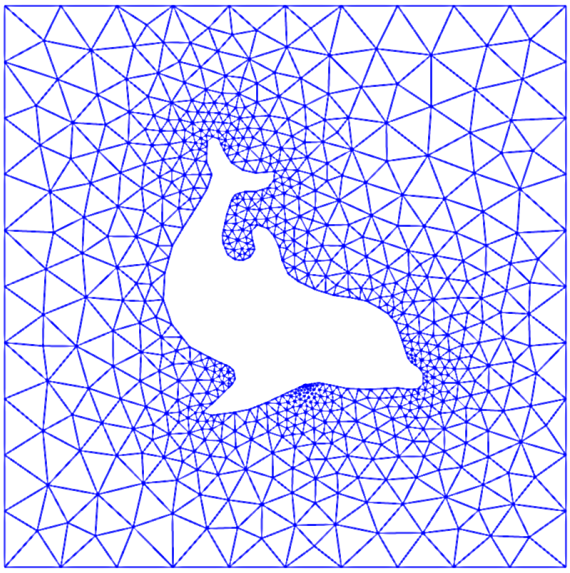
Bachelor Final Project Thesis

TU Delft, Faculty of Electrical Engineering, Mathematics and Computer Science

Mixed Finite Element Method for Elliptic Partial Differential Equations

Gemengde Eindige-Elementenmethode voor Elliptische Partiële Differentiaalvergelijkingen



Author:
Julian Johnston (4581075)

Supervisor:
Fred Vermolen

Additional Committee Members:
Neil Budko
Emiel van Elderen

This report has been written as part of the study *Applied Mathematics* at *Delft University* as to obtain a *Bachelor of Science* degree.

Delft, June 2019

TU Delft

Abstract

This report has the main aim of comparing the Mixed Finite Element Method to the standard Finite Element Method. The other aim is to let the reader understand what these methods entail. The latter is done by first journeying through the theory behind the FEM. It is first explored in one dimension to keep the setting simple. Next, the Mixed FEM is explored. A new way of approximating the gradient from the found solution is constructed, using the basis of Finite Differences and the ideas from the Finite Volume Method.

Following the theory is the implementation of the mentioned methods and the analysis of the results. It yields that, when looking at the L^2-norm, the classic FEM has a convergence order of 2, comparable to that of similar numerical methods. The Mixed FEM seemed to converge with an order of 4 in the same norm. Our constructed method only had a measly first order convergence, implying much greater accuracy of the Mixed FEM.

Contents

1 Introduction

As computing power and efficiency have increased massively over the past few decades, the branch of numerical analysis is larger than ever in the field of mathematics. Also other studies are relying more and more on the numerical sector, such as mechanical engineering and even life sciences, like biology. One of the numerical methods that has proved to be very successful in a great many applications is the Finite Element Method (FEM). It subdivides the region of interest into smaller regions that are easier to deal with, called elements, and then uses these to construct a system of equations. This way of computing means that the FEM is very suitable for unstructured grids, which is good for applied mathematics, as in most practical situations unstructured grids are required. Another reason the FEM is so widely used, is that it can easily be adapted to the problem at hand, and many variations on the method have been developed. One of these variations is the Mixed Finite Element Method (Mixed FEM). This method can calculate multiple physical quantities at once, usually implying greater accuracy of the results.

In the following chapter, the FEM and its theory are explored in one dimension. Then, in Chapter 3, the Mixed FEM is looked at. Chapter 4 then examines both methods in two dimensions. Next, the theory of so-called reference elements is examined. Finally, we analyse the numerical results of these methods and study the computational differences between them. In this part, we also create a new method for finding the gradient from the calculated solution.

This report has been written as part of the Bachelor Final Project of the study Applied Mathematics at the Delft University of Technology.

2 One-Dimensional Finite Element Method

In this chapter, we will introduce the ideas of the Finite Element Method (FEM) by describing them in one dimension. In this way, we feel one can get used to the new ideas in a comfortable scenario. Later on, when examining two dimensions, some of these ideas will be more skimmed over, as they will regularly be very similar to the ones presented here. The FEM is usually presented as a successor of the *Galerkin Method*, which is based on expressing the solution as a linear combination of *basis functions*. The idea of expressing the unknown function as such has proved to be very successful in different numerical methods. This chapter will however not rely on knowledge of this method; it will discuss the theory of the FEM from the beginning, while still maintaining an intuitive and logical progression through the ideas of the method. The ideas from this chapter follow mainly from the book *Numerical Methods in Scientific Computing* [1].

2.1 Definitions and the Weak Formulation

The best way to start something difficult is by looking at an easy example. Hence, to keep things simple at first, we look at the region $\Omega = (0, 1)$. The presented ideas can however easily be translated to different intervals or regions of $\mathbb{R}$. Consider the following differential equation:

$$\begin{cases} -u''(x) = f(x), & \text{on } \Omega, \\ u(0) = u(1) = 0. \end{cases} \tag{2.1}$$

This is the one-dimensional Poisson equation (with homogeneous boundary conditions), which arises frequently in differential equation analysis. We will first ignore the boundary conditions, but they will be discussed later in the chapter. In (2.1), f is a given function and u is the unknown function. We now consider the function space $\Sigma_0(\Omega) := \{u : u \text{ is sufficiently smooth, } u|_{\partial\Omega} = 0\}$, where $\partial\Omega$ represents the boundary of our region Ω. The statement that u be "sufficiently smooth" is rather vague, but it turns out that stating that it means that $u \in H^1(\Omega) = \{u \in L^2(\Omega) : u' \in L^2(\Omega)\}$ is sufficient for our purposes. Without changing the truth value of (2.1), we can multiply both sides by a function $\phi \in \Sigma_0(\Omega)$, usually called a *test function*:

$$-\phi u'' = \phi f.$$

Next, we integrate over our region Ω:

$$-\int_0^1 \phi u''\, dx = \int_0^1 \phi f\, dx.$$

We can use integration by parts to minimise the order of the derivative on the left hand side:

$$\begin{aligned} -\int_0^1 \phi u''\, dx &= -\left[\phi u'\right]_0^1 + \int_0^1 \phi' u'\, dx \\ &= \int_0^1 \phi' u'\, dx, \end{aligned}$$

by the boundary conditions, posed in $\Sigma_0(\Omega)$. We have done all this so that we can formulate the *Weak Formulation* of (2.1):

$$\textbf{Find } u \in \Sigma_0(\Omega), \textbf{ such that } \quad \int_0^1 \phi' u'\, dx = \int_0^1 \phi f\, dx, \quad \textbf{for all } \phi \in \Sigma_0(\Omega). \tag{2.2}$$

This Weak Formulation is the foundation on which the FEM is built; therefore it is important that it is correct. The Weak Formulation relies on the fact that the mentioned integrals exist. To see that this is true, we can use the *inequality of Cauchy-Schwarz*, if we interpret the integral of a product of functions as an inner product. This inequality says that

$$\left| \int_0^1 \phi' u' dx \right| \leq \left[\int_0^1 (\phi')^2 \right]^{\frac{1}{2}} \left[\int_0^1 (u')^2 \right]^{\frac{1}{2}}.$$

These right hand side integrals of course exist since $u, \phi \in \Sigma_0(\Omega)$, thus the left hand side integral must also exist.

2.2 Basis Functions

As in most numerical methods, the region in question needs to be discretised. We subdivide the region Ω into the intervals $[x_{k-1}, x_k]$, where the x_k are the *grid nodes*. Here, $x_0 = 0$ and $x_n = 1$. Furthermore, $x_0 < x_1 < \ldots < x_n$. In a structured grid scenario, we have that $x_k = k\Delta x$, where $\Delta x = \frac{1}{n}$. We denote these intervals by e_k, and we call them the *elements*. Note that the elements are finite in size and in number. This is why the Finite Element Method is named as such. These elements are shown in Figure 2.1. Note however that these elements need not all have the same length; unstructured grids are actually rather common.

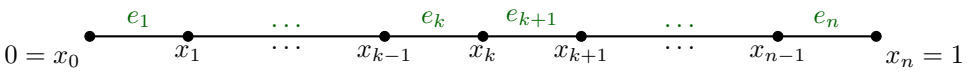


Figure 2.1: The discretisation of Ω.

From now on, u_k denotes $u(x_k)$. We can imagine u being approximated by simply connecting the u_k by straight lines. This approximation is then piecewise linear per element. Of course, this approximation is imaginary, since u is not (yet) known. However, we can replicate this approximation using the basis functions:

$$\tilde{u}(x) = \sum_{j=0}^{n} u_j \phi_j(x). \tag{2.3}$$

To satisfy our previous conditions, these ϕ_i are subject to two rules:

$$\begin{aligned} &1. \quad \phi_i \text{ is linear per element;} \\ &2. \quad \phi_i(x_j) = \delta_{ij}, \end{aligned} \tag{2.4}$$

where δ_{ij} is the *Kronecker delta*, which is 1 if $i = j$, and 0 otherwise. Typically, the basis functions look like the one drawn in Figure 2.2.

