9\,10^5$	$2.4\,10^6$	$9.4\,10^6$	$3.8\,10^7$
$p = 8$	$1.1\,10^6$	$4.2\,10^6$	$1.7\,10^7$	$6.7\,10^7$

Table 4.6: Number of degrees of freedom for a regular mesh with different numbers of quadrants and for different degrees of interpolation.

Regular meshes

Let us now look at what happens to the number of iterations when we decrease the mesh size for different degrees of interpolation. Table 4.6 shows the number of degrees of freedom for the different meshes used (indeed, for a given mesh, the higher the degree of the interpolation, the more global nodes we have). We can already see that, thanks to the coarse preconditioner, we are able to have a lot more degrees of freedom than in the case where we only had the fine preconditioner.

For the following tests, we also have tighten the tolerance on the norm of the residual. We now require that :

$$\frac{||r_k||_2}{||r_0||_2} < 10^{-5}$$

Figure 4.11 shows the number of iterations needed to reach that tolerance of the norm of the residual when we use our two scale preconditioner for different degrees of interpolation. The first remark we can make is that the number of iterations does not increase with the number of quadrants (as it was the case when we only had the fine preconditioner). So we can conclude that the coarse preconditioner does the job it was designed to do. We can even note that the number of iterations slightly decreases. For example, for $p = 6$, we need 12 iterations for 128^2 quadrants but we only do 11 iterations for 1024^2. An explanation for this phenomenon might be that when we increase the number of quadrants, we actually better separate the actions of the fine and coarse preconditioners and therefore the sum of the two is a better approximation of

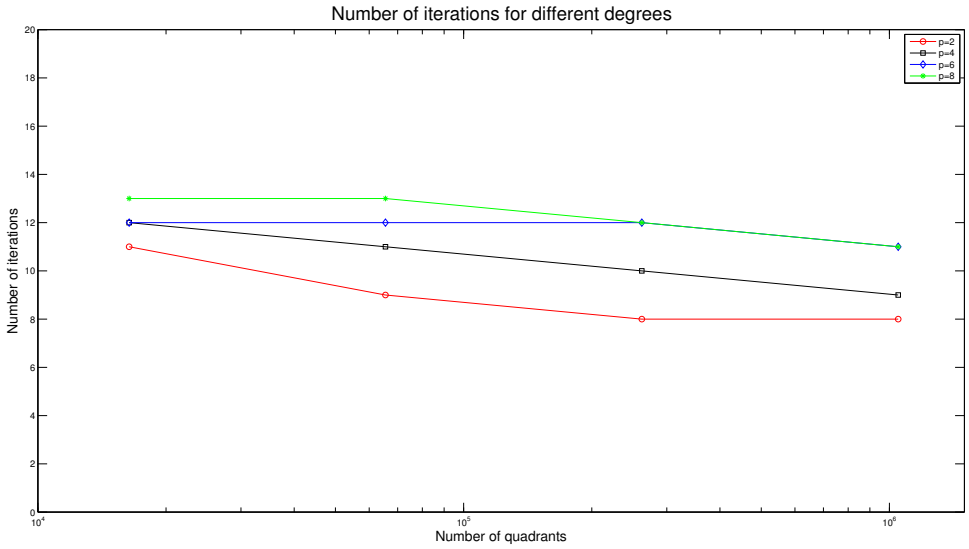


Figure 4.11: Number of iterations of PCG with the two scale preconditioner for degree p of interpolation needed to reach the given tolerance of the norm of the residual as a function of the number of quadrants in a regular mesh.

A^{-1}. We can mention that for $p = 8$, we only need 11 iterations to solve a system that has more than 67 millions unknowns.

A second remark we can make, as already observed when we only had the fine preconditioner, is that the number of iterations grows with the degree of the interpolation. For example, with 1024^2 quadrants, we need 8 iterations when $p = 2$ but we need to do 11 iterations when $p = 8$. As before, this can be explained by the fact that as the degree of the interpolation increases, the size of the overlap decreases. A fixed sized overlap would fix this issue.

Meshes with distorted elements

We will now look at what happens when the mesh is not regular but we have quadrants that are more and more distorted. We will use the same meshes already used in the previous section and obtained with the progression tool of GMSH (see figure 4.6 for example of such meshes).

For an interpolation of degree $p = 2$, we looked at the number of iterations needed to reach the given tolerance as we increased the number of quadrants and for elements more and more distorted. Figure 4.12 shows the results. We can see that the number of iterations stays roughly the same when we increase the number of quadrants so once again the coarse part of the preconditioner does the job it is designed to do.

