B.Sc.Eng. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Mesh Adaptive Techniques for Finite Element Solvers

Matias Fjeldmark (s113202)

Kongens Lyngby 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The theoretical foundation of the finite element method is presented and the major mathematical results from finite element analysis are presented and some are proven.

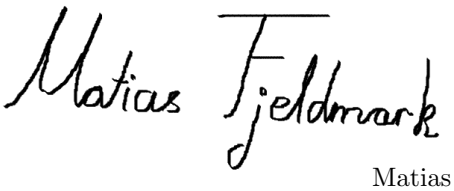
The implementation of two methods for solving steady state problems are presented. In particular, we show the implementation of an Adaptive Mesh Refinement algorithm, and its performance is evaluated. We find that the adaptive algorithm greatly outperforms a standard finite element method. Methods for handling time-dependent problems with the finite element method are also presented, along with an implementation of one of those methods.

`MATLAB` code of the implemented finite element solver is provided, which can solve the Poisson problem and the heat equation on a rectangular domain.

Preface

The following B.Sc.Eng. thesis was prepared at the department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark in fulfilment of the requirements for acquiring the degree bachelor of science in engineering. It has been written in the period from the 8^{th} of September 2014 till the 30^{th} of January 2015 and has a weight of 15 ECTS-points.

Kongens Lyngby, January 30, 2015



Matias Fjeldmark (s113202)

Acknowledgements

I would like to thank my advisor Allan P. Engsig-Karup for making this project possible, and for being available for any assistance I needed throughout the whole project.

Contents

List of symbols

$\forall$	"For all."
$\|\| \cdot \|\|$	Norm.
$\langle \cdot, \cdot \rangle$	Inner product.
$\mathbf{u}$	Vector.
$\mathbf{A}$	Matrix.
$\mathbf{I}$	Identity Matrix.
∇	Nabla.
$\nabla \mathbf{u}$	Divergence of $\mathbf{u}$.
∇f	Gradient of f.
$\mathcal{V}$	Vector space.
$\mathcal{H}$	Hilbert space.
$\mathcal{V}'$	Dual space of $\mathcal{V}$.
Ω	Domain.
$\partial\Omega$	Boundary of domain.

CHAPTER 1

Introduction

Differential equations have been of great interest to mathematicians and physicists for centuries – an interest that arose during the time of Isaac Newton and Gottfried Leibniz in the 17$^{\text{th}}$ century when they invented calculus. Even though the differential equation now is a couple of centuries old, mathematicians continue to study them to this day, and it is not without good reason. It turns out that many natural phenomena can be described by differential equations. Examples include, but are certainly not limited to, the motion of fluids, the growth of populations, temperature changes, the behaviour of waves, and the behaviour of electric and magnetic fields. The solutions to these equations describe the position of the fluid, the population size, the temperature etc., and therefore finding the solution will give us the ability to predict the behaviour of these various natural phenomena without physically testing them.
To demonstrate an example of a differential equation it seems fitting to use one of the earliest examples: Newton's Second Law of Motion. This law describes how the net force acting on an object is equal to the rate of change of its linear momentum and is mathematically stated as

$$\mathbf{f} = \frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t},$$

where $\mathbf{f}$ is the net force, $\mathbf{p}$ is the linear momentum, and t is time. The linear momentum of an object is equal to its mass m times its velocity $\mathbf{v}$, and if the object is of constant mass the equation reduces to its more known form:

$$\mathbf{f} = \frac{\mathrm{d}(m\mathbf{v})}{\mathrm{d}t} = m\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = m\mathbf{a}.$$

The net force acting on an object (of constant mass) is equal to its mass times its acceleration.

Although mathematicians have come a long way since the times of Newton and Leibniz in the area of differential equations, it unfortunately turns out that many differential equations are very hard to solve. Luckily, in practice we rarely need an infinite precision in our solution so we are content with a good approximation of the solution. This is where numerical methods enter the picture. Numerical methods that solve differential equations usually attempt to transform the differential equation into several "normal" equations, which are more easily solved, and the more equations we transform it into, the preciser our approximation is to the true solution to the differential equation.
One of the most commonly used numerical methods to solve differential equations is the *Finite Element Method* and it is the basis of this text. In the following chapters it will be explained what exactly the method is, the theory behind it and why it works, and how one can program a computer-implementation of the method using `MATLAB`. However, let us first give a bit of intuition for what the finite element method is.

The idea behind the finite element method is that most functions can be locally approximated by other very simple functions. This idea should be familiar to anyone who has studied calculus since a tangent to a function at a point is simply a linear approximation at that point. In the finite element method, however, it does not have to be linear approximations, though it often it. Instead, we just consider any type of functions, which we call the basis functions, and the idea is then if we use enough of these simple functions we will end up with an approximation that is precise enough for our needs.

Consider a differential equation, for now it does not really matter which, and let us say we are looking for a solution $u(x, y)$. This means that we have a 2-dimensional domain, a plane, with dimensions x

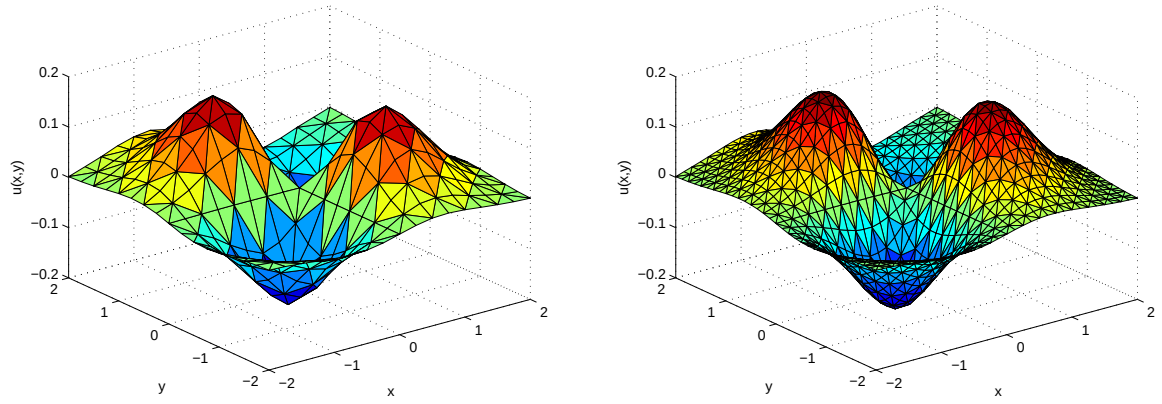


Figure 1.1: *Two typical FE approximations. Precision is gained when using more elements.*

and y. The solution could $u(x, y)$ could, for example, represent the temperature at the point (x, y). We then restrict us to some small portion of the plane, this could be a square $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. Next we divide this square into many, small triangles, which we call elements. Within each of these elements we approximate the the solution by the simple basis functions, ending up with a solution that is defined piecewise by our basis functions. It may seem very challenging at first to use hundreds, if not thousands, of functions to describe a single function, but the point is that these basis functions are very simple, and computers are perfect for the job of performing many, simple, calculations. In figure (1.1) an example of two finite element approximations can be seen, where the basis functions are simple linear functions. Furthermore, the figure illustrates how dividing the domain into smaller elements increases the accuracy.

CHAPTER 2

Theory

The finite element method has been used as a method of approximating solutions to differential equations for many years, and consequently it has a strong theoretical foundation behind it. In the following chapter the most important theoretical concepts behind the method will be introduced. It will be explained how one sets up the *Weak Formulation*, and we will give proof of some particularly nice results that follow if the weak formulation has a certain form, e.g. for Poisson's problem. These mathematical results guarantee existence and uniqueness of the approximation, and the error of the approximation will be analysed. Before we get to these results, though, a number of more basic concepts have to be introduced as they will be used without explanation throughout the text. For the interested reader, the following sources can be used to read the subjects in greater detail. The preliminary theory is mostly based on [1]. The subject of the weak formulation can be found in most places that discuss the finite element method, e.g. [2],[3], or [4]. The more abstract mathematical concepts such as functional analysis or the study of dynamical systems can be found in [5] and [6], respectively.

2.1 Preliminary Theory

The first that we will introduce is the concept of a *norm* of a vector or a function.

Definition 1 (Norm)**.** *Let $\mathcal{V}$ be a (complex) vector space. A norm on $\mathcal{V}$ is a function*

$$\|\cdot\| : \mathcal{V} \to \mathbb{R}$$

that satisfies the following conditions:

(i) $\|\mathbf{v}\| \geq 0, \quad \forall \mathbf{v} \in \mathcal{V}$ *and* $\|\mathbf{v}\| = 0 \iff \mathbf{v} = \mathbf{0}$ *(Nonnegativity)*

(ii) $\|\alpha\mathbf{v}\| = |\alpha| \cdot \|\mathbf{v}\|, \quad \forall \mathbf{v} \in \mathcal{V}, \alpha \in \mathbb{C}$ *(Absolute scalability)*

(iii) $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|, \quad \forall \mathbf{v}, \mathbf{w} \in \mathcal{V}$ *(Triangle inequality)*

A vector space equipped with a norm is called a normed vector space.

The norm of an object is a quantity that in some (possibly abstract) sense describes the length, size, or extent of the object. An example of a norm that should be familiar is the distance of a point in the plane, say $(x, y)^\intercal \in \mathbb{R}^2$, to the origin which is given by

$$\|(x, y)^\intercal\|_2 = \sqrt{x^2 + y^2}$$

which is just an application of the Pythagorean Theorem, and it is also known as the Euclidean distance. In the finite element method we are particularly interested in the norm (i.e. size) of the error of our approximation, compared to the true (and usually unknown) solution. The error must be the difference between the approximation u_h and the true solution u, and hence its size is given by

$$\varepsilon = \|u - u_h\|$$

for some type of relevant norm. The hope is that our method can guarantee a sufficiently small ε. When solving differential equations, one seeks to find a *function* that solves the equations, and therefore

it should not come as a surprise that spaces containing functions and vectors are important. The first important type of space is the *Banach* space. Firstly, we introduce a related key definition:

Definition 2 (Cauchy sequence)**.** *Let $\mathcal{V}$ be a normed vector space. A sequence $(\mathbf{v}_k)_{k=1}^{\infty}$ of elements in $\mathcal{V}$ is a Cauchy sequence if for any $\varepsilon > 0$ there exists an $N \in \mathbb{N}$ such that*

$$\|\mathbf{v}_k - \mathbf{v}_\ell\| \leq \varepsilon \quad \textit{for all } k, \ell \geq N.$$

Intuitively, this means that the distance between the elements of a Cauchy sequence $(\mathbf{v}_k)_{k=1}^{\infty}$ gets arbitrarily close to each other, whenever k is large enough. At first glance, it might seem like the term Cauchy sequence simply is another name for a convergent sequence, but there is a small yet important difference. We define a convergent sequence as follows:

Definition 3 (Convergence in normed spaces)**.** *A sequence $(\mathbf{v}_k)_{k=1}^{\infty}$ in a normed vector space $\mathcal{V}$ is said to be convergent to $\mathbf{v} \in \mathcal{V}$ if*

$$\lim_{k \to \infty} \|\mathbf{v} - \mathbf{v}_k\| = 0. \tag{2.1}$$

Equivalently, we write

$$\lim_{k \to \infty} \mathbf{v}_k = \mathbf{v}.$$

In a normed vector space $\mathcal{V}$, a convergent sequence is also necessarily a Cauchy sequence, but not vice versa. The difference is that when we speak of convergence of a sequence $(\mathbf{v}_k)_{k=1}^{\infty}$ in $\mathcal{V}$ to some element $\mathbf{v}$, it is required that the limit element $\mathbf{v}$ also is contained in the space, i.e $\mathbf{v} \in \mathcal{V}$. This brings us to the definition of the Banach space.

Definition 4 (Banach space)**.** *A normed vector space $\mathcal{V}$ is called a Banach space if every Cauchy sequence $(\mathbf{v}_k)_{k=1}^{\infty}$ converges to some $\mathbf{v} \in \mathcal{V}$.*

Banach spaces are very important and have been studied greatly, but in this text we are particularly interested in a special kind of Banach space, namely *Hilbert spaces*. This is the last major concept that will be introduced before we will return to the differential equations, but to explain what a Hilbert space is, we need to define the term *inner product*:

Definition 5 (Inner product)**.** *Let $\mathcal{V}$ be a (complex) vector space. An inner product on $\mathcal{V}$ is a mapping*

$$\langle \cdot, \cdot \rangle : \mathcal{V} \times \mathcal{V} \to \mathbb{C}$$

that satisfies the following conditions:

(i) $\langle \mathbf{v}, \mathbf{v} \rangle \geq 0, \quad \forall \mathbf{v} \in \mathcal{V}$ *and* $\langle \mathbf{v}, \mathbf{v} \rangle = 0 \iff \mathbf{v} = \mathbf{0}$ *(Nonnegativity)*

(ii) $\langle \alpha \mathbf{v} + \beta \mathbf{w}, \mathbf{u} \rangle = \alpha \langle \mathbf{v}, \mathbf{u} \rangle + \beta \langle \mathbf{w}, \mathbf{u} \rangle, \quad \forall \mathbf{v}, \mathbf{w}, \mathbf{u} \in \mathcal{V}, \, \alpha, \beta \in \mathbb{C}$ *(Linearity in first argument)*

(iii) $\langle \mathbf{v}, \mathbf{w} \rangle = \overline{\langle \mathbf{w}, \mathbf{v} \rangle}, \quad \forall \mathbf{v}, \mathbf{w} \in \mathcal{V}$ *(Conjugate symmetry)*

A vector space equipped with an inner product is called an inner product space.

The inner product between two objects is a quantity that in some (possibly abstract) sense allows us to measure the angle between the objects. When an inner product space is a real vector space, the conditions can be simplified further. The conjugate symmetry is replaced by pure symmetry, i.e.

$\langle \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{w}, \mathbf{v} \rangle$. The symmetry in the real inner product also leads to linearity in the second argument, since

$$\langle \mathbf{v}, \alpha\mathbf{w} + \beta\mathbf{u} \rangle = \langle \alpha\mathbf{w} + \beta\mathbf{u}, \mathbf{v} \rangle = \alpha\langle \mathbf{w}, \mathbf{v} \rangle + \beta\langle \mathbf{u}, \mathbf{v} \rangle = \alpha\langle \mathbf{v}, \mathbf{w} \rangle + \beta\langle \mathbf{v}, \mathbf{u} \rangle.$$

In this text most vector spaces will be real, simply because most (not all) differential equations that govern natural phenomena are real, not complex. An operator with two arguments that is linear in both is called bilinear. Some bilinear operators turn out to have important properties which will be discussed later.
Every inner product space is a normed space with respect to the norm given by

$$\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} \tag{2.2}$$

as this satisfies all of the conditions given in the definition of a norm. This leads us to the definition of a Hilbert space.

Definition 6 (Hilbert space)**.** *A vector space equipped with an inner product $\langle \cdot, \cdot \rangle$, which is a Banach space with respect to the norm (2.2) is called a Hilbert space.*

We can now already see that Hilbert spaces have some nice properties. They are spaces with inner products and have a notion of lengths between elements (norm), additionally they have the property that every Cauchy sequence converges (since they also are Banach spaces). Banach spaces are what we call *complete*, which intuitively means that there are no "holes," compared to an example such as the rational numbers, which has an infinite number of holes filled with irrational numbers.
In any inner product space a very important inequality holds which will be used later, namely the Cauchy-Schwarz Inequality:

Theorem 1 (Cauchy-Schwarz Inequality)**.** *Let $\mathcal{V}$ be a vector space with inner product $\langle \cdot, \cdot \rangle$. Then*

$$|\langle \mathbf{v}, \mathbf{w} \rangle| \leq \langle \mathbf{v}, \mathbf{v} \rangle^{1/2} \langle \mathbf{w}, \mathbf{w} \rangle^{1/2}, \quad \forall \mathbf{v}, \mathbf{w} \in \mathcal{V} \tag{2.3}$$

If $\mathcal{V}$ is a Hilbert space with respect to the induced norm (2.2), this is equivalent to

$$|\langle \mathbf{v}, \mathbf{w} \rangle| \leq \|\mathbf{v}\| \|\mathbf{w}\|, \quad \forall \mathbf{v}, \mathbf{w} \in \mathcal{V} \tag{2.4}$$

Lastly, some notation will be explained. First off is the ∇-notation. This is an operator that in the 3-dimensional case is

$$\nabla = \begin{pmatrix} \dfrac{\partial}{\partial x_1} \\ \dfrac{\partial}{\partial x_2} \\ \dfrac{\partial}{\partial x_3} \end{pmatrix}.$$

The divergence of a vector, $\mathbf{v}$, is written $\nabla\mathbf{v}$ (the symbol used is named nabla). The divergence of a vector is the dot product between ∇ and the vector. Consider a vector in 3-dimensional space with dimensions x, y, and z, say $\mathbf{v} = (v_1, v_2, v_3)$. Then

$$\nabla\mathbf{v} = \frac{\partial v_1}{\partial x} + \frac{\partial v_2}{\partial y} + \frac{\partial v_3}{\partial z},$$

which easily translates into vectors of any dimension. When nabla is acting on a function, it is then simply the gradient of the function, i.e.

$$\nabla f = \begin{pmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial z} \end{pmatrix} \cdot f = \begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \\ \dfrac{\partial f}{\partial z} \end{pmatrix}.$$

When there only is one spatial dimension, the gradient is simply the usual derivative. When writing $\nabla^2 f$ what is meant is $\nabla \cdot (\nabla f)$, i.e. the dot product between ∇ and ∇f. In the 2-dimensional case this is

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}. \tag{2.5}$$

One can verify that the gradient of a function follows the product rule just as one would hope, that is

$$\nabla(f \cdot g) = \nabla f \cdot g + f \cdot \nabla g.$$

Integrals will be used often, so we will clarify the notation first. Let $\Omega \subset \mathbb{R}^2$ be a connected domain. Then the integral

$$\int_\Omega f \, \mathrm{d}S$$

is the integral over the entire domain. As an example, if $\Omega = (0,1) \times (0,2)$ is a rectangular domain then

$$\int_\Omega f \, \mathrm{d}S = \int_0^1 \int_0^2 f(x,y) \, \mathrm{d}y \, \mathrm{d}x,$$

whereas the suffix $\mathrm{d}s$ denotes a line integral, which in this text only will be used as an integral over the boundary of the domain, $\partial\Omega$.