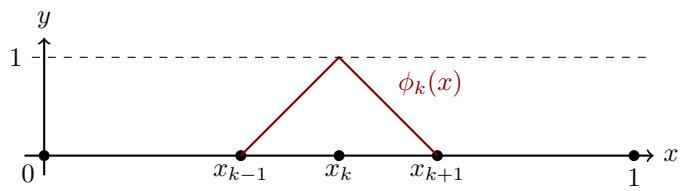


Figure 2.2: A typical basis function over e_k.

Note that $\phi_j(0) = \phi_j(1)$, for all $j \in \{1, \ldots, n-1\}$. One can also define quadratic, cubic and even higher degree basis functions. This is done by defining more nodes per element and then interpolating the *Lagrange polynomial* of the same degree such that rule 2 from (2.4) holds [2]

(See Appendix A). The following ideas will then still hold. However, we will present them only for linear basis functions.

We can now use (2.3) and Figure 2.2 to start the construction of the system of equations, which will ultimately be used to find an approximation of u.

2.3 Constructing the System of Equations

The basis functions are named as such because they form a basis of $\Sigma_0(\Omega)$. Therefore, we can demand that the arbitrary function ϕ from (2.2) be written as a linear combination of the basis functions:

$$\phi = \sum_{j=0}^{n} b_j \phi_j, \tag{2.5}$$

where the b_j are constants. Using the approximation of u from (2.3) and the expression of ϕ from (2.5) we can rewrite the equation in (2.2) as:

$$\int_0^1 \frac{d}{dx}\left(\sum_{j=0}^{n} b_j \phi_j\right) \frac{d}{dx}\left(\sum_{i=0}^{n} u_i \phi_i(x)\right) dx = \int_0^1 f \sum_{j=0}^{n} b_j \phi_j \; dx. \tag{2.6}$$

We can choose b_j to be 1 exactly once and 0 all other times, which implies the following system of equations:

$$\sum_{i=0}^{n} u_i \int_0^1 \frac{d\phi_i}{dx}\frac{d\phi_j}{dx} \; dx = \int_0^1 f\phi_j \; dx, \quad \text{for } j = 0, 1, ..., n. \tag{2.7}$$

Note that this is the same as saying that $\phi = \phi_j$, for each j. Therefore, we will hold this idea in our minds from now on.

Element matrices and element vectors

Of course, we want to write (2.7) in matrix-vector notation, i.e.

$$S\mathbf{u} = \mathbf{f},$$

since this is the notation that computers can handle. We can construct the matrix S and the vector $\mathbf{f}$ in an element-wise way, using the fact that

$$S_{ij} = \int_0^1 \frac{d\phi_i}{dx}\frac{d\phi_j}{dx} \; dx = \sum_{k=1}^{n} \int_{e_k} \frac{d\phi_i}{dx}\frac{d\phi_j}{dx} \; dx.$$

Since ϕ_i and its derivative are mostly zero, this is easier to compute than one might think; only four of these integrals are non-zero per element. These four integrals are stored in the element's *element matrix*:

$$S^{e_k} = \begin{bmatrix} \int_{e_k} \frac{d\phi_{k-1}}{dx}\frac{d\phi_{k-1}}{dx} \; dx & \int_{e_k} \frac{d\phi_{k-1}}{dx}\frac{d\phi_k}{dx} \; dx \\ \int_{e_k} \frac{d\phi_k}{dx}\frac{d\phi_{k-1}}{dx} \; dx & \int_{e_k} \frac{d\phi_k}{dx}\frac{d\phi_k}{dx} \; dx \end{bmatrix}.$$

When using the basis functions as drawn in Figure 2.2, these integrals are easy to compute. In that case, the derivative of the basis function is always either $1/\Delta x$ or $-1/\Delta x$, depending on the element and assuming an equidistant gridsize where $x_k - x_{k-1} = \Delta x$ for all k. The element matrices that arise in our example are

$$S^{e_k} = \frac{1}{\Delta x}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad \text{for all } k.$$

Furthermore, each element has its own *element vector*:

$$\mathbf{f}^{e_k} = \begin{bmatrix} \int_{e_k} f\phi_{k-1} \; dx \\ \int_{e_k} f\phi_k \; dx \end{bmatrix}.$$

Depending on the given function f, these integrals might not be analytically computable. One has to use a numerical integration method instead, such as the Trapezium Rule or Simpson's Rule. For example, using the Trapezium Rule and the basis functions from Figure 2.2, we get the following element vectors:

$$\mathbf{f}^{e_k} = \frac{1}{2}\Delta x \begin{bmatrix} f(x_{k-1}) \\ f(x_k) \end{bmatrix}, \quad \text{for all } k.$$

After calculating these for each element, we can calculate the whole matrix and vector by the following recurrences:

$$S^0 = \mathbf{0}_{n\times n}, \qquad S^k = S^{k-1} + S^{e_k},$$
$$\mathbf{f}^0 = \mathbf{0}, \qquad \mathbf{f}^k = \mathbf{f}^{k-1} + \mathbf{f}^{e_k}.$$

Note, however, that the k-th element matrix is added on the k-th row and column of S. The same applies to $\mathbf{f}$.

2.4 Dealing with the Boundary Conditions

Recall that we imposed boundary conditions on our problem, namely:

$$u(0) = u(1) = 0.$$

Since $x_0 = 0$ and $x_n = 1$, we can write this as

$$u_0 = u_n = 0.$$

This implies that the first and the last term of the sum in (2.3) are both zero, in turn implying that it run from 1 to $n-1$, rather than from 0 to n. This of course eliminates the first and last row and column of the matrix S, and the first and last element of $\mathbf{f}$.

These homogeneous boundary conditions are easy enough to deal with, but what about non-homogeneous boundary conditions? Consider the same problem, but with one inhomogeneous boundary condition:

$$u(0) = \alpha, \; u(1) = 0.$$

Naturally, the right hand side boundary condition is dealt with in the same way as before. However, the inhomogeneous one is somewhat differently approached. We may now write (2.3) as

$$\tilde{u}(x) = \sum_{j=1}^{n} u_j \phi_j(x) + \alpha\phi_0(x),$$

since clearly $u_0 = \alpha$. Now, this implies that (2.7) be written as:

$$\sum_{i=1}^{n} u_i \int_0^1 \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}\, dx = \int_0^1 f\phi_j \, dx - \alpha\int_0^1 \frac{d\phi_0}{dx}\frac{d\phi_j}{dx}\, dx, \quad \text{for } j = 1, 2, ..., n.$$

Note that $j = 0$ is now not part of the system. The new right hand side term implies that we can just incorporate the boundary condition in the right hand side vector. Note that this new term is only non-zero when $j = 1$, as discussed previously, therefore the first row and column of the matrix may still be removed. However, the first entry of the right hand side vector $\mathbf{f}$ is now

$$\Delta x f_1 + \frac{\alpha}{\Delta x},$$

where f_1 denotes $f(x_1)$. Neumann boundary conditions, such as $u'(0) = \beta$, do not pose as big a problem. In fact, homogeneous Neumann boundary conditions imply that (2.7) stay exactly the same! Inhomogeneous Neumann boundary conditions simply add a term to one entry of $\mathbf{f}$ again.

After S and $\mathbf{f}$ have been found, one can solve the equation with a method of their liking, such as CG or GMRES. These methods will not be discussed here since it is beyond the scope of this report and there are many good sources available that discuss these methods, such as [1] and [3].

3

One-Dimensional Mixed Finite Element Method

Some problems, such as flow problems, require that u as well as u' are known. One could approximate u' directly from u, after calculating it with the FEM. Another way, however, is to use the Mixed FEM, which — in a way — calculates both at the same time. This method uses many of the ideas from Chapter 2, but worked in a slightly different way.