We can also node that the more we distort the quadrants the more iterations we need to reach the given tolerance. This is to be expected since we developed the fine preconditioner to work optimally on quadrants aligned with the axis. This means that the more a quadrant is distorted the less accurate is the fine preconditioner part and therefore the more iterations we need. However, even with the most distorted mesh, the number of iterations stays acceptable and as explained in the previous section, the gain of not having to solve exactly on every quadrant is huge. We also have to say that the meshes presented here have an intrinsic structure since they are generated using the progression tool in GMSH. Therefore, the errors we make with our approximation are always in the same direction and the number of iterations needed increases. We might not have such an increase with a random mesh containing quadrants with the same quality measure.

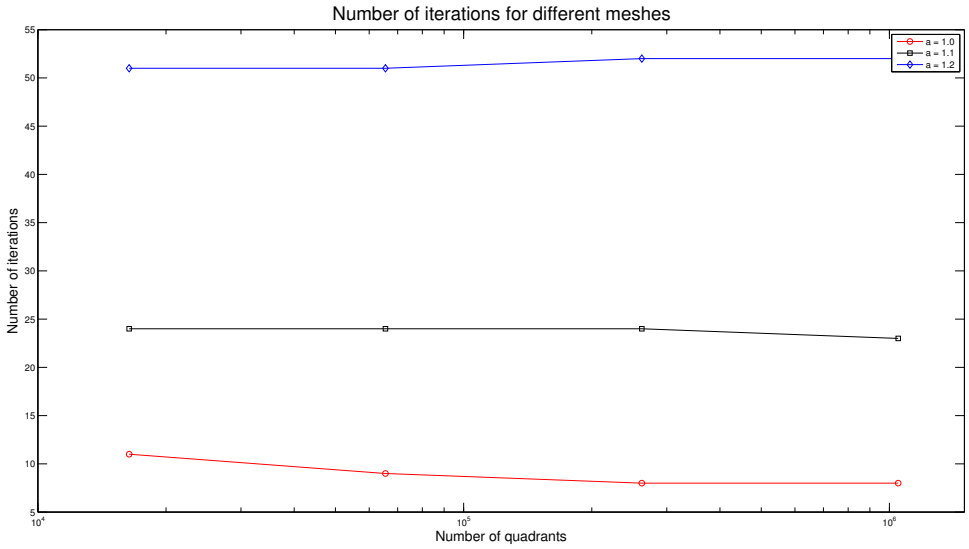


Figure 4.12: Number of iterations of PCG with the two scale preconditioner for an interpolation degree $p = 2$ needed to reach the given tolerance on the norm of the residual as a function of the number of quadrants for meshes with quadrants more and more distorted.

4.3.2 Influence of hanging nodes

Let us now move on to the cases where we have hanging nodes. As in the part with only the fine preconditioner, we will look at a problem where the solution has a jump so that we create hanging nodes when we use a recursive refine function in p4est. In addition to the hyperbolic tangent already presented in the previous subsection, we will include a high frequency sine wave to see how the grid adapts to capture it and how the two scale preconditioner performs in its presence.

The problem we will solve is :

$$\begin{aligned} \nabla^2 u = &-2\tanh(12x)\tanh(12y)\left[12^2(1-\tanh(12x)^2)+12^2(1-\tanh(12y)^2)\right] \\ &-10\pi^2\sin(10\pi x)\sin(10\pi y) && \text{on } \Omega \quad (4.11) \\ u = &\tanh(12x)\tanh(12y)+\frac{1}{20}\sin(10\pi x)\sin(10\pi y) && \text{on } \Gamma \quad (4.12) \end{aligned}$$

Where $\Omega = [-1;1]^2$ and Γ is the boundary. This problem has an analytic solution that is given by :

$$u(x,y) = \tanh(12x)\tanh(12y)+\frac{1}{20}\sin(10\pi x)\sin(10\pi y)$$

Figure 4.13 shows an example of the numerical solution computed on a non conforming mesh with PCG and the two scale preconditioner. We can see on the plot both the steep jump due to the hyperbolic tangent and the small oscillations due to the sine wave.

Increasing the relative number of hanging nodes

Let us now look at the influence of increasing the number of hanging nodes in the mesh. To achieve this, we will use the same rule in the recursive refinement of the quadrants as in the previous section, given by relation 4.7. We will also tighten the parameter *tol* used in order to have more quadrants than in the case when we only had the fine preconditioner.