2.2 The Weak Formulation

The concept of the weak formulation is an important one in the area of finite element. To introduce this concept we will, as an example, look at Poisson's Problem:

$$-\nabla^2 u = f \quad \text{in } \Omega \subset \mathbb{R}^d \tag{2.6}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{2.7}$$

This is sometimes called the strong formulation of the problem, and is no more or less than just a precise formulation of a partial differential equation problem. Problems like this are usually referred to as *boundary value problems* (BVP), since we have a differential equation along with a *boundary condition* (BC). This particular type of boundary condition, where one knows the values of the solution on the boundary $\partial\Omega$ of a domain Ω, are referred to as Dirichlet boundary conditions. Another type of boundary condition is the *Neumann boundary condition*, where the solution's gradient is known instead. However, we will mainly stick to Dirichlet boundary conditions. This text is mostly concerned with the case where spatial domain is 2-dimensional, i.e. $\Omega \subset \mathbb{R}^2$, so this will be assumed from now on. With that said, the following arguments are easily translated to any dimension d. To derive the weak formulation of Poisson's problem we will need to introduce a theorem. This theorem is called the Divergence Theorem, or sometimes Gauss's Theorem, and in the 2-dimensional case can be stated as follows:

Theorem 2 (Divergence)**.** *Let Ω be a region in the plane with boundary $\partial\Omega$. Then it holds that*

$$\int_\Omega \nabla \mathbf{f} \,\mathrm{d}S = \int_{\partial\Omega} \mathbf{f} \cdot \mathbf{n} \,\mathrm{d}s \tag{2.8}$$

where $\mathbf{n}$ is the outward normal vector to the boundary.

The divergence theorem is a mathematical statement of the physical fact that the density within a region of space can change only by having it flow into or away from the region through its boundary. Consider now the equation (2.6). To establish the weak formulation, multiply both sides of the equation by a test function v satisfying the homogeneous boundary conditions

$$v(\partial\Omega) = 0.$$

After multiplying by this test function, we integrate both sides over the domain which yields

$$-\int_\Omega v\nabla^2 u \,\mathrm{d}S = \int_\Omega v f \,\mathrm{d}S.$$

Using the product rule

$$\nabla(v\nabla u) = \nabla v \nabla u + v\nabla^2 u \qquad \Longrightarrow \qquad v\nabla^2 u = \nabla(v\nabla u) - \nabla v \nabla u$$

lets us rewrite the left-hand side, i.e.

$$-\int_\Omega v\nabla^2 u \,\mathrm{d}S = \int_\Omega \nabla v \nabla u \,\mathrm{d}S - \int_\Omega \nabla(v\nabla u) \,\mathrm{d}S = \int_\Omega v f \,\mathrm{d}S$$

Using the Divergence Theorem, along with the boundary condition of the test function v, we have that

$$\int_\Omega \nabla(v\nabla u) \,\mathrm{d}S = \int_{\partial\Omega} v\nabla u \cdot \mathbf{n} \,\mathrm{d}s = \int_{\partial\Omega} 0 \cdot \nabla u \cdot \mathbf{n} \,\mathrm{d}s = 0.$$

Therefore the equation simplifies to

$$\int_\Omega \nabla v \nabla u \,\mathrm{d}S = \int_\Omega v f \,\mathrm{d}S. \tag{2.9}$$

This is called the weak formulation of (2.6). The intuition behind the integral and the test functions is this: we want to test if the PDE holds in a weighted average sense, where the weights are given by v, the test function (and hence sometimes referred as the weight function as well)[3]. Now a natural question arises: why do we call it "The Weak Formulation?" First off, just because u solves the weak formulation (2.9) for a particular test function v, it does not necessarily solve the original PDE (2.6) as well. However, if u solves (2.9) for all test functions v from a sufficiently large set, then it turns out that it also solves (2.6). What exactly this means will be explained in the next section.
In the strong formulation of the PDE, it is assumed that the partial derivatives of u exist up to order 2 (and therefore are continuous), additionally it also expected that the right-hand side function f is continuous. However, in the weak formulation, the PDE has been *relaxed* in the sense that only the first order partial derivatives must exist, and that the product of f and any v is integrable. For this reason, we call it the weak formulation. In the new formulation the assumptions on u and f are not as strong as in the original formulation. We actually do not even require the first order partial derivatives to be continuous, in fact in the weak formulation we only require that u and v are *weakly differentiable*. Again, what this exactly means will be explained in the next section.

The weak formulation of Poisson's problem can now be written in the following simple form:

$$a(u, v) = \ell(v) \tag{2.10}$$

where

$$a(u,v) = \int_\Omega \nabla u \nabla v \,\mathrm{d}S \quad \text{and} \quad \ell(v) = \int_\Omega v f \,\mathrm{d}S.$$

Here $a(\cdot,\cdot)$ is a bilinear form, which is important later and it is not difficult to show. It is trivial from the definition to see that it is symmetric, i.e. $a(u,v) = a(v,u)$, hence if it is linear in the first argument, it is linear in the second argument as well. Let α and β be scalars, then

$$\begin{aligned} a(\alpha u + \beta w, v) &= \int_\Omega \nabla(\alpha u + \beta w)\nabla v \,\mathrm{d}S \\ &= \int_\Omega (\nabla(\alpha u)\nabla v + \nabla(\beta w)\nabla v) \,\mathrm{d}S \\ &= \int_\Omega (\alpha \nabla u \nabla v + \beta \nabla w \nabla v) \,\mathrm{d}S \\ &= \alpha \int_\Omega \nabla u \nabla v + \beta \int_\Omega \nabla w \nabla v \,\mathrm{d}S \\ &= \alpha a(u,v) + \beta a(w,v), \end{aligned}$$

which was what we wanted to show, hence $a(\cdot,\cdot)$ is a bilinear form. We will later show why it is important that the weak formulation of Poisson's problem can be written on the form (2.10) where $a(\cdot,\cdot)$ is bilinear, but first we will go into a bit more detail on the earlier asked questions regarding the weak formulation.

2.3 Sobolev Spaces

In the previous section we derived the weak formulation of Poisson's problem by multiplying by a test function, v, and integrating, but for the integrals to make sense we have to put some restrictions on v, otherwise we are not guaranteed that the products are integrable.
In particular, it should be allowed that $v = u$ in (2.9), so their product is integrable if

$$\int_\Omega |\nabla u|^2 \,\mathrm{d}S < \infty$$

and similarly for the right-hand side in (2.9) if $v = f$. This means that we should require that $\nabla u, u$ and f are square-integrable. Consider the following space:

$$\mathcal{L}^p(\Omega) = \left\{ f : \Omega \to \mathbb{C} \;\middle|\; \int_\Omega |f|^p \,\mathrm{d}S < \infty \right\}.$$

To say that we require that the functions to be square-integrable, is equivalent to saying that the functions are contained in the vector space $\mathcal{L}^2(\Omega)$. The space $\mathcal{L}^2(\Omega)$ is a Hilbert space with respect to the inner product given by

$$\langle f, g \rangle_2 = \int_\Omega f\, \overline{g} \,\mathrm{d}S \tag{2.11}$$

and hence a Banach space with respect to the norm given by

$$\|f\|_2 = \langle f, f \rangle_2^{1/2} = \left(\int_\Omega |f|^2 \,\mathrm{d}S \right)^{1/2}. \tag{2.12}$$

However, it was not enough that the solution $u \in \mathcal{L}^2(\Omega)$, its partial derivatives should also be square integrable. Let us from this point on restrict us to the 2-dimensional case, i.e. $f = f(x,y)$. The following space now contains functions with the given requirements:

$$\mathcal{H}^1(\Omega) = \left\{ f \;\middle|\; f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \in \mathcal{L}^2(\Omega) \right\}. \tag{2.13}$$

Lastly, the solution u and test function v should satisfy the homogeneous boundary conditions, hence we introduce the following Sobolev space:

$$\mathcal{H}_0^1(\Omega) = \left\{ f \,\middle|\, f \in \mathcal{H}^1(\Omega), f(\partial\Omega) = 0 \right\},$$

that is

$$\mathcal{H}_0^1(\Omega) = \left\{ f : \Omega \to \mathbb{C} \,\middle|\, \int_\Omega |f|^2 \,\mathrm{d}S < \infty, \int_\Omega \left|\frac{\partial f}{\partial x}\right|^2 \mathrm{d}S < \infty, \int_\Omega \left|\frac{\partial f}{\partial y}\right|^2 \mathrm{d}S < \infty, f(\partial\Omega) = 0 \right\}. \quad (2.14)$$

A Sobolev space is simply a space that is equipped with a norm that is a combination of $\mathcal{L}^p$-norms of the function itself and its derivatives. The derivatives are here understood in a weak sense to make the space complete (Banach). As an example

$$f : \mathbb{R} \to \mathbb{R}$$
$$f(x) = |x|$$

is not differentiable in the classical sense, but it has a weak derivative which is given by

$$\nabla f(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if x} = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

as one would expect. However it is not a continuous function, as is required in the classical derivative. The reason why $\mathcal{H}_0^1(\Omega)$ would not be a complete vector space if we only considered strong derivatives is that one can construct sequences of (strongly) differentiable functions that converge to non-differentiable functions, e.g.

$$f_n(x) = \sqrt{\frac{1}{n^2} + x^2} \to \sqrt{x^2} = |x|$$

as n approaches infinity and yet f_n is differentiable in the strong sense, but its limit $|x|$ is not.
We will now equip this space with the following inner product:

$$\langle f, g \rangle_{\mathcal{H}_0^1} := \left\langle \frac{\partial f}{\partial x}, \frac{\partial g}{\partial x} \right\rangle_2 + \left\langle \frac{\partial f}{\partial y}, \frac{\partial g}{\partial y} \right\rangle_2. \quad (2.15)$$

It is not immediately obvious that the space $(\mathcal{H}_0^1(\Omega), \langle \cdot, \cdot \rangle_{\mathcal{H}_0^1})$ is a Hilbert space, but we will argue that it is in the following paragraphs.

Recall the definition of a Hilbert space. It is trivial to verify that $\langle \cdot, \cdot \rangle_{\mathcal{H}_0^1}$ indeed is an inner product. It then suffices to show that the norm $\| \cdot \|_{\mathcal{H}_0^1}$ satisfies the norm conditions, and the space is a Banach space with respect to that norm. Let us first observe that the norm induced by the inner product (2.15) is given by

$$\|f\|_{\mathcal{H}_0^1} = \langle f, f \rangle_{\mathcal{H}_0^1}^{1/2} = \left(\left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial x} \right\rangle_2 + \left\langle \frac{\partial f}{\partial y}, \frac{\partial f}{\partial y} \right\rangle_2 \right)^{1/2} = \left(\left\| \frac{\partial f}{\partial x} \right\|_2^2 + \left\| \frac{\partial f}{\partial y} \right\|_2^2 \right)^{1/2}. \quad (2.16)$$

Let us first consider the nonnegativity condition. Suppose $f \in \mathcal{H}_0^1(\Omega)$. Recall from the definition of the Sobolev space that both f and its partial derivatives also are contained in $\mathcal{L}^2(\Omega)$. Then

$$\|f\|_{\mathcal{H}_0^1} = \left(\left\| \frac{\partial f}{\partial x} \right\|_2^2 + \left\| \frac{\partial f}{\partial y} \right\|_2^2 \right)^{1/2} \geq 0$$

is trivially true since $\|\cdot\|_2$ is the norm of the space $\mathcal{L}^2$ and hence it satisfies the non-negativity condition for all $f \in \mathcal{L}^2(\Omega)$. There is equality when the partial derivatives are zero, that is, when f is constant. Since f is zero on the boundary and is constant, it is zero everywhere, hence

$$\|f\|_{\mathcal{H}_0^1} = 0 \iff f = 0.$$

Next, we wish to show that the norm satisfies the absolute scalability condition. Let $\alpha \in \mathbb{C}$, then

$$\|\alpha f\|_{\mathcal{H}_0^1} = \left(\left\| \frac{\partial(\alpha f)}{\partial x} \right\|_2^2 + \left\| \frac{\partial(\alpha f)}{\partial y} \right\|_2^2 \right)^{1/2} = \left(|\alpha|^2 \left(\left\| \frac{\partial f}{\partial x} \right\|_2^2 + \left\| \frac{\partial f}{\partial y} \right\|_2^2 \right) \right)^{1/2} = |\alpha| \|f\|_{\mathcal{H}_0^1}.$$

The fact that a norm induced by an inner product satisfies the triangle inequality can be proven in general using the Cauchy-Schwarz inequality in the following way:

$$\begin{aligned}
\|f+g\|^2 &= \langle f+g, f+g \rangle \\
&= \langle f, f \rangle + \langle g, g \rangle + \langle f, g \rangle + \langle g, f \rangle \\
&= \|f\|^2 + \|g\|^2 + \langle f, g \rangle + \langle g, f \rangle \\
&\leq \|f\|^2 + \|g\|^2 + 2|\langle f, g \rangle| \\
&\leq \|f\|^2 + \|g\|^2 + 2\|f\|\|g\| \quad \text{(Cauchy-Schwarz)} \\
&= (\|f\| + \|g\|)^2 .
\end{aligned}$$

Hence

$$\|f+g\| \leq \|f\| + \|g\|$$

for any norm induced by an inner product. We conclude that $\|\cdot\|_{\mathcal{H}_0^1}$ satisfies the triangle inequality and therefore satisfies all the conditions for being a norm on $\mathcal{H}_0^1$. The final requirement is that $\mathcal{H}_0^1(\Omega)$ is complete, that is to say that the weak derivatives should fill the holes of the sequences of (strongly) differentiable functions that converge to weakly differentiable functions. It can be shown that $\mathcal{H}_0^1(\Omega)$ is the *completion* of $\mathcal{C}_0^1(\Omega)$ under the $\mathcal{H}_0^1$-norm, where $\mathcal{C}_0^1(\Omega)$ is the space of continuous functions $f : \Omega \to \mathbb{R}$ that are zero on the boundary for which the first (strong) partial derivatives exist. This means that $(\mathcal{H}_0^1, \langle \cdot, \cdot \rangle_{\mathcal{H}_0^1})$ is a Banach space with respect to the induced norm, and it is therefore a Hilbert space.

When dealing with one spatial dimension, one can simply choose the inner product

$$\langle f, g \rangle_{\mathcal{H}_0^1} = \int_\Omega \nabla f \overline{\nabla g} \, \mathrm{d}x = \int_\Omega \frac{\mathrm{d}f}{\mathrm{d}x} \overline{\frac{\mathrm{d}g}{\mathrm{d}x}} \, \mathrm{d}x = \left\langle \frac{\mathrm{d}f}{\mathrm{d}x}, \frac{\mathrm{d}g}{\mathrm{d}x} \right\rangle_2,$$

and norm

$$\|f\|_{\mathcal{H}_0^1} = \langle f, f \rangle_{\mathcal{H}_0^1}^{1/2} = \left(\int_\Omega |\nabla f|^2 \right)^{1/2} \mathrm{d}x = \left(\int_\Omega \left| \frac{\mathrm{d}f}{\mathrm{d}x} \right|^2 \mathrm{d}x \right)^{1/2} = \left\| \frac{\mathrm{d}f}{\mathrm{d}x} \right\|_2 .$$

Showing that this the 1-dimensional case also forms a Hilbert space is simple using similar arguments as for the 2-dimensional case above.