3.1 The Setting

To keep things simple again, we consider the same problem as in Chapter 2, namely (2.1). We consider a second function, p, defined by

$$p = u'.$$

Substituting this into our problem (2.1) implies the following:

$$\begin{cases} -p' = f(x), \\ u' = p, \end{cases} \tag{3.1}$$

again on $\Omega = (0, 1)$. Again, of course, one could easily extend the ideas presented to other regions of $\mathbb{R}$. We now have a problem to solve for u and p. We can then multiply each equation by its own test function — denoted by ϕ and ψ, respectively — and integrate over Ω:

$$-\int_0^1 p'\phi \, dx = \int_0^1 \phi f \, dx; \tag{3.2a}$$

$$\int_0^1 u'\psi \, dx = \int_0^1 \psi p \, dx. \tag{3.2b}$$

We can use integration by parts on (3.2a) to get:

$$\begin{aligned} -\int_0^1 p'\phi \, dx &= -\left[p\phi\right]_0^1 + \int_0^1 p\phi' \, dx \\ &= \int_0^1 p\phi' \, dx. \end{aligned}$$

We could also choose to integrate (3.2b) by parts instead (see Appendix B). We can now conclude that $p, \psi \in L^2(\Omega)$ and $u, \phi \in H^1(\Omega)$, since p and ψ have no derivatives in the integrals, while u and ϕ do.

3.2 System of Systems of Equations

Assuming the same elements as in Chapter 2, we can write approximations of p and u and set ϕ and ψ to one of their respective basis functions, just as was done in Chapter 2:

$$\tilde{p} = \sum_{j=0}^{n} p_j \psi_j, \qquad \psi = \psi_j, \tag{3.3a}$$

$$\tilde{u} = \sum_{j=0}^{n} u_j \phi_j, \qquad \phi = \phi_j. \tag{3.3b}$$

Note that with our chosen boundary conditions, $u_0 = u_n = 0$. However, we still let the sum run from 0 to n, so that the matrices have the same dimensions in a later step, and also to keep things as general as possible. Neumann boundary conditions imposed on u would have a similar effect on the first sum, since they are really just Dirichlet boundary conditions on p. We can plug the new expressions (3.3a, 3.3b) into (3.2a) and (3.2b) (after partial integration) to get a system of systems of equations:

$$\begin{cases} \displaystyle\sum_{j=0}^{n} p_j \int_0^1 \psi_j \frac{d\phi_i}{dx}\, dx = \int_0^1 \phi_i f\, dx, & \text{for all } i, \\ \displaystyle\sum_{j=0}^{n} p_j \int_0^1 \psi_j \psi_i\, dx - \sum_{j=0}^{n} u_j \int_0^1 \frac{d\phi_j}{dx} \psi_i\, dx = 0, & \text{for all } i. \end{cases} \tag{3.4}$$

In this case, if we write this in the conventional matrix-vector notation, we get an equation with a block matrix:

$$\begin{bmatrix} A & \mathbf{0} \\ B & -A^\top \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix}, \tag{3.5}$$

where

$$A_{ij} = \int_0^1 \frac{d\phi_i}{dx} \psi_j\, dx, \qquad B_{ij} = \int_0^1 \psi_i \psi_j\, dx \quad \text{and} \quad \mathbf{f}_i = \int_0^1 \phi_i f\, dx.$$

Some of these integrals might need to be approximated with previously discussed methods. Now the solution vector consists of an approximation of u and $p = u'$ at the same time, which is exactly what we wanted. Any inhomogeneous boundary conditions will appear in the right hand side vector of (3.5).

4

Two-Dimensional FEM and Mixed FEM

We have seen the ideas of the FEM and Mixed FEM in the previous chapters. These ideas extend easily to two dimensions and more. The procedures are very similar to the one-dimensional case, but do have to be adjusted a bit.

4.1 FEM in Two Dimensions

Again, we want to keep things simple, but easily generalisable. We say that our region is $\Omega = [0,1] \times [0,1]$, the unit square. We consider the two-dimensional Poisson Equation:

$$\begin{cases} -\Delta u = f, & \text{on } \Omega, \\ u = 0, & \text{on } \partial\Omega, \end{cases} \tag{4.1}$$

where $\Delta := \nabla^2$ is the *Laplacian*. After multiplying by a test function and integrating over Ω, we get

$$-\int_\Omega \phi \Delta u \; d\Omega = \int_\Omega \phi f \; d\Omega,$$

which can be integrated by parts:

$$\int_\Omega \nabla\phi \cdot \nabla u \; d\Omega = \int_\Omega \phi f \; d\Omega. \tag{4.2}$$

These steps are of course very similar to the one-dimensional case and they naturally lead to the Weak Formulation, as discussed in Chapter 2. Now, we can again approximate u using basis functions:

$$\tilde{u} = \sum_{j=0}^{n} u_j \phi_j(\mathbf{x}). \tag{4.3}$$

We cannot define the two-dimensional basis functions until we define the two-dimensional elements, though. Hence, we will define these first. A common choice for two-dimensional elements is a triangle shape. For an example, see Figure 4.1:

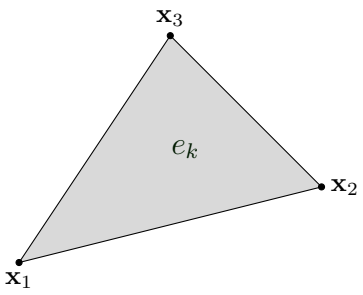


Figure 4.1: An example of an element in two-dimensional space

Note that these elements are defined solely by their vertices. The basis functions are implicitly defined by the rules from before in (2.4). Since they have to be linear, they have the following form:

$$\phi_i(\mathbf{x}) = \alpha_i + \beta_i x + \gamma_i y.$$

The coefficients α_i, β_i and γ_i depend on the element and can be calculated with a system of three equations. A typical basis function looks like in Figure 4.2.

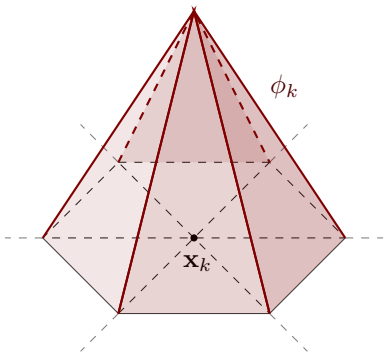


Figure 4.2: A typical two-dimensional basis function

In Figure 4.2, we note that the basis function satisfies $\phi_k(\mathbf{x}_k) = 1$, while it vanishes in all other grid nodes, just as we conditioned. Now that we know what the basis functions look like, we can substitute (4.3) into (4.2) and set $\phi = \phi_i$, to get the system of equations:

$$\sum_{j=0}^{n} u_j \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, d\Omega = \int_\Omega \phi_i f \, d\Omega, \qquad \text{for all } i. \tag{4.4}$$

Then the element matrices are:

$$S^{e_k} = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix},$$

with $S_{ij} = \int_{e_k} \nabla\phi_i \cdot \nabla\phi_j \, d\Omega$. This can be expressed in the following way:

$$S_{ij} = \frac{|\Delta|}{2}(\beta_i\beta_j + \gamma_i\gamma_j),$$

where

$$\Delta = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

Furthermore, the element vectors are:

$$\mathbf{f}^{e_k} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}, \qquad f_i = \int_{e_k} \phi_i f \, d\Omega.$$

Now, two-dimensional versions of the previously mentioned integration rules might need to be applied, depending again on the given function f.

4.2 Mixed FEM in Two Dimensions

Just like in one-dimensional space, we consider a second function p, now defined such that

$$p = \nabla u.$$

Substituting this into (4.1) gives:

$$\begin{cases} -\nabla \cdot p = f, \\ \nabla u = p. \end{cases} \tag{4.5}$$

Our standard procedure of multiplying by test functions, integrating over the domain, and integrating by parts gives us:

$$\begin{aligned} -\int_\Omega \psi \nabla \cdot p \; d\Omega &= \int_\Omega \phi f \; d\Omega; \\ -\int_\Omega u \nabla \cdot \psi \; d\Omega &= \int_\Omega \psi p \; d\Omega. \end{aligned} \tag{4.6}$$

The further procedures are very nearly identical to those in Chapter 3, and hence will not be discussed further. The procedures presented here can become quite hard to implement once complicated grid structures come into play. This is solved in the following chapter.

5 Mapping to a Reference Element

It has been said before in this report that the FEM is well suited for unstructured grids. One way it achieves this is by using a *reference element* when encountering complicated grid structures. We will first explain the details in one dimension, and then extend the ideas to two dimensions. These ideas follow mostly from personal interviews with the supervisor [4], but similar ideas are presented in [1] and [2].

5.1 Reference Element in One Dimension

We now consider the region $\Omega \subseteq \mathbb{R}$ and its discretisation to be arbitrary. Consider a simple arbitrary element $e_k = [x_{k-1}, x_k]$ of this discretisation. We can define a transformation function T that transforms this element to the easy to deal with interval $e' := [0, 1]$. e' is called the *reference element*. See Figure 5.1 for a visualisation of this idea.