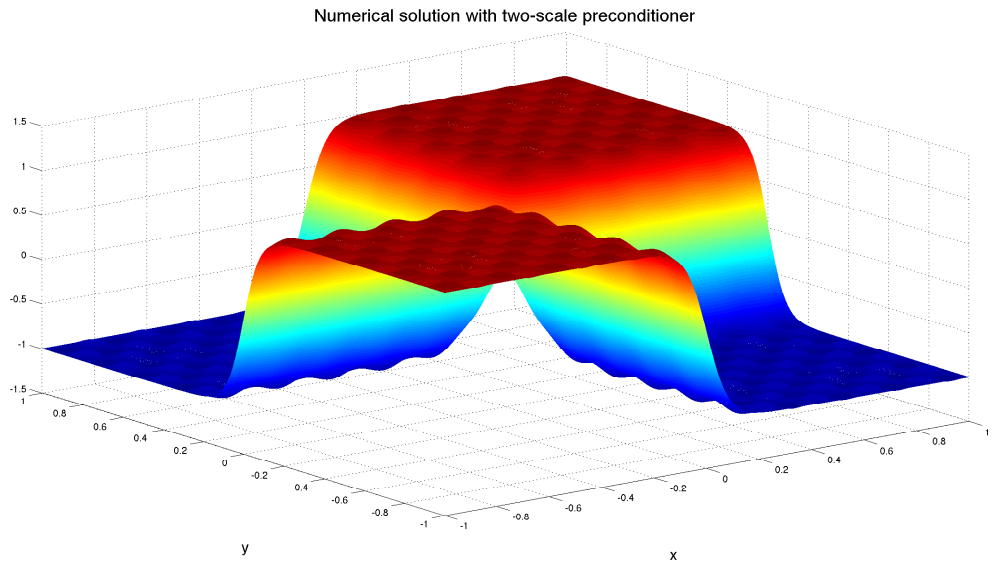


Figure 4.13: Numerical solution to problem 4.11 using an interpolation of order $p = 2$ on a non conforming mesh and obtained with PCG with the two scale preconditioner.

Parameter *tol*	0.01	0.008	0.006	0.004	0.002	0.001	0.0005
Number of hanging nodes	6080	8064	11008	14272	23200	33696	70112
Number of global nodes	19921	25545	32265	49265	120201	208137	445425
hang	30.52%	31.57%	34.12%	28.97%	19.30%	16.19%	15.74%

Table 4.7: Statistics about the different meshes used for the tests regarding the influence of hanging nodes on the number of iterations as well as the ratio *hang* defined by equation 4.8. The number of global and hanging nodes are for an interpolation degree $p = 2$.

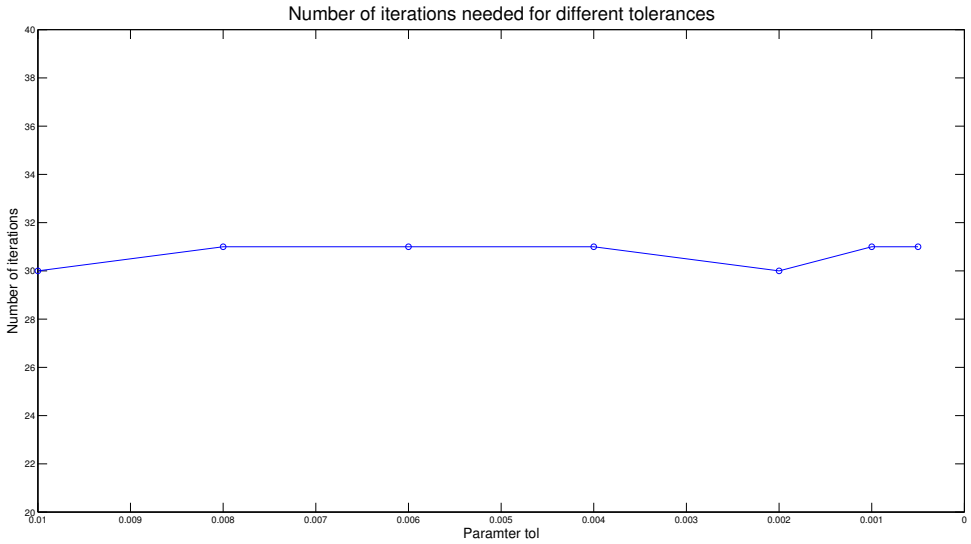


Figure 4.14: Number of iterations of PCG with the two scale preconditioner with a degree of interpolation $p = 2$ on meshes with hanging nodes defined by the parameter *tol*.