With the introduction of the Sobolev space $(\mathcal{H}_0^1(\Omega), \|\cdot\|_{\mathcal{H}_0^1})$, the last important point is that if a function $u \in \mathcal{H}_0^1(\Omega)$ is a solution to the weak problem for all test functions $v \in \mathcal{H}_0^1(\Omega)$ then it is also a solution to the strong problem.

2.4 Galerkin Approximation

We have earlier reduced Poisson's problem to the weak form, which can be written as following the problem:
Find $u \in \mathcal{V}$ solving the problem

$$\forall v \in \mathcal{V} : \quad a(u, v) = \ell(v). \tag{2.17}$$

In the previous section we introduced the space $\mathcal{V} = \mathcal{H}_0^1(\Omega)$. There is an important theorem regarding problems of this form which is called the Lax-Milgram Theorem, assuming that $a(\cdot,\cdot)$ satisfies certain conditions. For problems of this form we consider all functionals ℓ of the so-called *dual space* of $\mathcal{V}$. The dual space of $\mathcal{V}$, denoted $\mathcal{V}'$, is the space of all bounded linear functionals $\ell : \mathcal{V} \to \mathcal{V}$. Let us now introduce the theorem:

Theorem 3 (Lax-Milgram)**.** *Let $(\mathcal{V}, \|\cdot\|)$ be a Hilbert space and $a(\cdot,\cdot)$ a bilinear form on $\mathcal{V}$ that for all $u, v \in \mathcal{V}$ satisfies*

(i) $|a(u,v)| \leq C\|u\|\|v\|$ *for some* $C > 0$. *(Bounded)*

(ii) $a(u,u) \geq \alpha\|u\|^2$ *for some* $\alpha > 0$. *(Coercive)*

Then, for any $\ell \in \mathcal{V}'$, there exists a unique solution $u \in \mathcal{V}$ such that

$$a(u,v) = \ell(v)$$

for all $v \in \mathcal{V}$.

A full proof of this theorem will not be given in this text as it requires introducing another theorem that is not necessary to understand the theory, other than for this proof. However, a small outline of the proof is given. The required theorem is known as the Riesz Representation Theorem. This theorem establishes an important connection between Hilbert spaces and their dual space. Every bounded linear functional (i.e. from the dual space) on a Hilbert space can be represented in terms of the inner product on the space with a *unique* representation. It can then be shown that $a(u,v)$ is an inner product on $\mathcal{V}$, using its coercitivity, and equipped with this inner product $\mathcal{V}$ is a Hilbert space. The boundedness of the inner product is required to establish that the space is complete (Banach). It follows then from Riesz representation theorem that there is a unique u such that $a(u,v) = \ell(v)$.
The Lax-Milgram theorem is an important theorem as it immediately guarantees unique solutions to a certain type of problems. Let us consider such a problem. Let $\mathcal{V}_h$ be a finite dimensional subspace of $\mathcal{V}$. The *Galerkin approximation problem* is then:
Given $\ell \in \mathcal{V}'$, find $u_h \in \mathcal{V}_h$ solving the problem:

$$\forall v_h \in \mathcal{V}_h : \quad a(u_h, v_h) = \ell(v_h). \tag{2.18}$$

The Galerkin approximation is a method (or truly a class of methods) where one reduces the dimensions of the problem in an attempt to discretize the problem. The Galerkin approximation is exactly the problem given in (2.17), with the exception of the space $\mathcal{V}$. In the Galerkin approximation we consider a finite dimensional subspace of $\mathcal{V}$. Even though we will not discuss an implementation of the method yet, the motivation behind the finite dimensional subspace is setting up a discrete problem which can be easily solved.
We will now show that there exists a unique Galerkin approximation to Poisson's problem in both 1 and 2 spatial dimensions, when u and v are real-valued. Recall from the weak formulation of Poisson that

$$a(u,v) = \int_\Omega \nabla u \nabla v \,\mathrm{d}S \qquad \text{and} \qquad \ell(v) = \int_\Omega v f \,\mathrm{d}S.$$

Consider first the 1-dimensional case, i.e. the gradients are simply the usual derivatives. Using the Cauchy-Schwarz inequality (2.4) we note that

$$|a(u,v)| = \left| \int_\Omega \nabla u \nabla v \,\mathrm{d}x \right| = |\langle u, v \rangle_{\mathcal{H}_0^1}| \leq \|u\|_{\mathcal{H}_0^1} \|v\|_{\mathcal{H}_0^1},$$

therefore $a(u,v)$ is bounded with $C = 1$. By direct computation we get that

$$a(u,u) = \int_\Omega \nabla u \nabla u \,\mathrm{d}x = \int_\Omega |\nabla u|^2 \,\mathrm{d}x = \|\nabla u\|_2^2 = \|u\|_{\mathcal{H}_0^1}^2,$$

and hence $a(u,v)$ is coercive with $\alpha = 1$, as well. Since $a(\cdot,\cdot)$ is a bounded, coercive bilinear form there exists by the Lax-Milgram theorem a unique solution $u_h \in \mathcal{V}_h$ to (2.18).

Let us now consider the 2-dimensional case, i.e. $u = u(x,y)$. We now have that

$$\begin{aligned}
|a(u,v)| &= \left| \int_\Omega \nabla u \nabla v \, \mathrm{d}S \right| \\
&= \left| \int_\Omega \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \end{pmatrix} \mathrm{d}S \right| \\
&= \left| \int_\Omega \left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} \right) \mathrm{d}S \right| \\
&= \left| \left\langle \frac{\partial u}{\partial x}, \frac{\partial v}{\partial x} \right\rangle_2 + \left\langle \frac{\partial u}{\partial y}, \frac{\partial v}{\partial y} \right\rangle_2 \right| \\
&= \left| \langle u, v \rangle_{\mathcal{H}_0^1} \right| \\
&\leq \|u\|_{\mathcal{H}_0^1} \|v\|_{\mathcal{H}_0^1},
\end{aligned}$$

using the Cauchy-Schwarz inequality (2.4) in the end. Hence $a(\cdot,\cdot)$ is bounded with $C = 1$. Similarly

$$\begin{aligned}
a(u,u) &= \left\langle \frac{\partial u}{\partial x}, \frac{\partial u}{\partial x} \right\rangle_2 + \left\langle \frac{\partial u}{\partial y}, \frac{\partial u}{\partial y} \right\rangle_2 \\
&= \left\| \frac{\partial u}{\partial x} \right\|_2^2 + \left\| \frac{\partial u}{\partial y} \right\|_2^2 \\
&= \|u\|_{\mathcal{H}_0^1}^2,
\end{aligned}$$

and hence $a(\cdot,\cdot)$ is coercive with $\alpha = 1$. Since $a(\cdot,\cdot)$ is a bounded, coercive bilinear form there exists by the Lax-Milgram theorem a unique solution $u_h \in \mathcal{V}_h$ to (2.18).

2.5 Computing the Galerkin Approximation

In any normed vector space $\mathcal{V}$, we call the sequence $(e_k)_{k\in\mathbb{N}}$ a basis for $\mathcal{V}$ if there for every element $f \in \mathcal{V}$ exist unique scalar coefficients $(c_k)_{k\in\mathbb{N}}$ such that

$$f = \sum_{k=1}^{\infty} c_k e_k. \tag{2.19}$$

The idea is that we wish to find the Galerkin approximation using a series representation like this. Since the Galerkin approximation exists in a finite-dimensional subspace $\mathcal{V}_h$ of $\mathcal{V}$ there exist unique scalar coefficients $(\hat{u}_k)_{k=1}^M$ such that

$$u = \sum_{k=1}^{M} \hat{u}_k e_k \tag{2.20}$$

for all $u \in \mathcal{V}_h$ given that $(e_k)_{k=1}^M$ is a basis for $\mathcal{V}_h$ and that M is the dimension of the space. Similarly, we can represent any test-function $v \in \mathcal{V}_h$ with the series

$$v = \sum_{k=1}^{M} \hat{v}_k e_k \tag{2.21}$$

for scalar coefficients $(\hat{v}_k)_{k=1}^M$. We can insert these representations into the Galerkin approximation problem (2.18), so that

$$a\left(\sum_{i=1}^{M}\hat{u}_i e_i, \sum_{j=1}^{M}\hat{v}_j e_j\right) = \ell\left(\sum_{i=1}^{M}\hat{v}_i e_i\right). \tag{2.22}$$

Using the bilinearity of $a(\cdot,\cdot)$ and the linearity of $\ell(\cdot)$ we can rewrite this as

$$\sum_{i=1}^{M}\sum_{j=1}^{M}\hat{u}_i\hat{v}_j\, a(e_i,e_j) = \sum_{i=1}^{M}\hat{v}_i\,\ell(e_i).$$

These expressions can be written using vectors and matrices, so that the equation can be written as

$$\hat{\mathbf{v}}^\intercal \mathbf{A}\hat{\mathbf{u}} = \hat{\mathbf{v}}^\intercal \boldsymbol{\ell} \tag{2.23}$$

where $\hat{\mathbf{v}} = (\hat{v}_k)_{k=1}^M$, $\hat{\mathbf{u}} = (\hat{u}_k)_{k=1}^M$, and $\boldsymbol{\ell} = (\ell(e_k))_{k=1}^M$ are column vectors and $\mathbf{A}$ is an $M \times M$ matrix whose (i,j)th element is given by

$$\mathrm{A}_{ij} = a(e_i, e_j).$$

Using the Euclidean inner product $\langle \mathbf{u}, \mathbf{v}\rangle_{\mathbb{R}^M} = \mathbf{u}^\intercal\mathbf{v}$, we can express (2.23) using inner products:

$$\langle \hat{\mathbf{v}}, \mathbf{A}\hat{\mathbf{u}}\rangle_{\mathbb{R}^M} = \langle \hat{\mathbf{v}}, \boldsymbol{\ell}\rangle_{\mathbb{R}^M}. \tag{2.24}$$

In any Hilbert space $\mathcal{H}$, if $\mathbf{u}, \mathbf{w} \in \mathcal{H}$, and

$$\langle \mathbf{v}, \mathbf{u}\rangle = \langle \mathbf{v}, \mathbf{w}\rangle$$

holds for all $\mathbf{v} \in \mathcal{H}$, then it follows that $\mathbf{u} = \mathbf{w}$. Using this fact, and that (2.24) holds for arbitrary $\hat{\mathbf{v}}$ it follows that

$$\mathbf{A}\hat{\mathbf{u}} = \boldsymbol{\ell}. \tag{2.25}$$

This means that the Galerkin approximation problem (2.18) is equivalent to the following problem: Find the vector $\hat{\mathbf{u}}$ solving (2.25) where $\ell \in \mathcal{V}'$ is given and $\mathbf{A}$ and $\boldsymbol{\ell}$ are defined element-wise as

$$\mathrm{A}_{ij} = a(e_i, e_j) \quad \text{and} \quad \ell_i = \ell(e_i).$$

This is simply a set of linear equations which can be solved using Gaussian elimination or similar techniques. The matrix $\mathbf{A}$ and the vector $\boldsymbol{\ell}$ are named the *stiffness matrix* and the *load vector*, respectively. Once the matrix-vector equation is solved, the Galerkin approximation can be easily computed using the series representation (2.20).

It is not, however, guaranteed that a matrix-vector equation of the form (2.25) has a unique solution, or has even a single solution. This is only guaranteed in the case where $\mathbf{A}$ is invertible, where the unique solution is simply given by

$$\hat{\mathbf{u}} = \mathbf{A}^{-1}\boldsymbol{\ell}.$$

In practice the inverse matrix $\mathbf{A}^{-1}$ is not actually computed since it is an expensive computation, but if we can prove that the matrix is invertible we are guaranteed a unique solution. A *positive definite* matrix is a matrix $\mathbf{M}$ for which it holds that

$$\mathbf{x}^\intercal\mathbf{M}\mathbf{x} \geq 0 \quad \text{and} \quad \mathbf{x}^\intercal\mathbf{M}\mathbf{x} = 0 \iff \mathbf{x} = \mathbf{0}$$

for all vectors $\mathbf{x}$. It is a property of positive definite matrices that they are invertible, hence if we can prove that $\mathbf{A}$ is positive definite it follows that it is invertible, and therefore there is a unique solution to (2.25). It is not given the stiffness matrix is positive definite for all differential equations, but it can be proven for Poisson's problem.

Proposition. *There exists a unique solution to (2.25) for Poisson's problem.*

Proof. Let $\mathcal{V}_h$ be a finite-dimensional subspace of $\mathcal{H}_0^1(\Omega)$ and $(e_k)_{k=1}^M$ a basis of $\mathcal{V}_h$. Let $\mathbf{x} = (x_k)_{k=1}^M$ be an arbitrary real vector, such that

$$x = \sum_{k=1}^{M} x_k e_k \in \mathcal{V}_h.$$

Recall that for Poisson's problem $a(u, v) = \langle u, v \rangle_{\mathcal{H}_0^1}$. Then

$$\begin{aligned}
\mathbf{x}^\intercal \mathbf{A} \mathbf{x} &= \sum_{i=1}^{M} \sum_{j=1}^{M} x_i x_j a(e_i, e_j) \\
&= a\left(\sum_{i=1}^{M} x_i e_i, \sum_{j=1}^{M} x_j e_j \right) \\
&= a(x, x) \\
&= \langle x, x \rangle_{\mathcal{H}_0^1} \\
&= \|x\|_{\mathcal{H}_0^1}^2 \geq 0.
\end{aligned}$$

Since $\|\cdot\|_{\mathcal{H}_0^1}$ is a norm, equality is reached if and only if $x = 0$. We conclude that $\mathbf{A}$ is positive definite, and hence invertible. Therefore there exists a unique solution given by $\hat{\mathbf{u}} = \mathbf{A}^{-1}\boldsymbol{\ell}$. □

2.6 Error Analysis

The hope is, of course, that the Galerkin approximation is a good approximation to the true solution, at least given we pick a subspace with high enough dimension. Let u_h be a solution to the finite dimensional problem (2.18) and let u be a solution to (2.17). Then we want $u_h \approx u$ for a sufficiently large dimension M, in a normed sense. Let us define ε as the size of the error of u_h, that is

$$\varepsilon = \|u - u_h\| \tag{2.26}$$

for some relevant norm. Then, more precisely, we want

$$\lim_{M \to \infty} \varepsilon = \lim_{M \to \infty} \|u - u_h\| = 0. \tag{2.27}$$

Recall from the definition of convergence in normed spaces, that this simply means that $u_h \to u$ as $M \to \infty$, which is exactly the goal of the approximation.

An important property that $a(\cdot, \cdot)$ has is the fundamental *Galerkin orthogonality* condition:

$$a(u - u_h, v_h) = 0, \quad \forall v_h \in \mathcal{V}_h. \tag{2.28}$$

Using the linearity of $a(\cdot, \cdot)$ it is shown by

$$a(u - u_h, v_h) = a(u, v_h) - a(u_h, v_h) = \ell(v_h) - \ell(v_h) = 0.$$

The fundamental orthogonality condition gives a bound on the error between the discrete finite element approximation and the true solution. This is called Céa's Theorem (or sometimes Céa's Lemma):

Theorem 4 (Céa)**.** *Let $(\mathcal{V}, \|\cdot\|)$ be a Hilbert space, and $\mathcal{V}_h$ be a finite dimensional subspace of $\mathcal{V}$. Let $a : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ be a bilinear form that is bounded with constant C and α-coercive. Furthermore, let $u \in \mathcal{V}$ be a solution to $a(u, v) = \ell(v)$ for all $v \in \mathcal{V}$, and $u_h \in \mathcal{V}_h$ a solution to $a(u_h, v_h) = \ell(v_h)$*

for all $v_h \in \mathcal{V}_h$. Then

$$\|u - u_h\| \leq \frac{C}{\alpha} \min_{v_h \in \mathcal{V}_h} \|u - v_h\| \quad \forall v_h \in \mathcal{V}_h \tag{2.29}$$

Hence u_h is the best possible finite dimensional solution in the given norm.

Proof. We first note that $v_h - u_h \in \mathcal{V}_h$, and recall the fundamental orthogonality property which states that $a(u - u_h, v_h) = 0 \quad \forall v_h \in \mathcal{V}_h$, hence

$$\begin{aligned}
\alpha\|u - u_h\|^2 &\leq a(u - u_h, u - u_h) && \text{(coercitivity)} \\
&= a(u - u_h, u - v_h + v_h - u_h) \\
&= a(u - u_h, u - v_h) + a(u - u_h, v_h - u_h) && \text{(linearity)} \\
&= a(u - u_h, u - v_h). && \text{(orthogonality)} \\
&\leq C\|u - u_h\|\|u - v_h\|. && \text{(boundedness)}
\end{aligned}$$

Next we divide through the inequality by $\alpha\|u - u_h\|$, so that

$$\begin{aligned}
\|u - u_h\| &\leq \frac{C}{\alpha}\|u - v_h\| \\
&\leq \frac{C}{\alpha} \inf_{v_h \in \mathcal{V}_h} \|u - v_h\| \\
&= \frac{C}{\alpha} \min_{v_h \in \mathcal{V}_h} \|u - v_h\|,
\end{aligned}$$

since $\mathcal{V}_h$ is closed. This completes the proof. □

In the case of Poisson's problem, we have shown that $\alpha = C = 1$ when $\mathcal{V} = (\mathcal{H}_0^1, \|\cdot\|_{\mathcal{H}_0^1})$. Using Céa's Theorem (2.29) we get the following nice result:

$$\|u - u_h\|_{\mathcal{H}_0^1} \leq \min_{v_h \in \mathcal{V}_h} \|u - v_h\|_{\mathcal{H}_0^1} \quad \forall v_h \in \mathcal{V}_h \tag{2.30}$$

where $\mathcal{V}_h$ is a finite dimensional subspace of $\mathcal{V}$. What this means is that there is no choice of a function $v_h \in \mathcal{V}_h$ with a smaller error ε, compared to the Galerkin approximation u_h, and hence the Galerkin aproximation is the best function inside the finite dimensional subspace to approximate the true function (in the sense of the given norm).