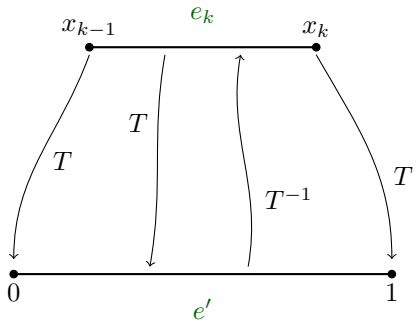


Figure 5.1: The transformation from e_k to e'

One can see in Figure 5.1 that we want T to satisfy $x_{k-1} \mapsto 0$ and $x_k \mapsto 1$. One can realise this by defining the parametrisation

$$x(s) := x_{k-1}(1 - s) + x_k s, \tag{5.1}$$

where $0 \leq s \leq 1$. This clearly satisfies $x(0) = x_{k-1}$ and $x(1) = x_k$. From (5.1) we can see that

$$x'(s) = x_k - x_{k-1}.$$

Note that this is just the length of the element e_k, and thus one could interpret this as a *scale factor*. Using this information we can for example integrate f as follows:

$$\int_{e_k} f(x)\, dx = \int_0^1 f(x(s))x'(s)\, ds = (x_k - x_{k-1}) \int_0^1 f(x(s))\, ds.$$

Notice the *Jacobian* $x'(s)$. The Jacobian is necessary because we are performing a transformation. Now, of course, we must also integrate the basis functions, as they are an integral part of the FEM. On the element e_k we have to deal with the two basis functions ϕ_{k-1} and ϕ_k. They are already simple, but when transforming them, they get even simpler:

$$\begin{cases} \phi_{k-1}(x(s)) = 1 - s, \\ \phi_k(x(s)) = s. \end{cases} \tag{5.2}$$

Intuitively, this is because, on the reference element, ϕ_{k-1} connects $(0,1)$ to $(1,0)$, and ϕ_k connects $(0,0)$ to $(1,1)$. Their respective derivatives — with respect to s — are then simply -1 and 1. One advantage of the reference element is that we fundamentally only have to define the basis functions for one element, namely the reference element.

As a check that we still get the same result as before in Chapter 2, we calculate the following integral:

$$\int_{e_k} \phi'_{k-1}\phi'_k \, dx,$$

as this integral, and similar ones, pop up a lot in the FEM. We first notice that the *chain rule* gives us that

$$\frac{d\phi_k}{dx} = \frac{d\phi_k}{ds}\frac{ds}{dx}, \qquad \frac{d\phi_{k-1}}{dx} = \frac{d\phi_{k-1}}{ds}\frac{ds}{dx}.$$

We already discussed the derivatives of the basis functions. The only term left to calculate is $\frac{ds}{dx}$, but it is also easily calculated:

$$\frac{ds}{dx} = \frac{1}{\frac{dx}{ds}} = \frac{1}{x_k - x_{k-1}}.$$

Now, without forgetting the Jacobian, we can calculate the integral from above:

$$\begin{aligned}
\int_{e_k} \phi'_{k-1}\phi'_k \, dx &= \int_0^1 \frac{d\phi_k}{ds}\frac{ds}{dx} \cdot \frac{d\phi_{k-1}}{ds}\frac{ds}{dx} \cdot \frac{dx}{ds} \, dx \\
&= \int_0^1 1 \cdot \frac{1}{x_k - x_{k-1}} \cdot -1 \cdot \frac{1}{x_k - x_{k-1}} \cdot (x_k - x_{k-1}) \, dx \\
&= \frac{-1}{x_k - x_{k-1}},
\end{aligned}$$

just as before. Notice, thus, how the integral domain is now the same for every element. We need only know the vertices of the elements to calculate the scale factor.

5.2 Reference Element in Two Dimensions

We can of course extend this idea of a reference element to two dimensions, and ultimately to any number of dimensions. The general idea is again given, this time in Figure 5.2.

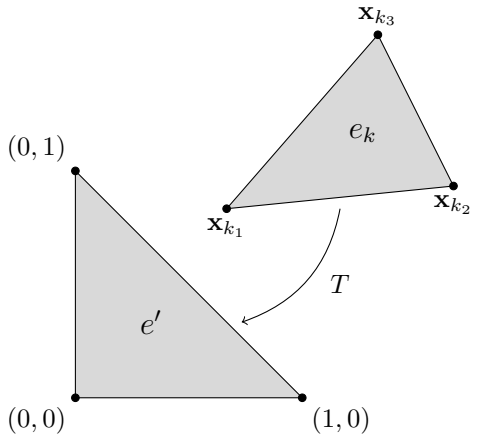


Figure 5.2: Transformation from e_k to e' in 2D

This time, we can say for example that T must satisfy $\mathbf{x}_{k_1} \mapsto (0,0), \mathbf{x}_{k_2} \mapsto (1,0)$ and $\mathbf{x}_{k_3} \mapsto (1,0)$. The rest of the procedures are very similar to before. We define

$$\mathbf{x}(s,t) = \mathbf{x}_{k_1}(1-s-t) + \mathbf{x}_{k_2}s + \mathbf{x}_{k_3}t,$$

where $0 \leq s \leq 1$ and $0 \leq t \leq 1-s$. It is easily verified that this definition satisfies the previously set conditions. This time the Jacobian is a tad more complicated. It is

$$\begin{aligned}\mathcal{J} := \left|\frac{\partial(x,y)}{\partial(s,t)}\right| &= \begin{vmatrix} \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \end{vmatrix} \\ &= (x_{k_2} - x_{k_1})(y_{k_3} - y_{k_1}) - (x_{k_3} - x_{k_1})(y_{k_2} - y_{k_1}).\end{aligned}$$

Then we can integrate f again, as follows:

$$\int_{e_k} f(x,y)\ d\Omega = \mathcal{J}\int_0^1 \int_0^{1-s} f(x(s,t), y(s,t))\ dt\ ds.$$

Note that setting $f \equiv 1$ in the equation above, gives that

$$\int_{e_k} d\Omega = \frac{1}{2}\mathcal{J}.$$

This means that $\mathcal{J}$ is simply twice the area of the original triangle element. The basis functions are of course integrated in a similar manner. We could use similar ideas for differently shaped elements, but this is beyond the scope of this report.

6 FEM and Mixed FEM Implementation and Analysis

We have discussed a lot of theory; it's about time we implement some of it. All of the code used in this chapter was written in Python or Python 3. The Dolfin package from the FEniCS Project [5] was used for the implementation of the FEM and Mixed FEM. This package was specifically created for FEM implementation, and thus is very useful for our purposes. It is assumed that the reader has at least a basic understanding of programming with Python and Python 3. The entirety of the code is given in Appendix C, but snippets of the code are given throughout the chapter to give the reader an easier understanding of what is going on in the programs.

6.1 Poisson Equation with $f = 1$

We again consider the Poisson equation. We now set $\Omega = [-1, 1] \times [-1, 1]$. To keep things simple, we consider homogeneous Dirichlet boundary conditions and set $f = 1$:

$$\begin{cases} -\Delta u = 1, & \text{on } \Omega, \\ u = 0, & \text{on } \partial\Omega. \end{cases}$$

We will not look at the Mixed FEM just yet; we will first check that everything works fine with the standard FEM. Creating a structured grid is very easy with Dolfin. For example, a crossed grid of 16×16 gridpoints on Ω is created as follows:

```
mesh = RectangleMesh(Point(-1.,-1.), Point(1.,1.), 16, 16, "crossed")
```

This will create a mesh that looks like in Figure 6.1.

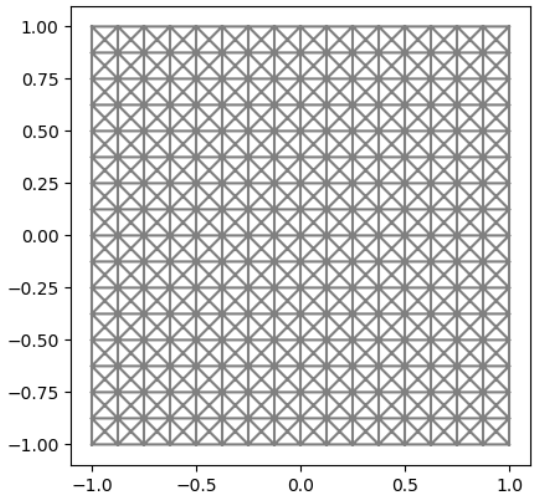


Figure 6.1: A crossed grid structure of 16×16 gridpoints

Notice how this creates triangular elements, just as discussed in Chapter 4. Next, we define the function space for the basis functions, as well as the unknown function. We do this with

```
V = FunctionSpace(mesh, "CG", 1)
```

The second argument means that we will be dealing with Lagrange elements, and the third argument means that the basis functions are of degree 1. Now, we define the unknown function, the basis functions, and the right hand side function:

```
u = TrialFunction(V)
phi = TestFunction(V)
f = Expression("1", degree=1)
```

Dolfin wants to receive the differential equation in the Weak Formulation (or variational form, by Dolfin's documentation). In our case it is:

$$\textbf{Find } u \in V, \textbf{ such that } \quad \int_\Omega \nabla\phi \cdot \nabla u \; d\Omega = \int_\Omega \phi f \; d\Omega, \quad \textbf{for all } \phi \in V.$$

In Dolfin we define the left and right hand side separately:

```
a = (inner(grad(phi),grad(u))) * dx
L = (f*phi) * dx
```

Then we can set the boundary conditions. Dolfin defaults to homogeneous Neumann boundary conditions, but it has a command for Dirichlet boundary conditions. First we need to define the actual boundary, after which we can use `DirichletBC()` to define the Dirichlet boundary conditions:

```
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0.), boundary)
```

The first function uses the variable `on_boundary`, which just checks if `x` is on the natural boundary of Ω. The only thing that remains for us to do is to solve for `u`. Luckily, Dolfin also has a function for this:

```
u = Function(V)
solve(a == L, u, bc)
```

After solving for `u`, one has many options. One can plot the found solution, calculate the norm or the errornorm. The contour plot of the found `u` with a grid of 512×512 looks like in Figure 6.2.