Table 4.7 shows some statistics about the different meshes used for the tests regarding the influence of hanging nodes on the number of iterations. The number of global and hanging nodes are for an interpolation of degree $p = 2$ and the ratio *hang* is given in equation 4.8.

We can see that, as expected, when we decrease the parameter *tol*, we have more hanging nodes. We can also see that, at first, tightening the parameter *tol* increase the ratio *hang* and we have more hanging nodes relatively to global nodes. Then, however, even tough the number of hanging nodes still grows, the number of global nodes grows faster and therefore the ratio *hang* decreases. This can be explained by the fact that if the parameter *tol* is very low then we start to refine all part more equally (even the ones where the function does not vary a lot) and the relative number of hanging nodes drops.

Figure 4.14 shows the number of iterations needed to reach the given tolerance with an interpolation degree $p = 2$ on the different meshes presented in table 4.7. We can see that the number of iterations stays roughly the same (it is either 30 or 31) for all meshes, whatever the number of hanging nodes and the value of the ratio *hang*. So we can conclude that the coarse preconditioner once again guaranties the h-independent convergence and that the number of hanging nodes does not have a great influence on the number of iterations.

We also have to note that, for the same problem, if we use a conforming mesh, the number of iterations needed drops to 16. So even if the number of hanging nodes (absolute or relative) does not greatly influence the number of iterations, just adding one non conforming quadrant does a lot to decrease the performance of the preconditioner. We can point the fact that, as mentioned earlier, when we have hanging quadrants, we do not handle every possibilities to compute the residuals in the overlaps. We can thus see how that affects the number of iterations.

Increasing the degree of the interpolation

Let us now move on to look at what happens when we increase the degree of the interpolation. We will use the mesh defined by $tol = 0.001$ and look at the number of iterations needed to reach the given tolerance for $p = 2, 4, 6, 8$. Figure 4.15 shows the results.

We can see that as we increase the degree of the interpolation, the number of iterations increases. It goes from 31 for $p = 2$ to 85 iterations when $p = 8$. We can see that it is quite a lot. But we also have to note that going from degree 2 to degree 8, we have also gone from 2.1 10^5

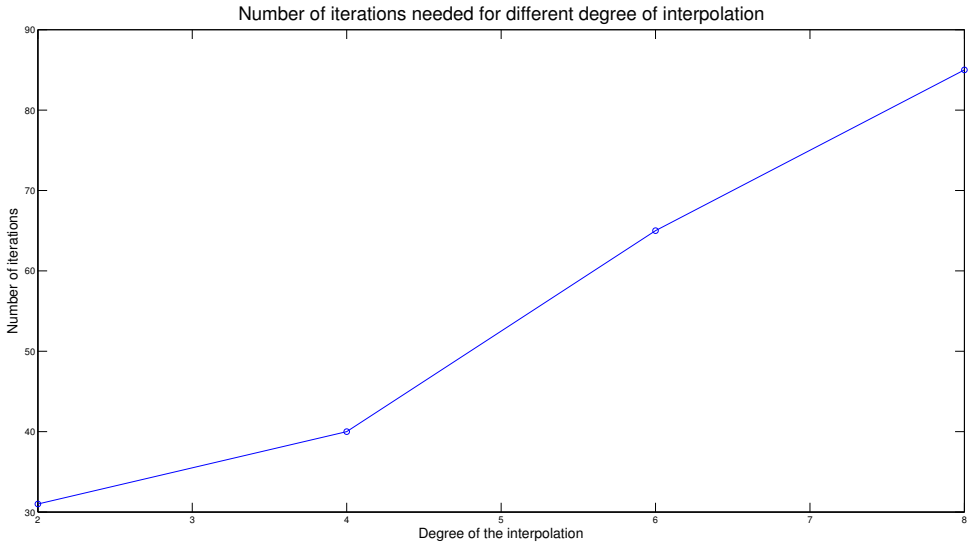


Figure 4.15: Number of iterations of PCG with the two scale preconditioner as a function of the degree of the interpolation used for a mesh obtained with $tol = 0.001$ in the recursive refine function.

degrees of freedom to $3.4\,10^6$.

As in the case when we only had the fine preconditioner, several factors can be put forward to explain the increase in the number of iterations. First, when we increase the degree, the size of the overlap decreases and therefore the fine preconditioner is less effective. As mentioned in [32], a fixed sized overlap would fix the issue.