2.7 Time-dependent Problems

In the previous sections we dealt only with steady state problems – problems with solutions that do not change in time. However, a large portion of problems that we encounter depend on time: a cup of coffee cools as time passes; a wave's position depends on time, et cetera. These are just two of many, many examples time-dependent problems. Let us introduce one such time-dependent equation, namely the heat equation given by:

$$\frac{\partial u}{\partial t} - \nabla^2 u = f. \tag{2.31}$$

The solution of this differential equation is now, contrary to earlier equations that we have dealt with, a function of time t. If we have 2 spatial dimensions, then we are looking for a solution $u(x, y, t)$. This equation can, for example, describe how the heat of some domain changes as time passes; just the like the example of the cup of coffee. When dealing with differential equations of this type, one is often given an *initial condition* (IC), which in this example would describe the heat of the domain for some

initial time, say $t_0 = 0$. The differential equation, a boundary condition, and the initial value form together an *initial value problem* (IVP):

$$\begin{aligned} \tfrac{\partial u}{\partial t} - \nabla^2 u &= f \quad \text{in} \quad \Omega \times (0,T) \\ u &= 0 \quad \text{on} \quad \partial\Omega \times (0,T) \\ u &= u_0 \quad \text{in} \quad \Omega \times \{0\} \end{aligned} \tag{2.32}$$

The Weak Formulation

For steady state problems, we proved that we were guaranteed unique weak solutions to a class of problems, e.g. for the Poisson BVP. To get to that result, we set up the weak formulation of the problem, and we will do exactly that again for the heat equation (2.32). The first thing one might notice is that the heat equation is very similar to the Poisson equation. To be more precise, if the solution is constant in time, that is its time-derivative is zero, the heat equation reduces to exactly the Poisson equation. For this reason, the weak formulation will look quite similar to that of Poisson.
To derive the weak formulation, we multiply by a test-function v that is zero on the boundary, and integrate over the domain.

$$\int_\Omega v \frac{\partial u}{\partial t} \,\mathrm{d}S - \int_\Omega v \nabla^2 u \,\mathrm{d}S = \int_\Omega v f \,\mathrm{d}S \tag{2.33}$$

Since $v\nabla^2 u = \nabla(v\nabla u) - \nabla v \nabla u$, we have using the divergence theorem that

$$\begin{aligned} \int_\Omega v \nabla^2 u \,\mathrm{d}S &= \int_\Omega \nabla(v\nabla u) \,\mathrm{d}S - \int_\Omega \nabla v \nabla u \,\mathrm{d}S \\ &= \int_{\partial\Omega} v \nabla u \cdot \mathbf{n} \,\mathrm{d}s - \int_\Omega \nabla v \nabla u \,\mathrm{d}S \\ &= -\int_\Omega \nabla v \nabla u \,\mathrm{d}S, \end{aligned}$$

where the first integral vanishes because of the boundary condition on v. The equation (2.33) simplifies to

$$\int_\Omega v \frac{\partial u}{\partial t} \,\mathrm{d}S + \int_\Omega \nabla v \nabla u \,\mathrm{d}S = \int_\Omega v f \,\mathrm{d}S \tag{2.34}$$

which is the weak formulation of (2.32).

This can be written on the form

$$\left\langle \frac{\partial u}{\partial t}, v \right\rangle_2 + a(u,v) = \langle f, v \rangle_2 \tag{2.35}$$

using the bilinear form $a(\cdot,\cdot)$ from the Poisson weak formulation, that is also implicitly dependent on time in this case. Hence, a function $u : \Omega \times (0,T) \to \mathcal{H}_0^1(\Omega \times (0,T))$ is a weak solution to (2.32) if for every $v \in \mathcal{H}_0^1(\Omega \times (0,T))$ the equality (2.35) holds and $u(x,y,0) = u_0(x,y)$.

Galerkin Approximation

We know from earlier (specifically when computing the Galerkin approximation of Poisson) that the following term can be approximated in a finite-dimensional subspace with the expression

$$\int_\Omega \nabla v \nabla u \,\mathrm{d}S \approx \langle \hat{\mathbf{v}}, \mathbf{A}\hat{\mathbf{u}} \rangle_{\mathbb{R}^M}$$

when $(e_i)_{i=1}^M$ forms a basis of $\mathcal{V}_h$ and we use the series expansions

$$u \approx \sum_{i=1}^M \hat{u}_i e_i \quad \text{and} \quad v \approx \sum_{i=1}^M \hat{v}_i e_i.$$

Similarly, we get that

$$\int_\Omega vf \,\mathrm{d}S \approx \langle \hat{\mathbf{v}}, \boldsymbol{\ell} \rangle_{\mathbb{R}^M},$$

where

$$\mathrm{A}_{ij} = \int_\Omega \nabla e_i \nabla e_j \,\mathrm{d}S \quad \text{and} \quad \ell_i = \int_\Omega e_i f \,\mathrm{d}S.$$

One thing to note is that the coefficients $\hat{u}$ in the previous computations of course must depend on time, unlike for steady state problems. We now approximate the final term

$$\begin{aligned}
\int_\Omega v \frac{\partial u}{\partial t} \,\mathrm{d}S &\approx \int_\Omega \left(\sum_{j=1}^M \hat{v}_j e_j \right) \left(\sum_{i=1}^M \frac{\partial}{\partial t} \hat{u}_i e_i \,\mathrm{d}S \right) \\
&= \sum_{i=1}^M \sum_{j=1}^M \hat{v}_j \frac{\partial}{\partial t} \hat{u}_i \int_\Omega e_i e_j \,\mathrm{d}S \\
&= \hat{\mathbf{v}}^\intercal \mathbf{B} \frac{\partial}{\partial t} \hat{\mathbf{u}} \\
&= \langle \hat{\mathbf{v}}, \mathbf{B} \frac{\partial}{\partial t} \hat{\mathbf{u}} \rangle_{\mathbb{R}^M}
\end{aligned}$$

where the matrix $\mathbf{B}$ is defined element-wise by

$$\mathrm{B}_{ij} = \int_\Omega e_i e_j \,\mathrm{d}S. \tag{2.36}$$

This means that our spatial discretization of the heat equation is

$$\langle \hat{\mathbf{v}}, \mathbf{B} \frac{\partial}{\partial t} \hat{\mathbf{u}} \rangle_{\mathbb{R}^M} + \langle \hat{\mathbf{v}}, \mathbf{A} \hat{\mathbf{u}} \rangle_{\mathbb{R}^M} = \langle \hat{\mathbf{v}}, \boldsymbol{\ell} \rangle_{\mathbb{R}^M}.$$

Since this inner product is real, it is linear in the second argument, so

$$\langle \hat{\mathbf{v}}, \mathbf{B} \frac{\partial}{\partial t} \hat{\mathbf{u}} + \mathbf{A} \hat{\mathbf{u}} \rangle_{\mathbb{R}^M} = \langle \hat{\mathbf{v}}, \boldsymbol{\ell} \rangle_{\mathbb{R}^M},$$

and since this holds for arbitrary $\hat{\mathbf{v}}$, we finally get that

$$\mathbf{B} \frac{\partial}{\partial t} \hat{\mathbf{u}} + \mathbf{A} \hat{\mathbf{u}} = \boldsymbol{\ell} \tag{2.37}$$

where

$$\begin{aligned}
\mathrm{B}_{ij} &= \int_\Omega e_i e_j \,\mathrm{d}S \\
\mathrm{A}_{ij} &= \int_\Omega \nabla e_i \nabla e_j \,\mathrm{d}S \\
\ell_i &= \int_\Omega e_i f \,\mathrm{d}S.
\end{aligned}$$

To sum this up, finding an approximation of the weak solution to (2.35) is equivalent to solving the linear system (2.37). The existence and uniqueness of the approximation can be proven, and a proof of this can be found in [4]. The theory required to understand the heat problem is quite similar to the theory of the Poisson problem, so it needs no explanations that has not been discussed in the theory of the Poisson problem. However, as it will be seen later, in the implementation there is a lot to be said for time-dependent problems

CHAPTER 3

Implementation

Up to now we have only dealt with the Finite Element Method on a theoretical basis, we have yet to delve into an actual implementation of the method. In the following chapter it will be shown how one constructs a numerical (2-dimensional) Finite Element solver, step by step.

3.1 Finite Element Mesh

When using the Finite Element Method, we seek to discretize our spatial domain into so-called elements. There is a choice of the shape of the elements to be made (usually triangular or rectangular), but the most popular are triangular elements, and this also what will be used here. In the 2-dimensional case the domain of interest can be almost any (connected) set. However, we will restrict us to a simple rectangular domain. To discretize the domain into elements simply means divide the domain into smaller pieces (elements). When using triangular elements, each element consist of 3 *nodes* (or *vertices*), which are connected by 3 *edges* (edges and vertices can be shared between multiple elements). The subdomain enclosed by the edges is what we call an element. In figure (3.1) an example of a rectangular finite element mesh is seen. We enumerate the elements and vertices column-wise top-down,

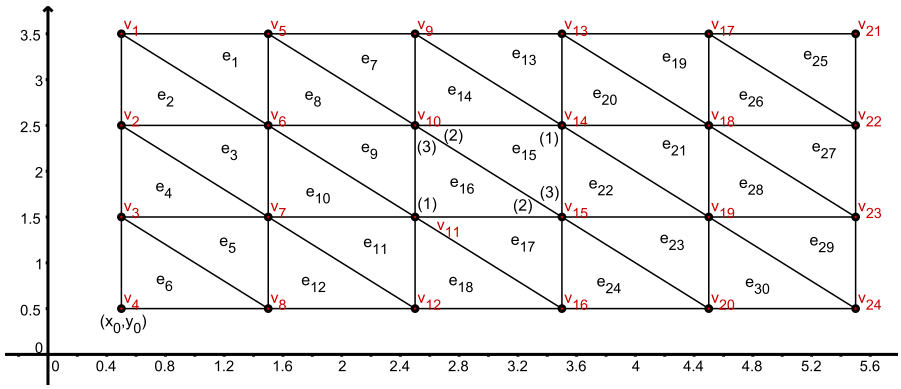


Figure 3.1: *A rectangular finite element mesh.*

as seen in the figure. To construct this mesh, we require the position in the plane (x_0, y_0), which will be the lower corner vertex in this implementation, the width `L1` and height `L2` of the rectangle should be given, as well as the number of elements the domain should be divided into horizontally `NoElms1` and vertically `NoElms2`. In the example in figure (3.1) the chosen parameters are `x0 = 0.5`, `y0 = 0.5`, `NoElms1 = 5`, `NoElms2 = 3`, `L1 = 5`, and `L2 = 3`. To keep track of the mesh, we construct the following data structures: `VX` and `VY` are vectors that in the ith entry, respectively, contain the x-position and y-position of the ith vertex. Algorithm (1) constructs this specific mesh, but one can of course make algorithms to construct other types of meshes.
Next we introduce an Element-to-Vertex table `EToV` which is an $N \times 3$ matrix, where N is the number of elements. In row n of `EToV` are the 3 vertex numbers corresponding to element e_n (listed in counter-clockwise order, starting with the vertex on the opposite side of the longest edge). An example of this table can be seen in table (3.2). The construction of the Element-to-Vertex table is done by algorithm (2).

Algorithm 1 Construction of `VX` and `VY`

function $XY(x_0, y_0, L_1, L_2,$ `NoElms1`, `NoElms2`$)$
  ElementWidth $:= L_1$ / `NoElms1`
  ElementHeight $:= L_2$ / `NoElms2`
  Row := `NoElms2`
  Col $:= 0$
  for $v_n \in V$ **do**
    `VX`$(n) := x_0 +$ Row $\cdot$ ElementWidth
    `VY`$(n) := y_0 +$ Col $\cdot$ ElementHeight
    Row := Row $- 1$
    if Row $= -1$ **then**
      Row := `NoElms2`
      Col := Col $+ 1$
    end if
  end for
  return `VX`, `VY`
end function

Algorithm 2 Construction of `EToV`

function *ConstructEToV*(`NoElms1`, `NoElms2`)
  for $j = 1$ to `NoElms1` **do**
    for $i = 1$ to `NoElms2` **do**
      $A := (1 +$ `NoElms2`$) \cdot j + i$
      $B := (1 +$ `NoElms2`$) \cdot (j - 1) + i$
      `EToV`$(2 \cdot i + 2 \cdot (j-1) \cdot$ `NoElms2` $- 1, 1..3) := (A, B, A - 1)$
      `EToV`$(2 \cdot i + 2 \cdot (j-1) \cdot$ `NoElms2`$, 1..3) := (B + 1, A + 1, B)$
    end for
  end for
  return `EToV`
end function

n	`EToV`		
1	5	1	6
2	2	6	1
3	6	2	7
4	3	7	2
⋮	⋮	⋮	⋮
30	20	24	19

Table 3.2: *Element-to-Vertex table of mesh given in figure (3.1).*

3.2 Assembling the Stiffness Matrix

To begin the implementation we will look a the general statement of the 2-dimensional Poisson problem:

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in} \quad \Omega \subset \mathbb{R}^2 \\ u &= g \quad \text{on} \quad \partial\Omega \end{aligned}$$

The goal of using the finite element method on a boundary value problem like this, is to find an approximate solution $\hat{u}(x, y) \approx u(x, y)$. In the earlier sections, we revealed that one can approximate functions using a linear combination of basis functions. To be exact, the Galerkin approximation is given by

$$\hat{u}(x, y) = \sum_{i=1}^{M} \hat{u}_i \phi_i(x, y), \tag{3.1}$$

where M is the number of nodes (vertices), and $\hat{u}_i$ are the unknown scalar coefficients. The scalar coefficients are approximations of the functions values, i.e. $\hat{u}_i \approx u(x_i, y_i)$. The set of functions $\{\phi_1, \phi_2, ..., \phi_M\}$ is the set of finite element basis functions, i.e. the finite dimensional vector space $\mathcal{V}_h$ is spanned by these basis functions. We have also shown that finding these scalar coefficients boils down to solving the following set of linear equations

$$\mathbf{A}\hat{\mathbf{u}} = \boldsymbol{\ell} \tag{3.2}$$

where the stiffness matrix and load vector are defined (for Poisson) element-wise respectively by

$$\mathrm{A}_{ij} = a(\phi_i, \phi_j) = \int_\Omega \nabla\phi_i \nabla\phi_j \, \mathrm{d}S$$

and

$$\ell_i = \ell(\phi_i) = \int_\Omega f\phi_i \, \mathrm{d}S$$

Finite Element Basis Functions

Before we can compute these integrals, we have to choose a set of basis functions. Perhaps the simplest basis functions are piecewise affine (first-order polynomials) basis functions. This means that within a particular element e_n, it can be expressed on the form

$$\phi_i^{(n)}(x, y) = a_i + b_i x + c_i y \quad (x, y) \in e_n.$$

Additionally, the basis functions are defined such that for any two (different, i.e. $i \neq j$) vertices with positions (x_i, y_i) and (x_j, y_j), we have that $\phi_i(x_i, y_i) = 1$ and $\phi_i(x_j, y_j) = 0$. To derive the explicit expressions for the basis functions, we will use the following observation: on any given element e_n, only three different local basis function can be nonzero which we will name $\phi_1^{(n)}(x, y), \phi_2^{(n)}(x, y)$, and $\phi_3^{(n)}(x, y)$, using a local numbering. If we consider the first of these, $\phi_1^{(n)}(x, y)$, and three different vertices with (locally numbered) positions $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3), then using the relations noted above we get

$$\begin{aligned} a_1 + b_1 x_1 + c_1 y_1 &= 1 \\ a_1 + b_1 x_2 + c_1 y_2 &= 0 \\ a_1 + b_1 x_3 + c_1 y_3 &= 0 \end{aligned}$$

These three linear equations can be solved for the three coefficients with the unique solution given by

$$\begin{aligned} a_1 &= \frac{x_2 y_3 - x_3 y_2}{2\Delta} \\ b_1 &= \frac{y_2 - y_3}{2\Delta} \\ c_1 &= \frac{x_3 - x_2}{2\Delta} \end{aligned}$$

where $|\Delta|$ is the area of the element in the mesh given by

$$|\Delta| = \frac{1}{2}\left|x_2y_3 - y_2x_3 - x_1y_3 + y_1x_3 + x_1y_2 - y_1x_2\right|.$$

The formulas for the two other local basis functions can be derived similarly, and we get the general local expressions

$$\begin{aligned}
\phi_1^{(n)}(x,y) &= \frac{1}{2|\Delta|}\left(A_1 + B_1x + C_1y\right)\\
\phi_2^{(n)}(x,y) &= \frac{1}{2|\Delta|}\left(A_2 + B_2x + C_2y\right)\\
\phi_3^{(n)}(x,y) &= \frac{1}{2|\Delta|}\left(A_3 + B_3x + C_3y\right)
\end{aligned}$$

where the coefficients are given by

$$\begin{aligned}
A_i &= x_jy_k - x_ky_j\\
B_i &= y_j - y_k\\
C_i &= x_k - x_j
\end{aligned}$$

and $(i,j,k) = (1,2,3)$, $(2,3,1)$, $(3,1,2)$.