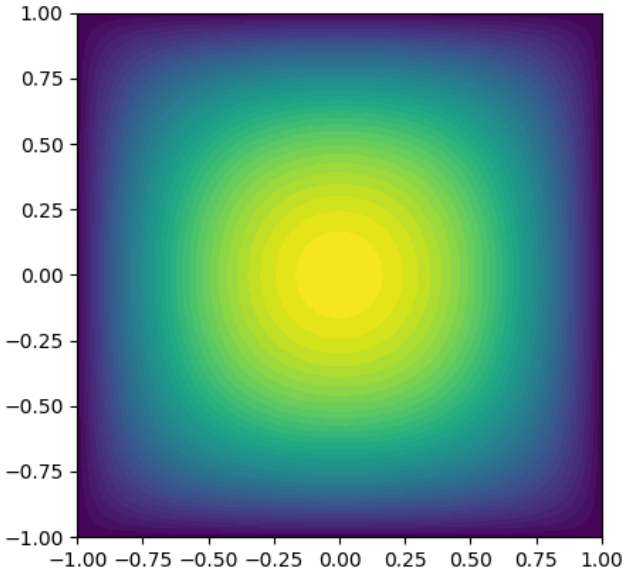


Figure 6.2: Approximation of the Solution to Poisson's Equation

We can use the `norm()` function of the Dolfin package to approximate the order of convergence by Richardson Extrapolation. The order k can be approximated by the following idea [6]:

$$2^k \approx \frac{u_h - u_{h/2}}{u_{h/2} - u_{h/4}} \tag{6.1}$$

We calculated the L^2-norm of `u`, using `norm(u)`, for different gridsizes and applied (6.1). The results are given in Table 6.1, rounded to 6 digits. Obviously, the last two values in the column of 2^k are not known since they require information that has not been calculated, namely the norms at a size of 1024 and 2048.

Size	Norm	2^k
16	0.329148	4.000086
32	0.329856	4.000036
64	0.330033	4.000010
128	0.330077	4.000002
256	0.330088	-
512	0.330091	-

Table 6.1: Results of Richardson Extrapolation on Poisson's Equation

From Table 6.1 it is clear that 2^k converges to 4, ergo k converges to 2. In this case, the method thus converges with order 2. This is comparable to the Finite Difference Method and the Finite Volume Method [1].

6.2 Method of Manufactured Solutions

In an ideal situation, we would want to compare the found approximation to the exact solution. The only problem is that most differential equations do not have a (known) analytical solution. However, one can use the Method of Manufactured Solutions (MMS) [7] to surpass this. Instead of trying to find a solution from a known problem, we assume a solution and work out the right hand side function and boundary conditions from it. For example, for the Poisson equation, we consider $u(x, y) = \sin(x)\cos(y)$. We then calculate the Laplacian of u:

$$\Delta u(x,y) = -2\sin(x)\cos(y).$$

We then have the following differential equation:

$$\begin{cases} -\Delta u = 2\sin(x)\cos(y), & \text{in } \Omega, \\ u(x,y) = \sin(x)\cos(y), & \text{on } \partial\Omega, \end{cases} \tag{6.2}$$

with the known solution $u(x,y) = \sin(x)\cos(y)$. Since we know the exact solution, we can compare the approximation directly to it and work out the convergence rate from that. (6.2) was implemented in Python, again with Dolfin, and the ideas of Richardson Extrapolation were once again applied. The results are given in Table 6.2. The errornorm denotes the L^2-norm of the difference of the approximated and the known solution.

Size	Errornorm	2^k
16	0.009782	1.995301
32	0.004897	1.998825
64	0.002450	1.999706
128	0.001224	1.999926
256	0.000612	-
512	0.000306	-

Table 6.2: Results of Richardson Extrapolation on (6.2)

The errornorm seems to halve every time the gridsize is doubled. This is verified by the 2^k column, which clearly converges to 2, implying that $k = 1$. This is rather surprising, considering the results from before.

6.3 Mixed FEM implementation

The Dolfin package also includes methods for mixed problems. We first consider again the Poisson equation on $\Omega = [-1,1] \times [-1,1]$ with $f = 1$, and formulate the mixed version:

$$\begin{cases} -\nabla \cdot p = 1, \\ \nabla u = p. \end{cases}$$

Since we now want to calculate two different quantities at once — namely, p and u — we need to use two different kind of elements.

```
BDM = FiniteElement("BDM", mesh.ufl_cell(), 1)
DG  = FiniteElement("DG", mesh.ufl_cell(), 0)
W = FunctionSpace(mesh, BDM * DG)
```

Here, `BDM` and `DG` are types of elements. `BDM` (Brezzi-Douglas-Marini) [8] means we will be dealing with gradients, while `DG` (Discontinuous Galerkin) is a simple kind of element. `W` then merges them together into a single function space on the given mesh. We then give the unknown functions (p and u), the basis functions (ϕ and ψ), and the right hand side function f:

```
(p, u) = TrialFunctions(W)
(phi, psi) = TestFunctions(W)
f = Expression("1", degree=1)
```

Next we want to define the two Weak Formulations. However, Dolfin only accepts a single Weak Formulation; we need to add the two together into one equation:

```
a = (-dot(p, phi) + div(phi)*u - div(p)*psi) * dx
L = (-f*psi) * dx
```

We define the boundary just as before, but we need to impose its conditions on a subspace of `W`, since the boundary conditions only apply to u:

```
bc = DirichletBC(W.sub(1), Constant(0.), boundary)
```

Now we simply solve the equation for a function `w` from `W`, and split it into `p` and `u` using `w.split()`, so that we can do some calculations. But first, we appreciate the beauty of the plot of p in Figure 6.3.

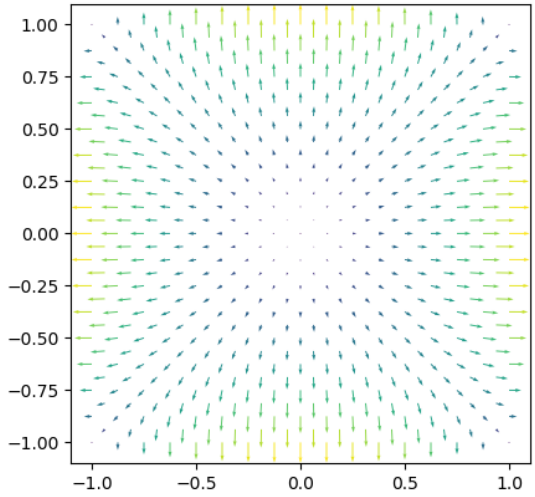


Figure 6.3: p plotted on a 16×16 crossed grid

Now, we look at some results. The simple L^2-norm results for u are exactly the same as before. However, applying the L^2-norm to p alone and applying (6.1) yields some very surprising results. Look in Table 6.3 for these results. We were not able to calculate the result for a size of 512, hence why the table starts at a size of 8.

Size	norm	2^k	k
8	0.749912	13.281402	3.731336
16	0.749875	13.674814	3.773449
32	0.749872	13.969504	3.804209
64	0.749872	14.201231	3.827944
128	0.749872	-	-
256	0.749872	-	-

Table 6.3: Results of Richardson Extrapolation on p after calculating its L^2-norm

Now, catastrophic cancellation might have had an impact on these results since the found values are so close together (the differences cannot even be seen in Table 6.3), but it seems that the value of p converges with an unexpected fourth order!