Second, we can say that the number of hanging nodes in the overlaps increases with the degree of the interpolation. Since we do not treat all possibilities when handling hanging quadrants, the error made is more important when we have a lot of nodes in the overlaps.

4.3.3 Most efficient degree to obtain a given accuracy

In this subsection, we will look at the best degree of interpolation p to obtain a given accuracy. We will use the problem defined by 4.9 and the distorted mesh using the progression tool of GMSH with $a = 1.2$ that we will uniformly refine. We will also tighten the tolerance on the norm of the residual since we want the error to come from the interpolation and not from the fact that we only solve the linear system approximately. We will impose that :

$$\frac{||r_k||_2}{||r_0||_2} < 10^{-12}$$

Afterwards, we will compute the error committed by the discretization in the L^2-norm. Let us define e^p, the error made by using an interpolation of degree p, u^p the solution of the discretized problem using a degree p, U_i^p the numerical solution at the global node i of the linear system that arises and m_i the mass matrix associated with the integration. Then we have :

$$e^p = \left(\int_\Omega (u - u^p)^2 dxdy \right)^{\frac{1}{2}} \tag{4.13}$$

$$\approx \left(\sum_i (u(x_i, y_i) - U_i^p)^2 m_i \right)^{\frac{1}{2}} \tag{4.14}$$

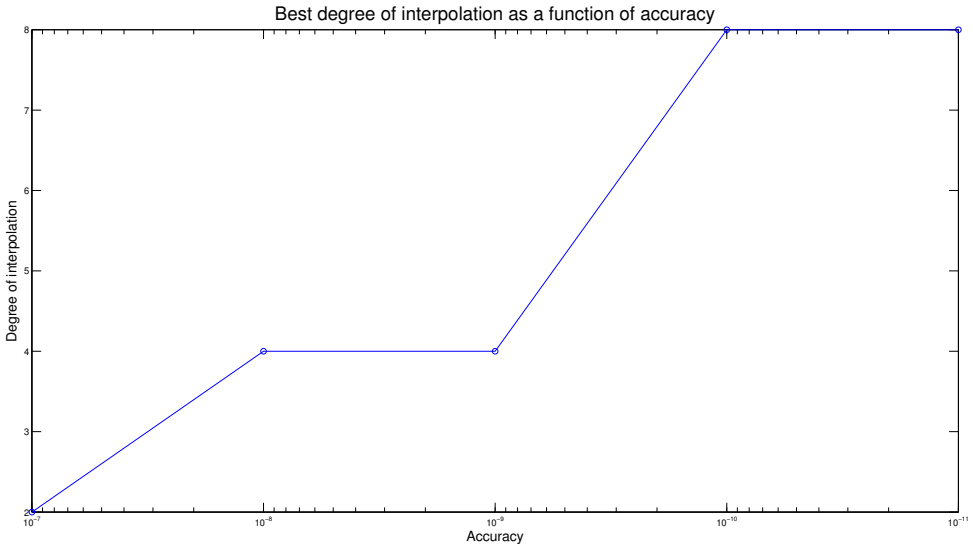


Figure 4.16: Best degree of interpolation to solve problem 4.9 on a distorted mesh as a function of the accuracy we want on the L^2-norm.

We can note that if we refine uniformly the mesh, then e^p will decrease but it will also cost us more. And for a given mesh, we have that e^p decreases as p increases (but it also costs us to increase the degree). So the question that arises is when is it more interesting to refine the mesh uniformly and when is it better to increase the degree of the interpolation to get the accuracy we want.

We will select degree p to be the best for an accuracy of 10^{-a} if and only if there is a mesh such that $e^p < 10^{-a}$ on this mesh and the time needed to obtain the numerical solution is smaller than for any degree on any other mesh and the same accuracy.

Figure 4.16 shows the results. We can see that when the wanted accuracy is rather low (for 10^{-7}) then it is quicker to use an order of interpolation $p = 2$ because the number of iterations needed to solve the linear system is smaller than for a higher order interpolation. Moreover, the number of degrees of freedom to obtain such an accuracy is not so high as to make one iteration very long.

On the other hand, when we want a better accuracy (10^{-10}, 10^{-11}) then it is cheaper to use higher order interpolation such as $p = 8$. This can be explained by the fact that, even though the number of iterations of PCG is higher (as mentioned earlier in this section), the number of degrees of freedom to obtain the wanted accuracy is much lower for $p = 8$ than for the other degrees. Therefore, one iteration takes much less time and $p = 8$ becomes more efficient.