Computing the Integrals

Having now derived the formulas for the local basis functions, we are now one step closer to evaluating the integral

$$\mathrm{A}_{ij} = \int_\Omega \nabla\phi_i \nabla\phi_j \,\mathrm{d}S. \tag{3.3}$$

To evaluate the integral over the entire domain, is the same as summing the integrals over each element. If $i \neq j$ the integrals over all but two elements will be zero, since at least one of the basis functions will be zero. Let us now compute the integrals of one such element e_n. That is

$$\begin{aligned}
&\int_{e_n} \nabla\phi_i^{(n)} \nabla\phi_j^{(n)} \,\mathrm{d}S\\
&= \int_{e_n} \left[\frac{\mathrm{d}\phi_i^{(n)}}{\mathrm{d}x}\frac{\mathrm{d}\phi_j^{(n)}}{\mathrm{d}x} + \frac{\mathrm{d}\phi_i^{(n)}}{\mathrm{d}y}\frac{\mathrm{d}\phi_j^{(n)}}{\mathrm{d}y}\right] \mathrm{d}S\\
&= \int_{e_n} \left[\frac{B_i}{2|\Delta|}\frac{B_j}{2|\Delta|} + \frac{C_i}{2|\Delta|}\frac{C_j}{2|\Delta|}\right] \mathrm{d}S\\
&= \left[\frac{B_iB_j + C_iC_j}{4|\Delta|^2}\right] \int_{e_n} 1 \,\mathrm{d}S\\
&= \frac{B_iB_j + C_iC_j}{4|\Delta|^2} \cdot |\Delta|\\
&= \frac{1}{4|\Delta|}(B_iB_j + C_iC_j),
\end{aligned}$$

for some local ordering $i,j \in \{1,2,3\}$. Let us name this quantity

$$k_{i,j}^{(n)} = \frac{1}{4|\Delta|}(B_iB_j + C_iC_j), \tag{3.4}$$

then the integral over the entire domain is either the integral over all the elements the ith vertex belongs to (if $i = j$), which is six quantities if the vertex is not on the boundary

$$\mathrm{A}_{ii} = k_{i,i}^{(n_1)} + k_{i,i}^{(n_2)} + k_{i,i}^{(n_3)} + k_{i,i}^{(n_4)} + k_{i,i}^{(n_5)} + k_{i,i}^{(n_6)}$$

or two elements if the basis functions are neighbours

$$\mathrm{A}_{ij} = k_{i,j}^{(n_1)} + k_{j,i}^{(n_2)} \tag{3.5}$$

for the local ordering $i, j \in \{1, 2, 3\}$. In this notation the elements e_{n_1}, e_{n_2} or $e_{n_1}, e_{n_2}, ..., e_{n_6}$ simply means the elements where the basis functions intersect. The last case is if the basis functions do not intersect at all, which is actually the most common case since this occurs every time the two basis functions are not neighbours or equal. In this case the integral will be zero, and because of this fact, the stiffness matrix $\mathbf{A}$ turns out to be very sparse (has proportionally few nonzero entries).
The right hand side integral given by

$$\ell_i = \int_\Omega f\phi_i \,\mathrm{d}S \tag{3.6}$$

is an integral we will have to numerically estimate, since the function f is arbitrary, so we cannot evaluate the integral in closed form. This integral will be summed up by integrals over each element as well. Let

$$\begin{aligned} l_i^{(n)} &= \int_{e_n} f\phi_i^{(n)} \,\mathrm{d}S \\ &= \int_{e_n} f\frac{1}{2|\Delta|}(A_i + B_i x + C_i y) \,\mathrm{d}S \\ &= \frac{1}{2|\Delta|}\int_{e_n} f(A_i + B_i x + C_i y) \,\mathrm{d}S \\ &\approx \frac{|\Delta|}{3}\tilde{f}, \end{aligned}$$

where $\tilde{f}$ is average value of f evaluated at the three vertices, i.e. with local ordering

$$\tilde{f} = \frac{1}{3}(f(x_1, y_1) + f(x_2, y_2) + f(x_3, y_3)).$$

One can also, instead of evaluating f at each vertex and average that, evaluate f at the average of the vertices, i.e. the center point, which is

$$\tilde{f} = f\left(\frac{x_1 + x_2 + x_3}{3}, \frac{y_1 + y_2 + y_3}{3}\right).$$

Then we can estimate the final integral again by summing the appropriate values of $l_i^{(n)}$, depending on how many elements the ith vertex belongs to. The implementation of assembling the stiffness matrix and load vector can be seen in algorithm (3).

3.3 Boundary Conditions

So far we have constructed a set of linear equations that account for the differential equation inside the domain, but we have yet to impose the boundary conditions. Specifically, in this case we need to impose the Dirichlet boundary condition $u = g$ on $\partial\Omega$. The first problem to consider is how one figures out which vertices are boundary vertices. In a rectangular domain this is not difficult at all, since it is just all vertices that have minimum or maximum x and y-coordinates, i.e. vertices that are of the form $(x_{\min}, y), (x_{\max}, y), (x, y_{\min})$, or $(x, y_{\max})$ are boundary vertices. Once we know all the

Algorithm 3 Stiffness Matrix and Load Vector Assembly

```
1: function ASSEMBLE(EToV, VX, VY, f)
2:     Allocate space A_{N×N} and ℓ_{N×1} with zeros.
3:     for n = 1 to N do
4:         Extract the global numbering (i, j) and position (x, y) for vertices in e_n.
5:         for r = 1 to 3 do
6:             Compute l_r^(n).
7:             ℓ(i) := ℓ(i) + l_r^(n)
8:             for c = 1 to 3 do
9:                 Compute k_{r,c}^(n).
10:                A(i, j) := A(i, j) + k_{r,c}^(n)
11:            end for
12:        end for
13:    end for
14:    return A, ℓ
15: end function
```

vertices $(x_i, y_i) \in \partial\Omega$ we require that $\hat{u}(x_i, y_i) = f(x_i, y_i)$, which is simply imposed by setting the series coefficients equal to the function values

$$\hat{u}_i := f(x_i, y_i), \quad (x_i, y_i) \in \partial\Omega. \tag{3.7}$$

Once we have imposed the boundary condition on a particular boundary vertex, the linear system changes as a result, and we have in some ways eliminated an unknown coefficient. In general a row i in the linear system has form

$$\sum_{j=1}^{M} \mathrm{A}_{ij}\hat{u}_j = \ell_i,$$

for example in a simple case with only 4 unknowns, and let us say or the sake of demonstration that the second vertex is on the boundary, then the equation from the second row is

$$\mathrm{A}_{2,1}\hat{u}_1 + \mathrm{A}_{2,2}\hat{u}_2 + \mathrm{A}_{2,3}\hat{u}_3 + \mathrm{A}_{2,4}\hat{u}_4 = \ell_2,$$

however, we know the value of $\hat{u}_2$ since it is on the boundary, say $\hat{u}_2 = f_2$, so the equation simplifies greatly to

$$0 \cdot \hat{u}_1 + 1 \cdot \hat{u}_2 + 0 \cdot \hat{u}_3 + 0 \cdot \hat{u}_4 = f_2.$$

To eliminate the variable $\hat{u}_2$ from all the other equations (other than second row), we simply need to subtract $\mathrm{A}_{i,2}f_2$ on both sides of the equation. The linear system of this simple example will now look like the following:

$$\begin{bmatrix} \mathrm{A}_{1,1} & 0 & \mathrm{A}_{1,3} & \mathrm{A}_{1,4} \\ 0 & 1 & 0 & 0 \\ \mathrm{A}_{3,1} & 0 & \mathrm{A}_{3,3} & \mathrm{A}_{3,4} \\ \mathrm{A}_{4,1} & 0 & \mathrm{A}_{4,3} & \mathrm{A}_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} = \begin{bmatrix} \ell_1 - \mathrm{A}_{1,2}f_2 \\ f_2 \\ \ell_3 - \mathrm{A}_{3,2}f_2 \\ \ell_4 - \mathrm{A}_{4,2}f_2 \end{bmatrix}. \tag{3.8}$$

This modification has to be done for each boundary vertex, and its implementation can be seen in algorithm (4).

We are now ready to construct a finite element approximation for a Dirichlet boundary value problem. First choose the appropriate parameters x_0, y_0, L_1, L_2 and the desired number of elements `NoElms1` and `NoElms2`. Then construct the data structures `VX`, `VY`, and `EToV` with algorithms (1) and (2). Now assemble the stiffness matrix and load vector with algorithm (3), and afterwards impose the Dirichlet boundary condition with algorithm (4). Lastly, solve the linear system $\mathbf{A}\hat{\mathbf{u}} = \boldsymbol{\ell}$ (in `MATLAB` this can be done by `u = A\l`).

Algorithm 4 Imposing Dirichlet boundary conditions in linear system

```
1: function DirichletBC(A, ℓ, f)
2:     Find all i such (x_i, y_i) ∈ ∂Ω.
3:     for all (x_i, y_i) ∈ ∂Ω do
4:         A(i, i) := 1
5:         ℓ(i) := f(x_i, y_i)
6:         for j = 1 to M (j ≠ i) do
7:             A(i, j) := 0
8:             if (x_j, y_j) ∉ ∂Ω then
9:                 ℓ(j) := ℓ(j) − A(j, i) · f_i
10:                A(j, i) := 0
11:            end if
12:        end for
13:    end for
14:    return A, ℓ
15: end function
```

3.4 Adaptive Mesh Refinement

In the preceding section we showed how to find the finite element approximation to a boundary value problem, given a choice of initial parameters. The approximation can be made more precise if one increases the number of degrees of freedom. The number of degrees of freedom is a measure of the size of the given problem, which in our case is the number of elements. If we increase the number of degrees of freedom (elements), we will in general get a more precise approximation, however this surely comes at a cost. The more elements we divide our domain into, the larger the system of linear equations becomes which increases computation time and memory requirements.
When solving a BVP with any kind of good precision, we often require a very fine mesh (i.e. many elements) in certain areas of the domain, and less fine in other areas. This occurs particularly when the solution is changing rapidly in areas (high absolute values of the first and second derivatives), then in those areas a high number of elements is required. The problem is, though, that we in general do not know where the solution is changing rapidly, since we do not know the solution. In this section we will describe how one can implement a method that automatically constructs a suitable mesh for the boundary value problem. This is called adaptive mesh refinement.

The Method

The adaptive mesh refinement algorithm boils down to the following two things:

1. A local mesh refinement algorithm
2. An error estimator

The mesh refinement algorithm should be able to refine the chosen areas of the mesh. In this implementation this will be done by dividing the relevant elements into two new elements, thereby refining the mesh in that particular area.
The error estimator, should, as the name implies, estimate the error of the given approximation. This lets us know where to refine the mesh, i.e. in the places with an error over some set error-tolerance. We estimate the error by comparing the solutions on two different meshes: a coarse mesh and a refined mesh. If the solution is different enough (difference over some tolerance) in some area in the two meshes, then this means that the coarse mesh is not fine enough in that area, and then we know where to do a refinement. The adaptive mesh refinement algorithm is algorithm (5).
Of course, we cannot implement this algorithm before we implement the mesh refinement algorithm and the error estimator, which is what we will do in the following sections.

Algorithm 5 Solve Dirichlet boundary value problem adaptively

```
1: function AdaptiveSolve
2:     Create an initial coarse mesh, and solve the BVP.
3:     Mark all elements for refinement.
4:     while there are marked elements do
5:         Refine all marked elements.
6:         Solve the BVP on the refined mesh.
7:         Estimate error between the two previous solutions.
8:         Mark elements with error over a set error-tolerance.
9:     end while
10: end function
```

Element Bisection

Firsly, we need an algorithm to refine marked elements. We will now present an element bisection algorithm, which is an algorithm that divides marked elements into two smaller elements. In a given element we define the *base* to be the longest edge, and the *peak* is the vertex on the opposite side of the edge. The idea is then, if the given element is marked, then we create an edge going from the peak to the middle of the base, thereby bisecting the element into two elements or equal size.
This approach is possibly the simplest one, however there is one problem that one has to look out for. We want an element to always be enclosed by exactly three edges, since we are using triangular elements. The problem is, when bisecting an element, the base edge is divided into two smaller edges, and this can cause problems. The counter to this problem is, if this problem were to occur we also refine the other element that shares this base. This requires us to refine elements that are not marked for refinement, but it is necessary if we wish to keep this structure.

Algorithm 6 Bisect an element into two smaller elements

```
1: function ElementBisection
2:     Let e_k be an element marked for refinement.
3:     if the base of e_k is a boundary edge then
4:         Divide e_k into two elements
5:     else
6:         Let e_j be the element that shares the edge that is base in e_k
7:         if the base of e_j is also the base of e_k then
8:             Divide both e_k and e_j into two elements each
9:         else
10:            Recursively call this function on the element e_j
11:            Recursively call this function on the element e_k
12:        end if
13:    end if
14: end function
```

It can be seen that there are three possibilities. The base edge in element e_k is a boundary edge, which causes no problems to the conformity if we bisect from base to peak. The next case is if two neighbouring elements (across the base), both have that particular edge as their base. To keep the conformity we just bisect both elements. The last case is when the base of an element, is not the base of the neighbouring element across that edge. This can cause problems to the conformity, so therefore we have to refine those elements first.

Error Estimation

The next step is to estimate the error of the solution, by comparing the initial solution and the refined one. Error estimation is a whole subject itself, as estimating errors is quite expensive, so there are many proposed methods depending on the mesh and basis functions. The idea is that we wish to enforce

$$\|u - u_h\| \leq \varepsilon_{\text{tol}}$$

where u is the exact solution, and u_h is the approximated solution, for some relevant norm. Of course, the problem is that we do not know the exact solution u, hence we have to estimate. We also have to make a choice of norm, and different norms have been discussed in various contexts in this text. In practice, a lot of different norms can be used, some more computationally expensive than others. One way to approximate the error is to compute the volume below the solution and enclosed by the element, i.e.

$$\text{vol}(u_h, e_n) = \int_{e_n} |u_h|$$

and then compute the volume below the refined solution u_h^*, which is equal to the sum of volumes of each sub-element (the elements the coarse element was refined into). Let e_n be refined into $e_{n_1}, ..., e_{n_m}$, then

$$\text{vol}(u_h^*, e_n) = \sum_{k=1}^{m} \text{vol}(u_h^*, e_{n_k}) = \sum_{k=1}^{m} \int_{e_{n_k}} |u_h^*|.$$

We then want the absolute difference between these volumes, the estimated error, to be below a set tolerance, i.e.

$$|\text{vol}(u_h, e_n) - \text{vol}(u_h^*, e_n)| \leq \varepsilon_{\text{tol}}. \tag{3.9}$$

The reason this will be used in this implementation is that one can derive a closed form solution of the volume, when using triangular elements and piecewise affine functions. Let $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3) be vertices of an element whose volume we wish to compute, and then let $z_i = u_h(x_i, y_i)$. It can then be shown that

$$\text{vol}(u_h, e_n) = \frac{1}{3}(|z_1| + |z_2| + |z_3|)|\Delta|. \tag{3.10}$$

The reason this error estimate works is that it behaves like the $\mathcal{L}^1$-norm, since we have

$$\|u_h\|_{\mathcal{L}^1(e_n)} = \int_{e_n} |u_h|,$$

hence

$$\begin{aligned} \|u_h - u_h^*\|_{\mathcal{L}^1(e_n)} &\geq \left| \|u_h\|_{\mathcal{L}^1(e_n)} - \|u_h^*\|_{\mathcal{L}^1(e_n)} \right| \\ &= \left| \int_{e_n} |u_h| - \int_{e_n} |u_h^*| \right| \\ &= \left| \text{vol}(u_h, e_n) - \text{vol}(u_h^*, e_n) \right|, \end{aligned}$$

using the reverse triangle inequality. Therefore the real $\mathcal{L}^1$-norm is larger than the error estimate, but it can only be slightly larger, since u_h and u_h^* are quite similar. On the other hand we always have the relationship

$$\|x\|_2 \leq \|x\|_1,$$

so the error estimate will still in practice generally be larger than the $\mathcal{L}^2$-norm of the error, i.e.

$$\|u_h - u_h^*\|_{\mathcal{L}^2(e_n)} \leq \|u_h - u_h^*\|_{\mathcal{L}^1(e_n)} = |\text{vol}(u_h, e_n) - \text{vol}(u_h^*, e_n)| + \delta, \tag{3.11}$$

for some small $\delta \geq 0$. The reason this approach is used, and not an actual estimate of the $\mathcal{L}^1$ or $\mathcal{L}^2$-norm, is simply because it is a computationally inexpensive approach which works well in practice.
The implementation of the error estimate can be seen in algorithm (7). This completes the implementation of the adaptive mesh generation algorithm (5).