7 Comparison of Mixed FEM and classic FEM

We have seen that the Mixed FEM gave some counter-intuitive results. It is time to see whether the standard FEM can meet this mark. First, we calculated `u` and `p` from before in the same way. According to Dolfin's documentation, there exists a built-in `grad()` function to suit our purposes, but it did not seem to work. Therefore, another way was sought. We inserted the data from the found values of `u` into .vtu files. We then used the vtk package for Python to read these files. This package was not available for Python 3, thus a new program needed to be written. Because our previous program decided that our grid was unstructured, the vtk unstructured grid reader was used. The .vtu files contain all the information we need to approximate the gradient, i.e. the number of triangles of the discretisation, the triangles themselves, the number of points, and, of course, the values of `u`. However, we encountered some problems when extracting the values of `u` from the .vtu file; the number of values was not equal to the number of grid nodes — as was expected — but rather the number of elements. It is presumed that Dolfin utilises some sort of averaging approach to calculate the value of `u` on every separate element. This did give us a problem on how to calculate the gradient, since it is usually done from the grid node values. A new method was created to circumvent this problem.

7.1 Directly Approximating the Gradient

This method makes use of ideas from the Finite Difference Method, as well as from the Finite Volume Method to accomplish this goal. In this part of the chapter, the words "triangle" and "element" will be used interchangeably. We will assume a left structured grid on a square region, such as the one seen in Figure 7.1.

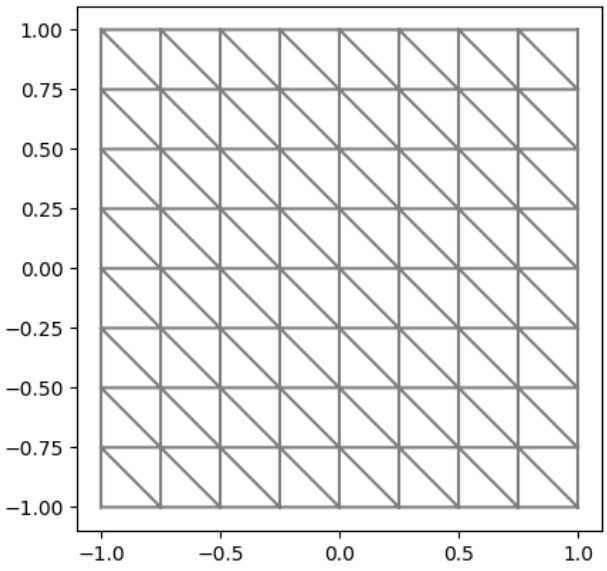


Figure 7.1: A left structured grid of 8×8 grid nodes

To approximate the gradient, we need to know what an element's neighbours are. On a regular square grid, the idea of neighbours is very intuitive: a general square has four neighbours, a North, East, South, and a West neighbour. On a grid with triangular elements this idea is a little bit different. Using Figure 7.1, we decided that we would still treat the element as if it had four neighbours. This idea is visualised in Figure 7.2.

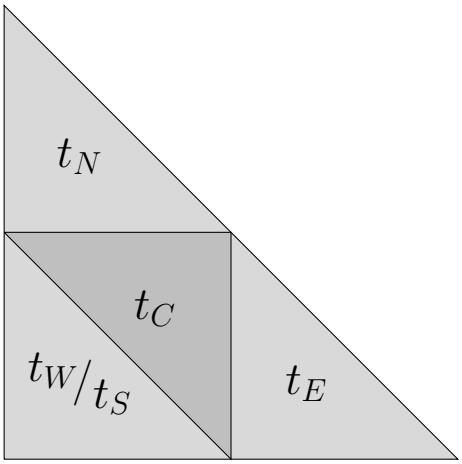


Figure 7.2: The "Four" Neighbours of a Triangular Element t_C

Of course, intuitively, t_W and t_S are the same element, but it turns out that it is easier to treat them as different elements in our following calculations. If the triangle t_C were oriented the other way, t_N and t_E would be the same element.

Now we need to consider the ordering of the elements, as to determine which elements neighbour which. We chose to use a simple and intuitive horizontal numbering system, given in Figure 7.3.

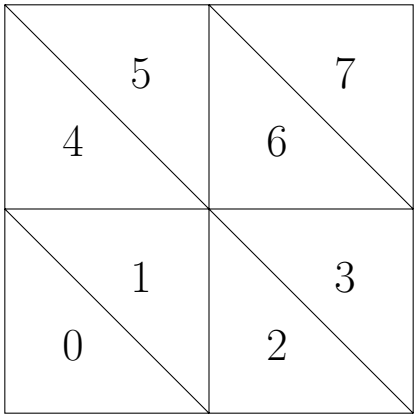


Figure 7.3: Numbering of Triangles on a Left Structured Grid

It is now simple to see that, using this numbering system, a general triangle's East and West neighbours are their actual East and West neighbours, when looking at the indices. For example, Triangle 2 has Triangle 3 as its East and Triangle 1 as its West neighbour. North and South neighbours are a bit more complicated. They depend on what orientation the triangle has. We will need to give a few definitions.

From now on — considering Figure 7.3 — we will call a triangle with 0's orientation *left-pointing* and 1's orientation *right-pointing*. Note that the orientation is directly dependent on whether the triangle's index is even or odd. Additionally, the total number of triangles will be denoted by N. The number of triangles per row (or column) will be denoted by N_r and is easily calculated using N:

$$N_r := \sqrt[+]{2N}.$$

For example, in the case of Figure 7.3, we have $N = 8$ and $N_r = \sqrt[+]{2 \cdot 8} = 4$. We will denote a triangle's index by I with the triangle in question as a subscript. For example, if t_C's index is 5, we write $I_{t_C} = 5$. Using these definitions we find the rules for North and South neighbours:

$$\begin{aligned} &1.\ \text{If } t_C \text{ is right-pointing, } I_{t_S} = I_{t_C} - 1 \text{ and } I_{t_N} = I_{t_C} + N_r - 1; \\ &2.\ \text{If } t_C \text{ is left-pointing, } I_{t_S} = I_{t_C} - N_r + 1 \text{ and } I_{t_N} = I_{t_C} + 1. \end{aligned} \tag{7.1}$$

We also need to consider the elements on the boundary, since they do not quite abide the rules posed in (7.1). An element is considered to be on the boundary if it has at least one of its edges intersect the natural boundary of the region. In Figure 7.3, the boundary elements are the ones with indices 0, 2, 3, 4, 5, and 7. In general, being on a boundary is controlled by the following rules:

$$\begin{aligned} &1.\ \text{If } I_t \mod N_r = 0,\ t \text{ is on the West boundary;} \\ &2.\ \text{If } I_t < N_r \text{ and } I_t \mod 2 = 0,\ t \text{ is on the South boundary;} \\ &3.\ \text{If } I_t > N - N_r \text{ and } I_t \mod 2 \neq 0,\ t \text{ is on the North boundary;} \\ &4.\ \text{If } (I_t + 1) \mod N_r = 0,\ t \text{ is on the East boundary.} \end{aligned} \tag{7.2}$$

These rules may not be that easy to read or intuitively understand, but they are easy to implement in a computer program.

The gradient of u is defined by:

$$\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right).$$

We can approximate each derivative by using a Finite Difference approach. In one dimension, we can approximate the first derivative of u at the point a using

$$\left. \frac{du}{dx} \right|_{x=a} \approx \frac{u(a+h) - u(a-h)}{2h},$$

the *central difference* approach. This central difference approach can work on our triangular grid, if we find a suitable value for h. It turns out that taking h to be half of the length of a straight side of the triangle, it will work perfectly for our purposes. This h can be directly calculated from the edge length of the region in question, and N_r. See Figure 7.4 to see that this approach works. The black dot is the middle of the dark grey triangle.