In between the two (10^{-8}, 10^{-9}), we can see that the best degree is $p = 4$ where we have a trade-off between the number of iterations needed to solve the system and the number of degrees of freedom.

The quantitative results presented here depend of course of the implementation, the problem to solve and the mesh used. However, we can note that the qualitative result stays true : the more accurate we want to be, the more interesting it becomes to use higher degree in the interpolation because then the number of DOF does not grow as large as it would with lower order polynomials.

4.4 Conclusion

Let us now summarize the results given in this chapter.

We first looked at the properties of our geometric multigrid solver. The key property of h-independent convergence has been observed for a large variety of parameters ν_1 and ν_2 and

various meshes that were regular or distorted. The presence of hanging nodes has no influence whatsoever on the convergence of the geometric multigrid method.

We then moved on to the PCG with only the fine scale preconditioner. Several observations were made. We saw that, even on a regular mesh where the preconditioner is optimal, increasing the number of quadrants increased the number of iterations needed. That was predicted since increasing the number of quadrants decreases the mesh size. We also noticed that we needed more iterations when the degree of interpolation p grew. That was explained by the fact that the size of the overlap diminishes when p increases. We then moved on to meshes with distorted quadrants and saw that we needed more iterations than in the regular case. It was to be expected since the fine preconditioner was designed to work best on quadrants aligned with the axes. Afterwards, we introduced non conforming meshes. The presence of hanging nodes also increased the number of iterations. That was explained by the fact that we do not restrict the residual exactly in the overlaps. When we increase the degree of the interpolation on a non conforming mesh, the number of iterations increases as in the conforming case.

The coarse grid correction was then added in the preconditioner. We first tested the PCG with the two scale preconditioner on conforming meshes and saw that we obtained the h-independence thanks to the coarse preconditioner : the number of iterations stayed constant whatever the number of quadrants and the mesh size. That was true for both meshes with regular quadrants and meshes with distorted quadrants. We can however note that the number of iterations needed was greater in the case of distorted quadrants. Again, this was to be expected since the fine preconditioner works better on quadrants aligned with the axes. We then experimented on non conforming meshes. We saw that changing the absolute or the relative number of hanging nodes in the mesh had no real impact of the number of iterations. That being said, we saw that we needed more iterations when we had hanging nodes (even a few) than when we had none. We also looked at what happened when we increased the degree of the interpolation. As before, it increases with the degree p. The same remarks as before also apply here.

Finally, we picked a problem and a mesh and looked at the best degree of interpolation to solve this problem, i.e. the degree which allowed us to get the solution within a given tolerance in the L^2-norm the quickest. We saw that at first, for a low tolerance, a low degree is best since then the fine preconditioner is cheap to compute and we can have a lot of degrees of freedom. The number if iterations of PCG is also lower for lower degrees. Then, as we tighten the tolerance, it becomes more and more attractive to use higher degrees since they allow us to keep the number of degrees of freedom down and still get a good enough approximation to be within the tolerance. The increase in the number of iteration for higher degrees is compensated by the fact that we have a lot less degrees of freedom. This means that for a given tolerance, we have a trade-off between the number of iterations of PCG and the number of degrees of freedom.

Conclusion

This master thesis covered the study, the implementation and the comparison between the experimental and theoretical results of a *Poisson solver*, i.e. an algorithm designed to solve Poisson's equation. This equation pervades across multiple domains in engineering : astrophysics, chemistry, mechanics, statistics and image processing are examples of areas where elliptic partial differential equations appear. Several numerical schemes also involve the resolution of Poisson's equation, such as different methods to solve Navier-Stokes equations for incompressible flows. That is why having a fast and robust *Poisson solver* that can scale to trillions of unknowns and millions of CPU cores is a necessity.

There exists a lot of algorithms that can be implemented into a *Poisson solver*. We explained in the first chapter that, when the forcing term is a smooth function with very localized features (as it is often the case in practice), the high-order geometric multigrid method shows the best performances. In the present work, we chose to use low-order geometric multigrid method as a preconditioner along with an overlapping additive Schwarz preconditioner for the preconditioned conjugate gradients.

The chosen algorithm for the discretization was the spectral element method on an adaptive mesh. This is a high-order method that allows us to meet high accuracy requirements. The fact that we are solving the problem on an adaptive mesh allowed us to capture the very local features of the forcing term while keeping the number of unknowns down since, in practice, large parts of the computational domain do not require high levels of refinement. The second chapter exposed the theoretical reasons behind the choices made and how to handle hanging nodes, whose presence comes from the adaptive refinement.