Algorithm 7 Estimate error between two solutions

```
1: function EstimateError
2:     for all elements e_n in the coarse mesh that are refined do
3:         Compute the volume vol(u_h, e_n) by (3.10)
4:         for all sub-elements e_{n_k} of e_n do
5:             Compute the volume vol(u*_h, e_{n_k}) by (3.10)
6:             Add the sub-volumes up, i.e. vol(u*_h, e_n) := vol(u*_h, e_n) + vol(u*_h, e_{n_k})
7:         end for
8:         Error(n) := |vol(u_h, e_n) − vol(u*_h, e_n)|
9:     end for
10:    return Error
11: end function
```

3.5 Time-dependent Problems

In the the previous implementations we have dealt with a typical boundary value problem, the Poisson problem, where we discretize the spatial dimensions to get a system of linear equations that is simple to solve. In the following sections we introduce the dimension of time, and it shown how we discretize time solutions at the desired points in time. To introduce evolution equations we have used the heat equation:

$$\begin{aligned} \tfrac{\partial u}{\partial t} - \nabla^2 u = f \quad &\text{in} \quad \Omega \times (0,T) \\ u = g \quad &\text{on} \quad \partial\Omega \times (0,T) \\ u = u_0 \quad &\text{in} \quad \Omega \times \{0\} \end{aligned}$$

An observant person might notice that the heat equation looks quite similar to the Poisson equation, or more precisely: if u is constant in time, that is if its time-derivative is 0, then the heat equation is exactly Poisson's equation. So one could expect that we will arrive at a system of linear equations that is similar to that of Poisson. We will later see that there definitely is some we can reuse from earlier algorithms, however the dimension of time does complicate things somewhat. Recall the definition of a derivative through limits:

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = \lim_{k \to 0} \frac{h(t+k) - h(t)}{k}.$$

We can express this as an approximation

$$\frac{\partial}{\partial t} u(x,y,t) \approx \frac{u(x,y,t+k) - u(x,y,t)}{k}$$

when the time-step k is small. This type of approximation is known as a *finite difference* approximation, and just as there are finite element methods, there is also a large class of methods known as finite difference methods for solving BVPs and IVPs. However, even when using the finite element method, this finite difference approximation turns out to be useful for solving initial value problems.

3.6 Method of Lines or Rothe's Method?

Now that we have revealed how the dimension of time is discretized, the natural question of which type dimension one should discretize first arises, and does it even matter? It turns out it does make a non-trivial difference, at least in implementation. Obtaining a semi-discrete system through discretizing the spatial dimensions first is commonly known as the *method of lines*, whereas discretizing time first is known as *Rothe's method*, and both methods have their pros and cons.
If one wants to use the same finite element mesh for each time-step, it does not make a big difference in the solution: the methods become very similar. The implementation of the method of lines, though, is

some ways less difficult, since we can reuse a lot of the methods developed for the steady state problems. However, if one wants to change the mesh at each time-step, which makes sense if you are to implement a fully adaptive method, then using Rothe's method makes the most sense, since you choose the spatial discretization independently at each time-step. It is also possible to update the mesh every once in a while to get a semi-adaptive method. We will now describe the two methods in more detail.

Method of Lines

When applying the method of lines, the discretization of the spatial dimensions comes first. The weak formulation of the heat equation was found in the first chapter to be

$$\int_\Omega v\frac{\partial u}{\partial t}\,\mathrm{d}S + \int_\Omega \nabla v \nabla u\,\mathrm{d}S = \int_\Omega vf\,\mathrm{d}S \tag{3.12}$$

which discretizes to the matrix form

$$\mathbf{B}\frac{\partial}{\partial t}\hat{\mathbf{u}} + \mathbf{A}\hat{\mathbf{u}} = \boldsymbol{\ell} \tag{3.13}$$

where

$$\begin{aligned}\mathrm{B}_{ij} &= \int_\Omega e_i e_j\,\mathrm{d}S\\ \mathrm{A}_{ij} &= \int_\Omega \nabla e_i \nabla e_j\,\mathrm{d}S\\ \ell_i &= \int_\Omega e_i f\,\mathrm{d}S.\end{aligned}$$

In general the method of lines is simply a method where we discretize all but one dimensions, which in this case is the dimension of time,

Time-stepping Scheme

We have now arrived at a semi-discrete system, in which the next goal is to discretize time. We insert the approximation of the time derivative and get

$$\mathbf{B}\frac{1}{k}(\hat{\mathbf{u}}^{n+1} - \hat{\mathbf{u}}^n) + \mathbf{A}\hat{\mathbf{u}}^n = \boldsymbol{\ell}^n, \tag{3.14}$$

where we have used the notation with superscript $\hat{\mathbf{u}}^n$ to mean the nth point in time, i.e. $\hat{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}(t_n+k)$. From this linear system it is possible to solve for $\hat{\mathbf{u}}^{n+1}$, so that one can compute the next time step from the previous $\hat{\mathbf{u}}^n$, and this would be a perfectly valid approach. However, one can gain higher stability in the method using an *implicit* method. Let us first rewrite

$$\begin{aligned}\hat{\mathbf{u}}^{n+1} &= \hat{\mathbf{u}}^n + k\mathbf{B}^{-1}(-\mathbf{A}\hat{\mathbf{u}}^n + \boldsymbol{\ell}^n)\\ &= \hat{\mathbf{u}}^n + k\mathbf{p}(t_n, \hat{\mathbf{u}})\end{aligned}$$

where

$$\mathbf{p}(t_n, \hat{\mathbf{u}}) = \mathbf{B}^{-1}(-\mathbf{A}\hat{\mathbf{u}}^n + \boldsymbol{\ell}^n).$$

Again we stress that numerically the inverse of $\mathbf{B}$ will not actually be computed. In the implicit method we will approximate by also using the next time step, in the following way: we replace $\mathbf{p}(t_n, \hat{\mathbf{u}})$ by $\theta\mathbf{p}(t_{n+1}, \hat{\mathbf{u}}) + (1-\theta)\mathbf{p}(t_n, \hat{\mathbf{u}})$ for some $\theta \in [0, 1]$. The value of θ can be seen as a weighing of the two time-steps. The step of introducing this θ-weight may seem like some trick that is not well-founded, however methods for temporal discretization is a whole new topic which will not be handled in detail here. This concept comes from a family of methods known as Runge-Kutta methods. Using this we get

$$\begin{aligned}\hat{\mathbf{u}}^{n+1} &= \hat{\mathbf{u}}^n + k[\theta\mathbf{p}(t_{n+1}, \hat{\mathbf{u}}) + (1-\theta)\mathbf{p}(t_n, \hat{\mathbf{u}})]\\ &= \hat{\mathbf{u}}^n + k[\theta\mathbf{B}^{-1}(-\mathbf{A}\hat{\mathbf{u}}^{n+1} + \boldsymbol{\ell}^{n+1}) + (1-\theta)\mathbf{B}^{-1}(-\mathbf{A}\hat{\mathbf{u}}^n + \boldsymbol{\ell}^n)].\end{aligned}$$

The next step is to collect all terms using the new time-step. Multiplying by $\mathbf{B}$ and collecting yields

$$(\mathbf{B} + k\theta\mathbf{A})\hat{\mathbf{u}}^{n+1} = (\mathbf{B} - k(1-\theta)\mathbf{A})\hat{\mathbf{u}}^n + k(\theta\boldsymbol{\ell}^{n+1} + (1-\theta)\boldsymbol{\ell}^n). \tag{3.15}$$

This is the time-stepping scheme that arises from using the method of lines. The linear system can be solved, and it is seen that the solution at the new time-step, is solely depending on solution from the previous time-step.

Assembling $\mathbf{A}$ and $\boldsymbol{\ell}^n$ is done with nearly the same method as done for stead state problems, the only real difference being that $\boldsymbol{\ell}^n$ is dependent on time, which we have to account for, so we will not go into detail with these. However, we do have to figure how to assemble the matrix $\mathbf{B}$. Just as we did for other integrals, we define a following quantity as the integral over a single element e_n, as opposed to the entire domain, and then we add the quantities. The integral can be shown to be equal to

$$b_{i,j}^{(n)} = \int_{e_n} e_i^{(n)} e_j^{(n)} \,\mathrm{d}S = \frac{|\Delta|}{6}$$

if $i = j$, else if $i \neq j$ it is given by

$$b_{i,j}^{(n)} = \frac{|\Delta|}{12},$$

when using the piecewise affine basis functions $e_i = \phi_i$.

Rothe's Method

When using Rothe's method, the discretization of time is done before the spatial discretization. Using the same time-discretization approach as done in the method of lines, we get that

$$\frac{\partial u}{\partial t} = \nabla^2 u + f$$

discretizes into

$$u^{n+1} = u^n + k\theta(\nabla^2 u^{n+1} + f^{n+1}) + k(1-\theta)(\nabla^2 u^n + f^n). \tag{3.16}$$

To discretize space we multiply by the test function v on integrate, as we have done many times by now, which after also applying the divergence theorem yields

$$\langle u^{n+1}, v\rangle_2 = \langle u^n, v\rangle_2 + k\theta[-\langle\nabla u^{n+1}, \nabla v\rangle_2 + \langle f^{n+1}, v\rangle_2] + k(1-\theta)[-\langle\nabla u^n, \nabla v\rangle_2 + \langle f^n, v\rangle_2]. \tag{3.17}$$

Using the series expansions

$$v = \sum_j \hat{v}_j^n e_j^n \quad \text{and} \quad u^{n+1} = \sum_i \hat{u}_i^{n+1} e_i^{n+1}$$

etc, we get that after "cancelling" the test function and collecting terms on each side, this simplifies into

$$(\mathbf{B}^{n+1,n} + k\theta\mathbf{A}^{n+1,n})\mathbf{u}^{n+1} = (\mathbf{B}^{n,n} - k(1-\theta)\mathbf{A}^{n,n})\mathbf{u}^n + k\theta\boldsymbol{\ell}^{n+1} + k(1-\theta)\boldsymbol{\ell}^n \tag{3.18}$$

where

$$\begin{aligned} \mathrm{B}_{ij}^{n,m} &= \langle e_i^n, e_j^m\rangle_2 \\ \mathrm{A}_{ij}^{n,m} &= \langle\nabla e_i^n, \nabla e_j^m\rangle_2 \\ \ell_i^m &= \langle f^m, e_i^n\rangle_2. \end{aligned}$$

One thing to notice is that if the mesh is unchanging between steps, then $\mathbf{B}^{n+1,n} = \mathbf{B}^{n,n} = \mathbf{B}$, and similarly for $\mathbf{A}$, and the time-stepping scheme is exactly the same as in the method of lines. This scheme has the advantage that the mesh can change in between steps, but this also introduces some complications for the implementation. First off, we have to integrate over products of different basis functions, and we also have to somehow control that the sizes of the matrices and vectors are the same, so that the computations are defined.

Imposing Boundary Conditions

Imposing boundary conditions is done in a similar way as it is done for steady state problems, however we now have a slightly different system of equations which is

$$\mathbf{B}\frac{\partial}{\partial t}\hat{\mathbf{u}} = -\mathbf{A}\hat{\mathbf{u}} + \boldsymbol{\ell}, \tag{3.19}$$

where we know that $u(x_i, y_i, t) = g(x_i, y_i, t)$ if the number i represents a number of a boundary vertex. This means that we have

$$\frac{\partial u}{\partial t} = \frac{\partial g}{\partial t} \tag{3.20}$$

on the boundary, hence we can change each boundary row i in the following way:

$$\begin{bmatrix} \mathrm{B}_{1,1} & \mathrm{B}_{1,2} & \mathrm{B}_{1,3} & \mathrm{B}_{1,4} \\ 0 & 1 & 0 & 0 \\ \mathrm{B}_{3,1} & \mathrm{B}_{3,2} & \mathrm{B}_{3,3} & \mathrm{B}_{3,4} \\ \mathrm{B}_{4,1} & \mathrm{B}_{4,2} & \mathrm{B}_{4,3} & \mathrm{B}_{4,4} \end{bmatrix} \begin{bmatrix} \partial_t \hat{u}_1 \\ \partial_t \hat{u}_2 \\ \partial_t \hat{u}_3 \\ \partial_t \hat{u}_4 \end{bmatrix} = - \begin{bmatrix} \mathrm{A}_{1,1} & \mathrm{A}_{1,2} & \mathrm{A}_{1,3} & \mathrm{A}_{1,4} \\ 0 & 0 & 0 & 0 \\ \mathrm{A}_{3,1} & \mathrm{A}_{3,2} & \mathrm{A}_{3,3} & \mathrm{A}_{3,4} \\ \mathrm{A}_{4,1} & \mathrm{A}_{4,2} & \mathrm{A}_{4,3} & \mathrm{A}_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} + \begin{bmatrix} \ell_1 \\ \partial_t g_2 \\ \ell_3 \\ \ell_4 \end{bmatrix},$$

where ∂_t simply means the derivative with respect to t, and these derivatives will be estimated with finite difference approximations. An implementation of the time-stepping algorithm when using unchanging meshes can be seen in algorithm (8)

Algorithm 8 Solve initial value problem on unchanging mesh

```
function TimeSolve
    Assemble A, B, ℓ^n, and ℓ^{n+1}
    Impose boundary conditions on the system
    for all time-steps do
        Time-step by solving (3.15) to find û^{n+1}
        Impose boundary condition on û^{n+1}
        Assemble the load vector for the next step ℓ and impose boundary conditions.
    end for
end function
```

3.7 Numerical Stability of Time-stepping

An important property to look at for time-stepping schemes is whether or not the scheme is numerically stable. In short, a scheme is numerically stable if small errors do not magnify as we step in time, which eventually would cause the solution to blow up, and hence not converge to the real solution. In a numerical stable method the errors will not add up, they will either stay around at a constant magnitude at each time step, or even better, they will diminish as we step forward in time. Let us look at the presented time-stepping schemes, in the simple version where the mesh is unchanging and $f = g = 0$. Then the terms with the load vector vanish and we have

$$(\mathbf{B} + k\theta\mathbf{A})\hat{\mathbf{u}}^{n+1} = (\mathbf{B} - k(1-\theta)\mathbf{A})\hat{\mathbf{u}}^{n}. \tag{3.21}$$

By multiplying with $\mathbf{B}^{-1}$ and isolating the new solution, we get

$$\hat{\mathbf{u}}^{n+1} = \mathbf{M}\hat{\mathbf{u}}^{n} \tag{3.22}$$

where

$$\mathbf{M} = \left(\mathbf{I} + k\theta\mathbf{B}^{-1}\mathbf{A}\right)^{-1}\left(\mathbf{I} - k(1-\theta)\mathbf{B}^{-1}\mathbf{A}\right).$$

By applying this relation multiple times

$$\hat{\mathbf{u}}^{n+1} = \mathbf{M}\hat{\mathbf{u}}^{n} = \mathbf{M}(\mathbf{M}\hat{\mathbf{u}}^{n-1}) = \cdots = \mathbf{M}^{n+1}\hat{\mathbf{u}}^{0}. \tag{3.23}$$

Hence, the stability of the scheme is dependent on that the matrix $\mathbf{M}$ does not cause the initial solution $\hat{\mathbf{u}}^0$ to blow up. From linear algebra we know that, if a matrix is diagonalizable, it can be written on the form

$$\mathbf{M} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

where the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{M}$, and $\mathbf{\Lambda}$ is a diagonal matrix consisting of the eigenvalues. A matrix is diagonalizable if it has a full set of linearly independent eigenvectors. Hence

$$\mathbf{M}^{n+1} = \left(\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}\right)^{n+1} = \mathbf{V}\mathbf{\Lambda}\underbrace{\mathbf{V}^{-1}\mathbf{V}}_{\mathbf{I}}\mathbf{\Lambda}\mathbf{V}^{-1}\ldots\mathbf{V}\mathbf{\Lambda}\underbrace{\mathbf{V}^{-1}\mathbf{V}}_{\mathbf{I}}\mathbf{\Lambda}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Lambda}^{n+1}\mathbf{V}^{-1}.$$

Inserting this into (3.23), we get that

$$\hat{\mathbf{u}}^{n+1} = \mathbf{V}\mathbf{\Lambda}^{n+1}\mathbf{V}^{-1}\hat{\mathbf{u}}^{0}. \tag{3.24}$$

The only possible way that this can blow up, is if the diagonal matrix blows up, which can only happen if an eigenvalue λ_i satisfies $|\lambda_i| > 1$. On the contrary the scheme is numerically stable if all eigenvalues λ_i satisfy $|\lambda_i| \leq 1$, and errors will decay if $|\lambda_i| < 1$. This leads us to two further possibilities: $0 < \lambda_i < 1$ and $-1 < \lambda_i < 0$. The first of these choices will lead to smooth decay, whereas the second leads to (decaying) oscillations. This is analogous to looking at the expressions $(1/2)^n$ and $(-1/2)^n$ for $n \in \mathbb{N}$. The first will decay smoothly, whereas the second will have decaying oscillations.