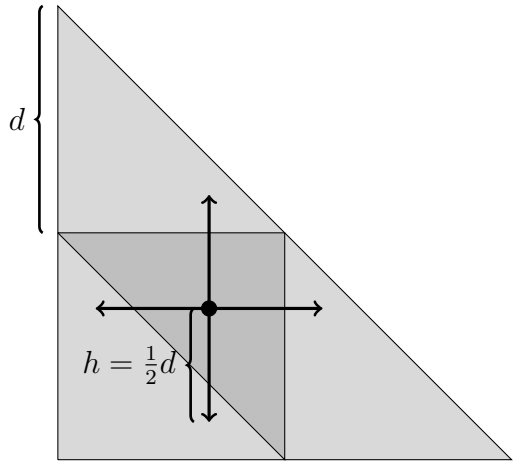


Figure 7.4: Central Difference Approach on a Triangular Grid

Now all the arrow heads land exactly in the neighbour we want them to. Since the value of u is now assumed to be the same everywhere on the same triangle, it does not matter where in the triangle it lands. Now we can define the approximation of ∇u in the triangle t_C on our triangular grid to be

$$\nabla u|_{\mathbf{x} \in t_C} \approx \left(\frac{u_E - u_W}{2h}, \frac{u_N - u_S}{2h} \right),$$

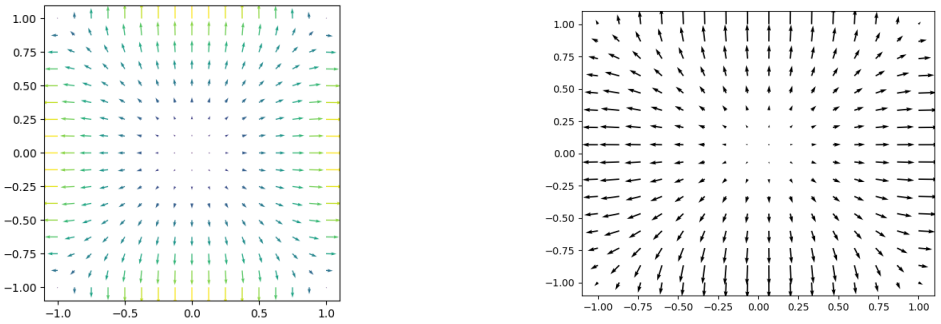
where u_E denotes the value of u on t_E, and similar for u_W, u_N and u_S. If a boundary element is encountered, its non-existent neighbour is set to have the value of the boundary conditions of the respective boundary.

7.2 Implementing the Method

We first took the values `u` and converted it to a NumPy array, using the a separate NumPy support package for vtk:

```
u = vtk_to_numpy(data.GetCellData().GetArray(0))
```

We then calculated the number of triangles per row, using the definition of N_r from before, and used this to implement the neighbour rules from before. These were then used to approximate the gradient as discussed before. We can see the comparison of the found gradients using the Mixed FEM, and using our approximation method on a 16×16 left structured grid in Figure 7.5.



(a) Gradient found with Dolfin

(b) Gradient found with Gradient Extrapolation

Figure 7.5: Comparison of the Gradients found using the Mixed FEM and gradient extrapolation

We can see that the two figures at least look like each other, but of course, this does not tell us everything we need to know. The real test lies in comparing the norms and using Richardson Extrapolation to approximate the order of convergence. The order of converge of the Mixed FEM was already found to be 4. Now let's see how our constructed method holds up. We calculated the L^2-norm manually using its definition. The results are in Table 7.1.

Size	k
8	1.000110
16	1.000007
32	1.000000
64	-
128	-

Table 7.1: Richardson Extrapolation on Gradient Extrapolation

It is almost indisputable that the gradient extrapolation method converges with an order of 1. This is certainly a much worse result than the order of 4 from before. It must be noted however that the gradient extrapolation was built on a Finite Difference approach, which itself, theoretically, converges with order 2. It was therefore not expected that the method would perform better than this.

8 Discussion

After analysing our findings, we found some surprising results that could do with some further investigation. The first surprising result was that the Method of Manufactured Solutions yielded a different result for the order of convergence than was found earlier. Further investigation might be required to understand why this happened.

The next unexpected result was that the Mixed FEM seemed to converge with an order of 4, when only looking at p and its L^2-norm. This is very remarkable, even more so considering that u "only" converged with an order of 2. As argued before, catastrophic cancellation might be an explanation, or something else entirely might have influenced the results. Again, further investigation will be required to conclude if this is the correct result.

Some final remarks will be made on our own method. First of all, it should not have been necessary to create it in the first place, since the FEM should calculate the values on the grid nodes, rather than per element. One could search for another package or program for implementing the FEM, such that these values are calculated where they should, in theory, be.

9 Conclusion

The main aim of this report was to give a comparison between the Mixed FEM and the standard FEM. It did this by giving the results of the Mixed FEM, while also suggesting a method for calculating the gradient without this method. The theory of these methods was first extensively explained, to give the reader a fair knowledge of what is going on behind the scenes, so to say.

Looking at the results of the classic FEM on Poisson with $f = 1$, we found a convergence estimation of order 2, when looking at the L^2-norm. This is an expected result, since most simple numerical methods converge with this order. As discussed, the fourth order convergence of the Mixed FEM was not expected, and might not even be correct, but it does imply that it yields more accurate results than those found with our constructed method. These results clearly suggest that the Mixed FEM is the way to go, when wanting to calculate ∇u as well as u. It not only gives better results, it is also easier to implement in a Python program, by utilising the built-in functions of the Dolfin package.

— Appendix —

- A

Higher Order Basis Functions

The ideas presented here follow from [2]. We have seen how to construct simple linear basis functions. It is however possible to construct higher order basis functions. This section of the appendix will explain how.

Say we want basis functions of order d. We then need $d+1$ nodes per element. For example, we will construct quadratic (so order 2) basis functions ϕ_{k-1}, ϕ_k and ϕ_{k+1} on the element containing the 3 nodes x_{k-1}, x_k and x_{k+1}. For simplicity, but without loss of generality, we assume that this element does not intersect the boundary of the region. We start with ϕ_{k-1}. It must satisfy

$$\phi_{k-1}(x_i) = \begin{cases} 1, & \text{if } i = k-1, \\ 0, & \text{otherwise.} \end{cases} \tag{A.1}$$

We achieve this by using the *Lagrange Polynomial*; in this case of degree 2. Given three data points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, we can construct it. In our case they are $(x_{k-1}, 1), (x_k, 0), (x_{k+1}, 0)$. Then

$$L(x) := \sum_{j=0}^{2} y_j l_j(x),$$

with

$$l_j(x) := \prod_{\substack{0 \le i \le 2 \\ i \ne j}} \frac{x - x_i}{x_j - x_i}$$

will satisfy (A.1). ϕ_k and ϕ_{k+1} are made with the same formulae, with their corresponding y_js. The end result is given in Figure A.1, with a reference to the linear equivalences.

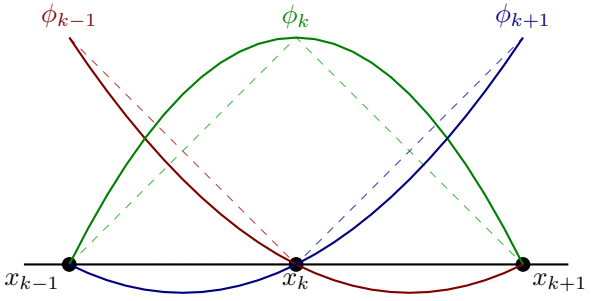


Figure A.1: Quadratic basis functions. The linear basis functions are dashed

We see that the functions sometimes dip below zero, while this was not the case in the linear setting. Note, however, that we still have that the sum of the basis functions is 1. It must also be noted that these quadratic basis functions are symmetric around their respective grid node. These ideas can easily be extended to even higher order basis functions.

B

A Note on the Partial Integration Step

One thing has to be addressed about the partial integration step in Chapter 3, namely that we could have chosen to partially integrate (3.2b) instead of (3.2a):

$$\int_0^1 \psi u' \, dx = [\psi u]_0^1 - \int_0^1 \psi' u \, dx = -\int_0^1 \psi' u \, dx.$$

This would make (3.4) look like:

$$\begin{cases} -\sum_{j=0}^{n} p_j \int_0^1 \frac{d\psi_j}{dx} \phi_i \, dx = \int_0^1 \phi_i f \, dx, & \text{for all } i, \\ \sum_{j=0}^{n} p_j \int_0^1 \psi_j \psi_i \, dx + \sum_{j=0}^{n} u_j \int_0^1 \phi_j \frac{d\psi_i}{dx} \, dx = 0, & \text{for all } i. \end{cases}$$

The resulting matrix-vector equation has a structure very similar to that of (3.5):

$$\begin{bmatrix} -A & \mathbf{0} \\ B & A^\top \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix},$$

with

$$A_{ij} = \int_0^1 \phi_i \frac{d\psi_j}{dx} \, dx, \qquad B_{ij} = \int_0^1 \psi_i \psi_j \, dx \quad \text{and} \quad \mathbf{f}_i = \int_0^1 \phi_i f \, dx.$$

This is very similar to what was found previously; it is mainly up to personal preference which is chosen. However, the problem itself may cause one of these to be more desirable.