The implementation of the algorithm resulted in a code written in plain C consisting of more than 8000 lines. It was not possible to present the entire structure of the program but the third chapter presents some important design features when implementing the algorithm. Especially, it was showed that the essential part of the design lies in the data structures used. The practical way to handle hanging nodes is also explained. In particular, the fact that the hanging nodes can only be found on the edges of the quadrants allows us to compute efficiently the relation between hanging and global nodes.

In the fourth chapter, we put the chosen algorithm to the test. Several observations were made and can be summarized as follows :

- The geometric multigrid method is very efficient and allows for h-independent convergence. For a large set of parameters, the number of iterations needed to reach the given tolerance did not depend on the number of quadrants in the mesh. The presence of hanging does not affect whatsoever the number of iterations. The method works both on meshes with regular quadrants and with distorted elements.

- Used on its own, the overlapping additive Schwarz preconditioner is not very efficient when the number of quadrant increases. Indeed, the number of iterations almost doubles when we divide the mesh size by two. This was expected in the absence of a coarse grid correction. On the other hand, when we increase the degree of the interpolation, the coarse preconditioner does the intended job and keeps the number of iterations down. We however still observe a small increase that is due to the fact that the size of the overlap decreases.

- For the fine scale preconditioner, we computed an analytic solution based on the assumption that the quadrants were aligned with the axes. Having distorted elements in the mesh has an influence on the number of iterations but the gain of not solving exactly the problem on each subdomain is still huge compared to the time lost by the growth of the number of iterations of PCG.

- The presence of hanging nodes has an influence of the fine preconditioner. It is less effective in non conforming meshes. That is because some cases are not treated when we compute the residual in the overlaps.

- Adding the coarse preconditioner to the fine one has a huge benefit. It allows to keep the number of iterations of PCG constant when we refine the grid. With the two scale preconditioner, we also have that the number of hanging nodes does not have an influence at all on the number of iterations. However, it has to be noted that we need fewer iterations when we do not have hanging nodes than when we do (even a few).

- If the accuracy requirements are low, it is quicker to use low order methods. This is because when we increase the degree of the interpolation, we also increase the number of iterations of PCG. As we increase the accuracy requirements, it becomes more and more important to use high-order methods. Indeed, for smooth functions, we need several order of magnitudes fewer degrees of freedom to reach a high accuracy with high-order method than with low-order ones. Therefore, the increase in the number of iterations of PCG is balanced by the fact that one iteration takes a lot less time to be performed.

This thesis essentially presents a fast *Poisson solver*. However, it could still be improved. First, we would like to be able to handle three dimensional problems. The second direct improvement would be to make the code parallel. Of course, the results presented here would not change (number of iterations, efficiency of the coarse and fine preconditioners,...) but it would be interesting to see if the algorithm developed in this thesis is also efficient when implemented on multiple cores. The parallelization of the matrix-vector products would be fairly straighforward, as would be the fine preconditioner since it is local-based. However, it would be more difficult for the geometric multigrid solver. Indeed, when we coarsen four quadrants into their parent to go up one level, we have no guarantee that the four children belong to the same processor. It is nonetheless feasible (for example in [35]). Another improvement we can mention is to try and keep the overlap constant even when we increase the degree of the interpolation in order to improve the fine preconditioner.

Bibliography

[1] Steven E. Benzley, Ernest Perry, Karl Merkley, Brett Clark, and Greg Sjaardema. A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elasto-plastic analysis. In *In Proceedings, 4th International Meshing Roundtable*, pages 179–191, 1995.

[2] Richard Pasquetti and Francesca Rapetti. Spectral element methods on triangles and quadrilaterals: comparisons and applications. *Journal of Computational Physics*, 198(1):349–362, 2004.

[3] David F Martin and Kelley L Cartwright. *Solving Poisson's equation using adaptive mesh refinement*. Citeseer, 1996.

[4] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[5] Michel O Deville, Paul F Fischer, and Ernest H Mund. *High-order methods for incompressible fluid flow*, volume 9. Cambridge university press, 2002.

[6] Philipp Schlatter. Spectral methods.

[7] FN Van de Vosse and PD Minev. Spectral elements methods: Theory and applications. Technical report, EUT Report 96-W-001 ISBN 90-236-0318-5, Eindhoven University of Technology, 1996.

[8] Jeff Thorson. Gaussian elimination on a banded matrix, 1982.