It can be shown, but will not be in this text, that the eigenvalues η_i of $\mathbf{B}^{-1}\mathbf{A}$ are all non-negative. We can then express the eigenvalues λ_i of $\mathbf{M}$ in terms of η_i in the following way

$$\lambda_i = \frac{1 - k(1-\theta)\eta_i}{1 + k\theta\eta_i}. \tag{3.25}$$

From this expression it can be seen that the stability of the time-stepping scheme depends on the time-step k, the choice of the parameter θ, and the magnitude of the eigenvalues of $\mathbf{B}^{-1}\mathbf{A}$. Let us consider three different choices of θ.
The first possible choice is that $\theta = 0$. This time-stepping scheme is also known as the forward or explicit Euler method. The magnitude of the eigenvalues λ_i are then given by

$$|\lambda_i| = |1 - k\eta_i|.$$

This magnitude is less than or equal to one if we choose a time-step that satisfies

$$k \leq \frac{2}{\max\limits_i \eta_i}, \tag{3.26}$$

since η_i is nonnegative and real. In other words, the method is stable if we choose k such that (3.26) holds. This value can be very small, so the method is only stable for *very* small values of k, and because of this the method is not very useful in practice.

The second choice is $\theta = 1/2$. This time-stepping scheme is also known as the Crank-Nicolson method. Setting we $\theta = 1/2$ leads to the requirement that

$$|\lambda_i| = \left|\frac{1 - \frac{1}{2}k\eta_i}{1 + \frac{1}{2}k\eta_i}\right| \leq 1, \tag{3.27}$$

but this holds for *all* choices of k. This guarantees stability of the Crank-Nicolson method, which makes it a good choice in practice. It should be noted, however, that if we choose

$$k > \frac{2}{\max_i \eta_i}$$

then $-1 < \lambda_i < 0$ and we will experience the aforementioned (perhaps unwanted) decaying oscillations.

Lastly, we look at the possibility of $\theta = 1$. This is also known as the backward or implicit Euler method. Setting $\theta = 1$ yields

$$|\lambda_i| = \left| \frac{1}{1 + k\eta_i} \right| \leq 1, \tag{3.28}$$

which also holds for any choice of k (since $k\eta_i \geq 0$). Therefore the implicit Euler method is also stable for any time-step k. Additionally, we always have that λ_i is nonnegative and hence we will always experience smooth decay, making the implicit Euler method a good choice.

3.8 What next?

The presented implementation of the finite element solver, is an implementation that works but there are certainly areas that can be improved. In the implementation we have used piecewise affine basis function, i.e. piecewise first-order polynomials. A very popular choice is to make use of higher-order polynomials as basis functions. This does not necessarily mean that one makes a specific choice, say polynomials of degree three, but it is possible to make certain elements use a particular degree of polynomials while other elements may use another degree. Polynomials of higher degree are much better at approximating smooth functions, and therefore the convergence of the approximation will be much faster when using higher-order polynomials. Implementing an adaptive algorithm with these polynomials, lead to choices of refining elements as done in this implementation, but there is also a choice of increasing the degree of the basis function in that element. This algorithm can give a very fast convergence, however it is more complicated to implement. More on this subject can be seen in [8].

The implementation that will be shown in the end does not use this, but it is possible to have mesh-adaptivity even when time-stepping, however it does introduce some challenges as revealed in the last section. The goal is of course to get a suitable mesh at every point in time. This means that it is not enough to refine after each time-step, as this will only cause the mesh to be finer at each step, and possible be very refined in areas that no longer need to be that fine at that point in time. An approach to solving this problem is using a refinement/derefinement-algorithm. The idea is to after each time-step derefine the mesh, coarsening up areas that no longer need to be as fine, and then refining as it has been described in this text, thereby refining in the places that need refinement. A detailed description of a refinement/derefinement-algorithm is seen in [7]. In the algorithm all triangulations (meshes) are stored, and so when an element is chosen to be derefined, the element simply goes back to the point where it was coarser.

The next major change one can implement, is the use of unstructured mesh. In this implementation we have used a very structured mesh that uses right-angled triangles as elements. Not restricting one self to right-angled triangles, can improve the adaptive mesh generation algorithm even further. It is, of course, also possible to not use triangles at all, other element types such as squares are also possible.

For those interested in going further with implementing a finite element solver there are various places to look. Detailed descriptions of adaptive mesh generation can be found in [2],[3], and [8].

CHAPTER 4

Results

In the first few chapters we have delved into the theory behind the finite element method. Existence and uniqueness of solution has been verified, and the precision of the approximate solution has also been a subject for discussion. In the next chapter, the implementations of the necessary algorithms were presented. Using an implementation in `MATLAB` as described there, we will in this chapter present solutions to some examples.

4.1 Steady State Problems

In this section we will look at the now-familiar Poisson Dirichlet boundary value problem:

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in} \quad \Omega \subset \mathbb{R}^2 \\ u &= g \quad \text{on} \quad \partial\Omega \end{aligned}$$

for different choices of u, and Ω. In reality, the function u is of course unknown, it is in fact what we are looking for. However, these are constructed test problems, and it can be an advantage knowing the solution for analysis purposes.

Example 1. For the first example, we have chosen the solution given by

$$u(x, y) = -xye^{-x^2-y^2} \tag{4.1}$$

and the domain is the rectangle $(-2, 2) \times (-2, 4)$. The finite element approximation can be seen in figure (4.1). We are particularly interested in how well the adaptive algorithm performs versus a standard

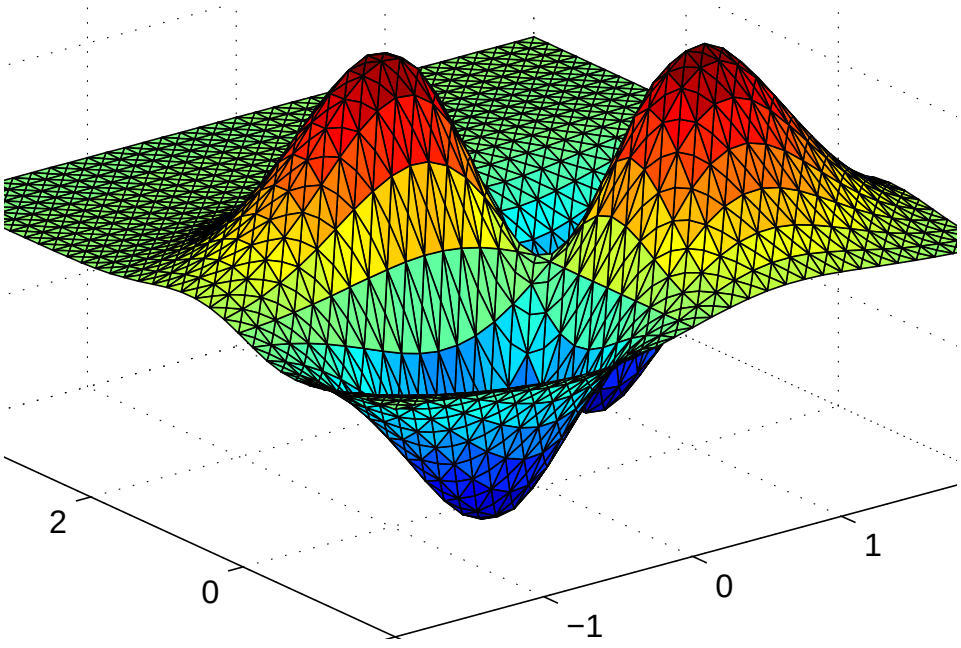


Figure 4.1: *A standard finite element approximation of (4.1).*

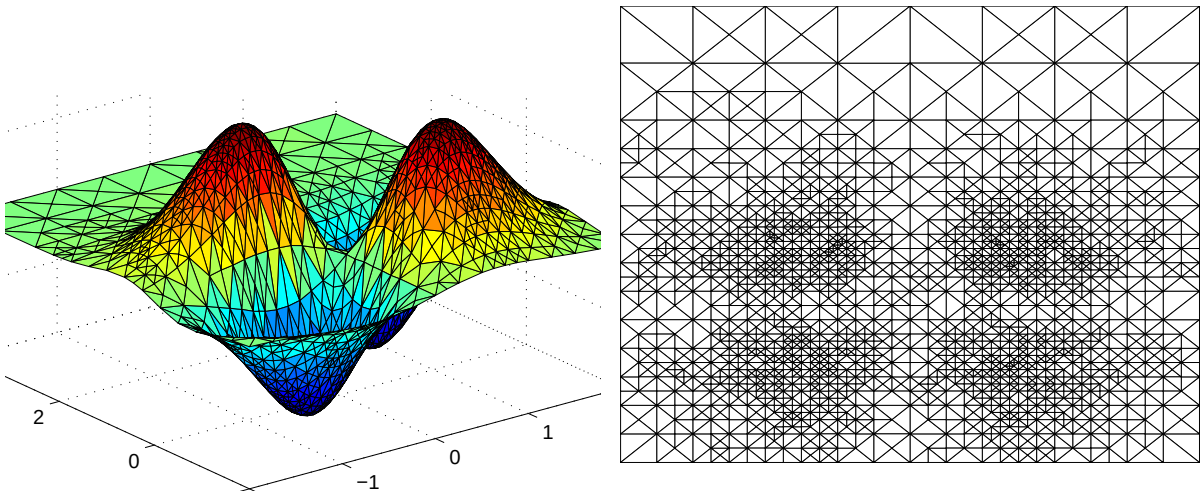


Figure 4.2: *An approximation to (4.1) using adaptive mesh refinement and the generated mesh.*

uniform refinement. In figure (4.2) the approximation with the adaptive mesh refinement algorithm can be seen along with the generated mesh, using an (estimated) error tolerance of $\varepsilon_{\text{tol}} = 10^{-3}$. It is clearly visible that the mesh is very refined in areas with higher curvature, and more coarse in the areas where the changes occurring are smaller. This is exactly what we want to achieve using the adaptive algorithm. This approximation has a smaller error than the solution seen in figure (4.1), even though some elements in the adaptive solution are larger (particularly at the top of the mesh).
The convergence of the estimated errors, using standard refinement versus adaptive refinement, can be seen in figure (4.3), where both algorithms are run until the error is below the tolerance. It is clear that the adaptive method achieves the same precision as the uniform refinement method with significantly fewer degrees of freedom (number of elements). In this example the adaptive and uniform methods terminate with 3294 and 32768 elements, respectively. This is to say that the adaptive algorithm achieves the same precision as the standard method, using only around one tenth of the number of elements.
It can also be seen, in the first few iterations, that the errors of the two methods are very similar. This is due to a very coarse initial mesh, where both methods will refine all elements in the beginning.

Example 2. In this example we will show how solutions iterate when using the adaptive mesh refinement algorithm. We will use the solution

$$u(x, y) = e^{-100\left((x-0.5)^2+(y-0.5)^2\right)} \tag{4.2}$$

on the standard square $(0, 1) \times (0, 1)$. This is a solution that has a steep peak around $(x, y) = (0.5, 0.5)$. In figure (4.4) the solutions of iteration $2, 4, 6$, and 8 can be seen. This illustrates how the solution on the coarse mesh is very far from the true solution, but as we refine elements the solution converges to a true solution.
Example 3. In the next example we use the solution

$$u(x, y) = \sin(2\pi x)e^{-x^2-y^2} \tag{4.3}$$

on the square $(0, 2) \times (0, 2)$. This example illustrates that we can get a good approximation even for functions that has large changes on the boundary, and it is also yet another example that shows that the adaptive algorithm refines strongly in particular areas. The solution and its mesh generated by the adaptive algorithm can be seen in figure (4.5), using an error tolerance of $\varepsilon_{\text{tol}} = 10^{-3.2}$. The convergence of the error of the two methods can be seen in figure (4.6). For this example the adaptive and uniform methods terminate with 2556 and 32768 elements, respectively.
In practice the two most important factors are likely memory requirements and computation time.

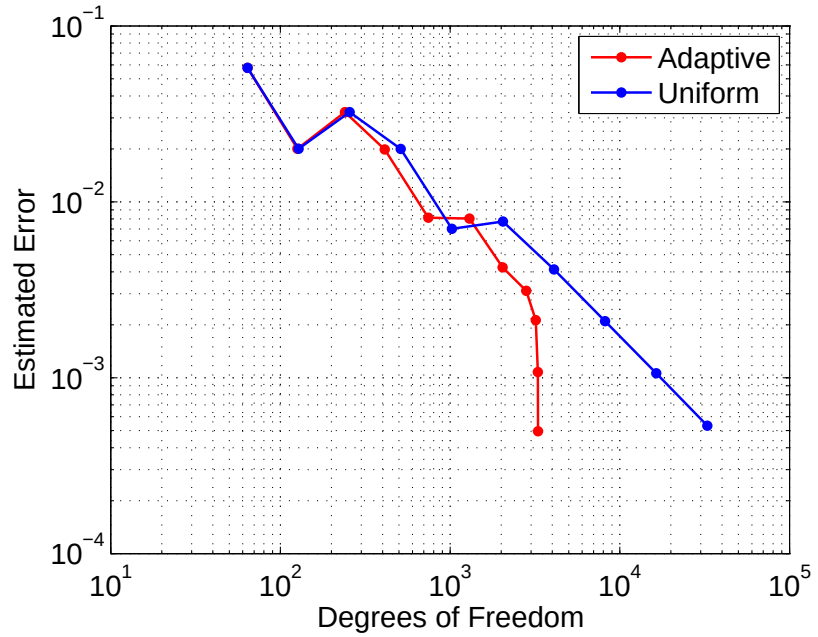


Figure 4.3: *The convergence of estimated errors of the two methods for (4.1).*

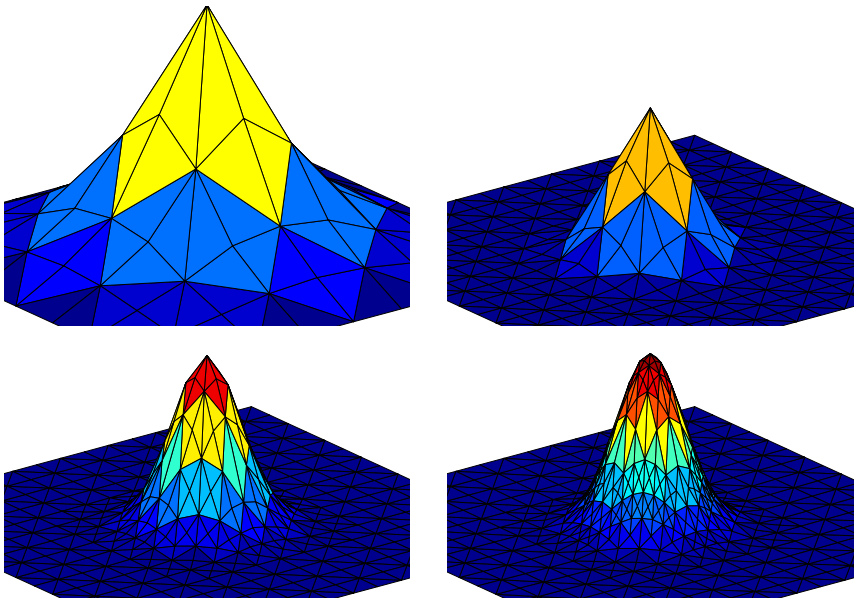


Figure 4.4: *Top: iteration* 2 *and* 4. *Bottom: iteration* 6 *and* 8. *Approximations of (4.2).*

Similarly to comparing the degrees of freedom required, we can compare the computation times of the two methods. The computation times of the two methods can be seen in figure (4.7) for this example (4.3). However, it should be noted the computation time depends highly on the how the program has been implemented, in which programming language, etc. Therefore the computation time should only be considered a piece of evidence, not a proof, that the adaptive algorithm outperforms the uniform algorithm.