C Complete Code

C.1 Poisson FEM

```
from dolfin import *
import matplotlib.pyplot as plt

i = 0
k = 2

u_lst = []

while i <= 7:

    mesh = RectangleMesh(Point(-1.,-1.),Point(1.,1.), k, k, "crossed")
    V = FunctionSpace(mesh, "CG", 2)
    u = TrialFunction(V)
    phi = TestFunction(V)

    f = Expression("1", degree=1)

    a = (inner(grad(phi),grad(u))) * dx
    L = (f*phi) * dx

    # Define essential boundary
    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, Constant(0.), boundary)

    u = Function(V)
    solve(a == L, u, bc)

    print(k, norm(u))
    u_lst.append(norm(u))

    i += 1
    k *= 2

print("")

for i in range(len(u_lst)-2):
    print( (u_lst[i] - u_lst[i+1]) / (u_lst[i+1] - u_lst[i+2]) )

plot(u)
plt.show()
```

C.2 Method of Manufactured Solutions

```python
from dolfin import *
import matplotlib.pyplot as plt

i = 0
k = 4

norm_lst = []

while i <= 6:

    mesh = RectangleMesh(Point(-1.,-1.),Point(1.,1.), k, k, "crossed")
    V = FunctionSpace(mesh, "CG", 2)
    u = TrialFunction(V)
    phi = TestFunction(V)

    f = Expression("2*sin(x[0])*cos(x[1])", degree=1)

    a = (inner(grad(phi),grad(u))) * dx
    L = (f*phi) * dx

    # Define essential boundary
    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, Expression("sin(x[0])*cos(x[1])", degree=1), boundary)

    u = Function(V)
    solve(a == L, u, bc)

    u_e = Function(V)

    u_e.assign(Expression("sin(x[0])*cos(x[1])", degree=1))

    plt.figure()
    plot(u)
    plt.figure()
    plot(u_e)
    plt.show()

    plt.figure()
    plot(u_e)

    plt.figure()
    plot(u)

    u_e = u_e.vector()
    u = u.vector()

    print(k, norm(u-u_e))
    norm_lst.append(norm(u-u_e))

    i += 1
    k *= 2

for i in range(len(norm_lst)-2):
    print( (norm_lst[i] - norm_lst[i+1]) / (norm_lst[i+1] - norm_lst[i+2]) )

plt.show()
```

C.3 Mixed FEM

```
from dolfin import *
import matplotlib.pyplot as plt
import math

norm_lst = []
sigma_lst = []

for i in [8,16,32,64,128]:

    # Create mesh
    mesh = RectangleMesh(Point(-1.,-1.), Point(1.,1.), i, i, "left")

    # Define finite elements spaces and build mixed space
    BDM = FiniteElement("BDM", mesh.ufl_cell(), 1)
    DG  = FiniteElement("DG", mesh.ufl_cell(), 0)
    W = FunctionSpace(mesh, BDM * DG)

    # Define trial and test functions
    (sigma, u) = TrialFunctions(W)
    (tau, v) = TestFunctions(W)

    # Define source function
    f = Expression("1", degree=1)

    # Define variational form
    a = (-dot(sigma, tau) + div(tau)*u - div(sigma)*v)*dx
    L = -f*v*dx

    # Define essential boundary
    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(W.sub(1), Constant(0.), boundary)

    delta = PointSource(W.sub(0),Point(0.,0.), 10)

    # Compute solution
    w = Function(W)
    solve(a == L, w, bc)
    (sigma, u) = w.split()

    plt.figure()
    plot(u)
    plt.figure()
    plot(sigma)

    stru = "mpoisson_u" + str(i) + ".pvd"
    strs = "mpoisson_p" + str(i) + ".pvd"

    file = File(stru)
    file << u
    file = File(strs)
    file << sigma

    print(i, norm(u, "L2", mesh), norm(sigma, "L2"))

    norm_lst.append(norm(u))
    sigma_lst.append(norm(sigma))

print(norm_lst)
for i in range(len(norm_lst)-2):
    norm_u = (norm_lst[i] - norm_lst[i+1]) / (norm_lst[i+1] - norm_lst[i+2])
    norm_s = (sigma_lst[i] - sigma_lst[i+1]) / (sigma_lst[i+1] - sigma_lst[i+2])
    ul = math.log(norm_u, 2)
    sl = math.log(norm_s, 2)
    print(norm_u ,norm_s, ul, sl)

# Plot sigma and u
plt.show()
```

C.4 Gradient Approximation

```
import numpy as np
import math
import vtk
from vtk.util.numpy_support import vtk_to_numpy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

sum_lst = []

for j in [8,16,32,64,128]:

    # The source file
    file_name = "/home/xulian/Documents/Bachelor EindProject/Code/mpoisson_u" \
        + str(j) + "000000.vtu"

    # Read the source file.
    reader = vtk.vtkXMLUnstructuredGridReader()
    reader.SetFileName(file_name)
    reader.Update()  # Needed because of GetScalarRange
    data = reader.GetOutput()

    points = data.GetPoints()
    npts = points.GetNumberOfPoints()
    x = vtk_to_numpy(points.GetData())

    triangles = vtk_to_numpy(data.GetCells().GetData())
    ntri = triangles.size//4

    tri = np.take(triangles,[n for n in range(triangles.size) if n%4 != 0]).reshape
    (ntri,3)

    n_arrays = reader.GetNumberOfPointArrays()

    u = vtk_to_numpy(data.GetCellData().GetArray(0))

    #compute gradient
    gradsx = []
    gradsy = []
    nd = int(np.sqrt(ntri*2))
    h2 = 4./nd
    for i in range(len(u)):

        #grad_x
        if i%nd == 0:
            grad_x = u[i+1]/h2
        elif (i+1)%nd == 0:
            grad_x = -u[i-1]/h2
        else:
            grad_x = (u[i+1]-u[i-1])/h2

        #grad_y
        if i<nd and i%2 == 0:
            grad_y = u[i+1]/h2
        elif i>ntri-nd and i%2 != 0:
            grad_y = -u[i-1]/h2
        elif i%2 == 0:
            grad_y = (u[i+1] - u[i-nd+1])/h2
        else:
            grad_y = (u[i+nd-1] - u[i-1])/h2

        gradsx.append(-grad_x)
        gradsy.append(-grad_y)

    new_gradsx = []
    new_gradsy = []

    for i in range(len(gradsx)):
        if i%2 == 0:
            new_gradsx.append(gradsx[i])
```

```
            new_gradsy.append(gradsy[i])

    g = np.zeros(len(new_gradsx))

    for i in range(len(new_gradsx)):
        g[i] = (new_gradsx[i]*new_gradsx[i] + new_gradsy[i]*new_gradsy[i])

    g_2 = np.reshape(g, (j,j))

    sum_g = 0.
    for k in range(len(g)):
        sum_g += g[i]

    sum_g = math.sqrt(sum_g)
    sum_lst.append(sum_g)

    X, Y = np.meshgrid(np.linspace(-1, 1, nd/2), np.linspace(-1, 1, nd/2))

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    plt.xlim(-1.1,1.1)
    plt.ylim(-1.1,1.1)
    ax.plot_surface(X, Y, g_2)

    fig1, ax1 = plt.subplots()
    plt.xlim(-1.1,1.1)
    plt.ylim(-1.1,1.1)
    Q = ax1.quiver(X, Y, new_gradsx, new_gradsy, units='width')

plt.show()

for l in range(len(sum_lst)-2):
    print ( sum_lst[l]-sum_lst[l+1] ) / ( sum_lst[l+1]-sum_lst[l+2] )
```

References

[1] J. van Kan & A. Segal & F. Vermolen. *Numerical Methods in Scientific Computing*. Delft Academic Press, Delft, 2014.

[2] Hans Petter Langtangen. Introduction to finite element methods, 2013. Retrieved on the 16th of May 2019 from `http://hplgit.github.io/INF5620/doc/pub/sphinx-fem/._main_fem003.html#`.

[3] Yousef Saad. *Iterative Methods for Sparse Systems (2nd Edition)*. SIAM, 2003.

[4] F.J. Vermolen, may-june 2019. Personal Interviews.

[5] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017.

[6] C. Vuik & F.J. Vermolen & M.B. van Gijzen & M.J. Vuik. *Numerical Methods for Ordinary Differential Equations*. Delft Academic Press, 2015.

[7] S. J. Hulshoff. Ae2220 ii, computational modelling. Technical report, TU Delft, 2017.

[8] Franco Brezzi & Jim Douglas & L. Donatella Marini. Two families of mixed finite elements for second order elliptic problems. *Numerische Mathematik*, 1985.

Front Page images:

- Dolphin Mesh image:
  Langtangen, H.P. (2013). *Dolfin-mesh* [digital image]. Retrieved from `http://hplgit.github.io/INF5620/doc/pub/sphinx-fem/._main_fem000.html`.
- TU Delft Logo:
  TU Delft. (2019). *TU Delft Logo* [digital image]. Retrieved from `https://www.kokenmetkennis.nl/portfolio/tu-delft/tu-delft-logo/`.