[9] Leslie Greengard and June-Yub Lee. Accelerating the nonuniform fast fourier transform. *SIAM review*, 46(3):443–454, 2004.

[10] Markus Geveler, Dirk Ribbrock, Dominik Göddeke, Peter Zajac, and Stefan Turek. Efficient finite element geometric multigrid solvers for unstructured grids on gpus. In *Second Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering*, page 22, 2011.

[11] Raymond S Tuminaro, Erik G Boman, Jonathan Joseph Hu, Andrey Prokopenko, Christopher Siefert, Paul H Tsuji, Jeremie Gaidamour, Luke Olson, Jacob Schroder, Badri Hiriyur, et al. Toward flexible scalable algebraic multigrid solvers. Technical report, Sandia National Laboratories (SNL-CA), Livermore, CA (United States); Sandia National Laboratories, Albuquerque, NM, 2013.

[12] Klaus Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1):281–309, 2001.

[13] Frank Ethridge and Leslie Greengard. A new fast-multipole accelerated poisson solver in two dimensions. *SIAM Journal on Scientific Computing*, 23(3):741–760, 2001.

[14] Harper Langston, Leslie Greengard, and Denis Zorin. A free-space adaptive fmm-based pde solver in three dimensions. *Communications in Applied Mathematics and Computational Science*, 6(1):79–122, 2011.

[15] Amir Gholami, Dhairya Malhotra, Hari Sundar, and George Biros. Fft, fmm, or multigrid? a comparative study of state-of-the-art poisson solvers for uniform and nonuniform grids in the unit cube. *SIAM Journal on Scientific Computing*, 38(3):C280–C306, 2016.

[16] Rüdiger Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. *Journal of Computational and Applied Mathematics*, 50(1-3):67–83, 1994.

[17] Seymour V. Parter. On the legendre–gauss–lobatto points and weights. *Journal of Scientific Computing*, 14(4):347–355, Dec 1999.

[18] Thomas-Peter Fries, Andreas Byfut, Alaskar Alizada, Kwok Wah Cheng, and Andreas Schröder. Hanging nodes and xfem. *International Journal for Numerical Methods in Engineering*, 86(4-5):404–430, 2011.

[19] Jan S Hesthaven and Einar M Ronquist. *Spectral and High Order Methods for Partial Differential Equations: Selected Papers from the ICOSAHOM'09 Conference, June 22-26, Trondheim, Norway*, volume 76. Springer Science & Business Media, 2010.

[20] Paolo Di Stolfo, Andreas Schröder, Nils Zander, and Stefan Kollmannsberger. An easy treatment of hanging nodes in hp-finite elements. *Finite Elements in Analysis and Design*, 121:101–117, 2016.

[21] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[22] J-F Remacle, R Gandham, and Tim Warburton. Gpu accelerated spectral finite elements on all-hex meshes. *Journal of Computational Physics*, 324:246–257, 2016.

[23] Koen Hillewaert, Nicolas Chevaugeon, Philippe Geuzaine, and Jean-François Remacle. Hierarchic multigrid iteration strategy for the discontinuous galerkin solution of the steady euler equations. *International journal for numerical methods in fluids*, 51(9-10):1157–1176, 2006.

[24] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[25] Alexandre Ern and Jean-Luc Guermond. Evaluation of the condition number in linear systems arising in finite element approximations. *ESAIM: Mathematical Modelling and Numerical Analysis*, 40(1):29–48, 2006.

[26] Wolfgang Hackbusch. *Multi-grid methods and applications*, volume 4. Springer Science & Business Media, 2013.

[27] Bärbel Janssen and Guido Kanschat. Adaptive multilevel methods with local smoothing for h^1-and h^curl-conforming high order finite element methods. *SIAM Journal on Scientific Computing*, 33(4):2095–2114, 2011.

[28] Tarek Mathew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations (Lecture Notes in Computational Science and Engineering)*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[29] X Cai. Overlapping domain decomposition methods. *Advanced Topics in Computational Partial Differential Equations*, pages 57–95, 2003.

[30] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[31] LAPACK - Linear Algebra PACKage. `http://www.netlib.org/lapack/`.

[32] Luca F Pavarino. Additive Schwarz methods for the p-version finite element method. *Numerische Mathematik*, 66(1):493–515, Dec 1993.

[33] GMSH. `http://gmsh.info/`.

[34] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.

[35] Rahul S Sampath and George Biros. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM Journal on Scientific Computing*, 32(3):1361–1392, 2010.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**