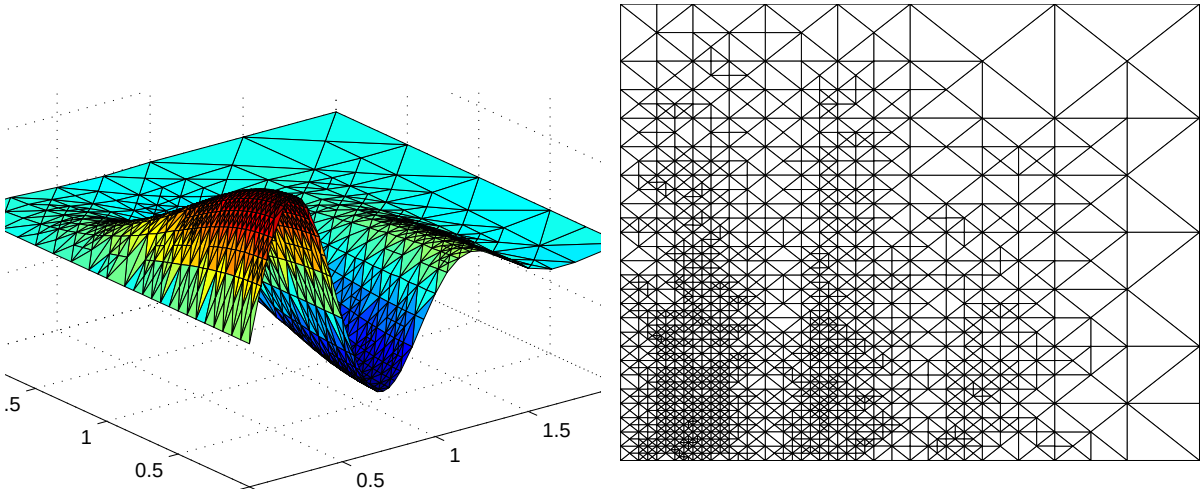


Figure 4.5: *An approximation and its mesh to (4.3) using adaptive mesh refinement.*

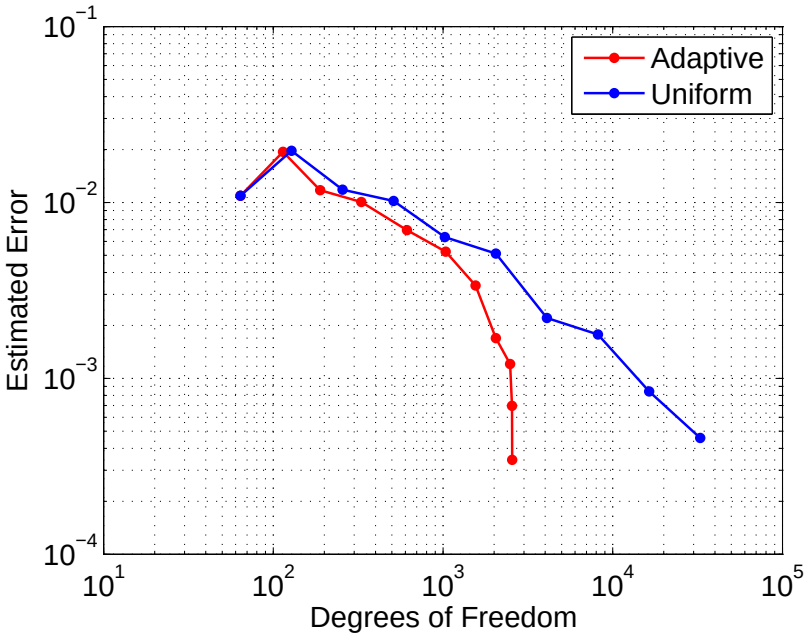


Figure 4.6: *The convergence of estimated errors of the two methods for (4.3).*

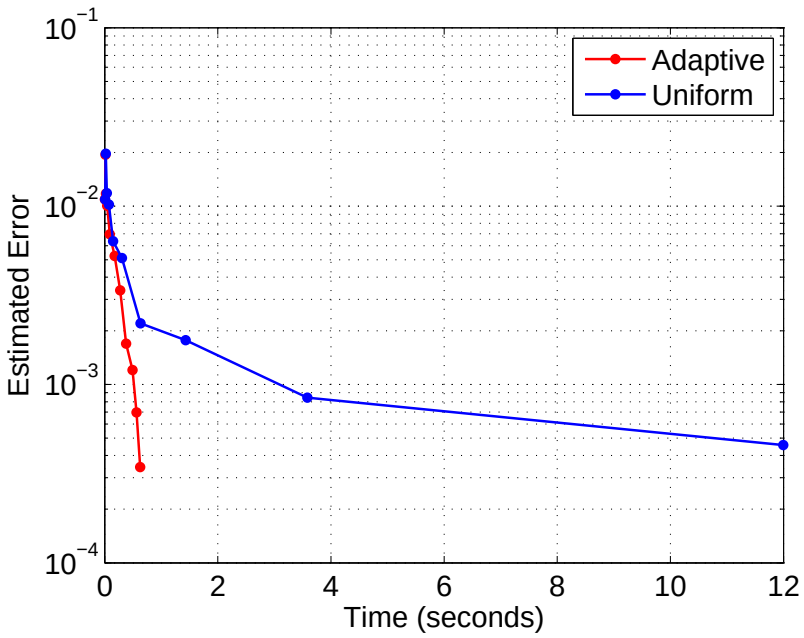


Figure 4.7: *The convergence of estimated errors of the two methods for (4.3).*

4.2 Time-dependent Problems

In this section we will look at results from the heat initial value problem

$$\begin{aligned} \tfrac{\partial u}{\partial t} - \nabla^2 u = f \quad &\text{in} \quad \Omega \times (t_0, T) \\ u = g \quad &\text{on} \quad \partial\Omega \times (t_0, T) \\ u = u_0 \quad &\text{in} \quad \Omega \times \{t_0\} \end{aligned} \tag{4.4}$$

for different choices of test solutions $u(x, y, t)$. These results are from the implementation of the method of lines time-stepping scheme.

Example 4. For the first time-dependent example we will let

$$u(x, y, t) = e^{-100\left((x-t)^2+(y-0.5)^2\right)} \tag{4.5}$$

which has a steep peak at the time-dependent point $(x, y) = (t, 0.5)$, hence the solution will look like a travelling peak. The finite element approximation on a non-changing mesh is plotted in figure (4.8) at four different moments in time, using the Crank-Nicolson ($\theta = 1/2$) time-stepping scheme. The solutions are viewed in (x, u)-view to make it easier to see that the peak does not lose height.

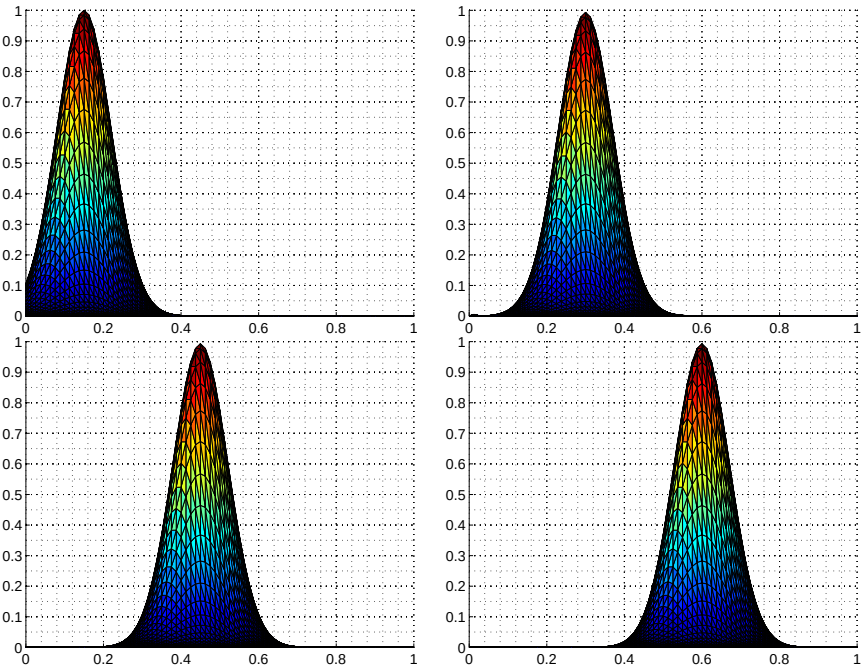


Figure 4.8: *Finite element approximation of (4.5) at* $t = 0.15, 0.30, 0.45, 0.60$.

Example 5. In this last example we will use the following solution

$$u(x, y, t) = \sin(2\pi x)\cos(2\pi y)\left(1 - \frac{1}{t+1}\right). \tag{4.6}$$

One thing to notice is that $u_0 = 0$, so at $t = 0$ the solution is just a flat surface, but as time progresses the function curves. The solution from the Implicit Euler ($\theta = 1$) time-stepping algorithm can be seen in figure (4.9). The figure shown uses a mesh of 800 elements.

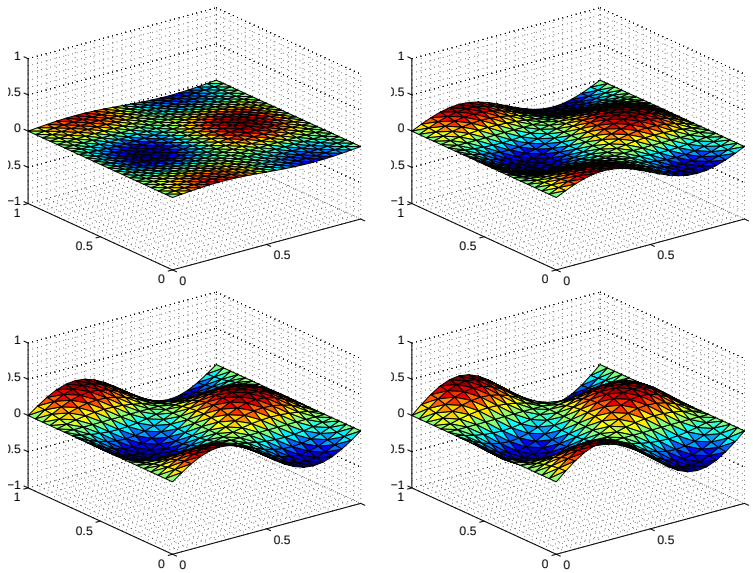


Figure 4.9: *Finite element approximation of (4.6) at* $t = 0.05, 0.25, 0.45, 0.60$.

We will now test the robustness of the Crank-Nicolson time-stepping scheme. To do this we will run the algorithm approximating (4.6) using an *erroneous* initial condition

$$u_0(x, y) = \frac{1}{2}\sin(4\pi x)\cos(4\pi y). \tag{4.7}$$

We will look at the nodal errors between the approximation and the true solution after K time-steps of length k. The nodal error is

$$E_i = |\hat{u}_i - u(x_i, y_i, kK)| \tag{4.8}$$

and we will look at the maximum error $E_{\max} = \max_i E_i$ and the mean error $E_{\text{mean}} = \text{mean}_i E_i$. The results can be seen in table (4.10).
We can see that the method converges to the correct solution even when using a wrong initial condition since the errors are small. The next observation we make is that the errors decay over time, which makes sense as we showed that this should be true for the Crank-Nicolson time-stepping scheme for any k. The errors decay quicker when using a small time-step as seen by comparing the two tables. The solutions seem to be more affected by the wrong initial condition when using a finer mesh initially, but the method does seem to converge regardless. The fact that the areas are larger on the finer mesh seem to be due to the oscillations of the Crank-Nicolson method which decay over time (the errors get smaller for a higher number of time-steps K).

Max/Mean Nodal Error ($k = 0.05$)			
K	DOF	$E_{\max}$	E_{mean}
10	$2 \cdot 20^2$	0.0650	0.0200
30	$2 \cdot 20^2$	0.0340	0.0082
60	$2 \cdot 20^2$	0.0210	0.0078
10	$2 \cdot 30^2$	0.1438	0.0211
30	$2 \cdot 30^2$	0.0812	0.0073
60	$2 \cdot 30^2$	0.0416	0.0057
10	$2 \cdot 40^2$	0.2048	0.0218
30	$2 \cdot 40^2$	0.1287	0.0071
60	$2 \cdot 40^2$	0.0786	0.0053

Max/Mean Nodal Error ($k = 0.01$)			
K	DOF	$E_{\max}$	E_{mean}
10	$2 \cdot 20^2$	0.0227	0.0033
30	$2 \cdot 20^2$	0.0082	0.0030
60	$2 \cdot 20^2$	0.0101	0.0040
10	$2 \cdot 30^2$	0.0589	0.0046
30	$2 \cdot 30^2$	0.0150	0.0023
60	$2 \cdot 30^2$	0.0068	0.0022
10	$2 \cdot 40^2$	0.0100	0.0055
30	$2 \cdot 40^2$	0.0304	0.0027
60	$2 \cdot 40^2$	0.0112	0.0018

Table 4.10: *The maximum and mean nodal errors for various parameters for approximation of (4.6), using erroneous initial condition.*

CHAPTER 5

Conclusion

The mathematical foundation behind the finite element method has been presented, showing rigorously how and why the method works. The major theorems regarding a large class of steady state problems are presented, in particular the existence and uniqueness of the Galerkin approximation is discussed, and its numerical validity is proved.

The implementation of an adaptive mesh refinement algorithm has been presented, and the performance of the method is clearly superior to a standard nonadaptive method. The adaptive algorithm achieves similar precision using only a fraction of the memory requirements and computation-time when compared to the standard method.

Lastly, a standard method for solving time-dependent partial differential equations using the finite element method has been presented. The method is shown to be numerically stable and accurate.

There are a few places where the project easily could worked further upon. The most obvious places would be to implement adaptivity in the heat equation solver, and to implement the use of higher-order polynomials as basis functions.

Bibliography

[1] O. Christensen. *Functions, Spaces, and Expansions*. Birkhäuser, 2010. ISBN 978-0817649791.

[2] A. P. Engsig-Karup. The spectral/hp-finite element method for partial differential equations. 2013.

[3] M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, 2006. ISBN 978-0898716146.

[4] J. K. Hunter. *Notes on Partial Differential Equations*. 2014.

[5] E. Kreyszig. *Introductory Functional Analysis with Applications*. John Wiley & Sons, 1989. ISBN 978-0471504597.

[6] J. D. Meiss. *Differential Dynamical Systems*. SIAM, 2007. ISBN 978-0898716351.

[7] M.-C. Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. 1989.

[8] P. Solín, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods*. Chapman & Hall, 2003. ISBN 978-1584884385.

APPENDIX A

Time Log

Time	Description
September	I began studying the theoretical background of the finite element method. Understanding of the weak formulation, etc. I began implementing a 1D finite element solver to prepare for the 2D-solver.
October	Further study of the theory. I began writing the section on preliminary theory. Work on the 2D adaptive solver began.
November	Most of the theory on steady state problems is written by the end of November, however only a small portion of the implementation is written. The adaptive algorithm works decently, but with some small issues.
December	The chapter on implementation is beginning to form. I study the various approaches to solving time-dependent problems. The adaptive algorithm for steady state problems is completely fixed.
January	The beginning of the month was spent on finishing up most of the chapter on theory and sections in chapter on implementation. Throughout the middle of the month I finish up the solver for time-dependent problems. Results are added, introduction is written, and the chapter on implementation continues. From here on I attempt in various ways to implement an adaptive solver for time-dependent problems but unfortunately I don't succeed. Most of the final week is spent on finishing the thesis.

APPENDIX B

Code

Generate Mesh

```
function [EToV,VX,VY] = GenerateMesh(x0,y0,L1,L2,noelms1,noelms2)
% Purpose: Generates mesh data structure.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   x0: x-position of the bottom left corner of rectangle.
%   y0: y-position of the bottom left corner of rectangle.
%   L1: Width of rectangle.
%   L2: Height of rectangle.
%   noelms1: Number of elements the width is divided into.
%   noelms2: Number of elements the height is divided into.
%
% OUTPUT PARAMETERS:
%   EToV: Element-to-Vertex table.
%   VX: x-position of vertices.
%   VY: y-position of vertices.

[VX, VY] = xy(x0,y0,L1,L2,noelms1,noelms2);
EToV = ConstructEToV(noelms1,noelms2);
end

function [VX, VY] = xy(x0,y0,L1,L2,noelms1,noelms2)
% Purpose: Computes positions of vertices of triangular
%          mesh in a rectangular domain.
nonodes = (noelms1+1)*(noelms2+1);
VX = zeros(nonodes,1);
VY = zeros(nonodes,1);
ElementWidth = L1/noelms1;
ElementHeight = L2/noelms2;

CurrentRow = noelms2;
CurrentCol = 0;
for i = 1:nonodes
    VX(i) = x0+CurrentCol*ElementWidth;
    VY(i) = y0+CurrentRow*ElementHeight;
    CurrentRow = CurrentRow-1;
    if CurrentRow == -1
        CurrentRow = noelms2;
        CurrentCol = CurrentCol+1;
    end
end
end

function EToV = ConstructEToV(noelms1, noelms2)
% Purpose: Constructs Element-to-Vertex table.
EToV = zeros(2*noelms1*noelms2,3);
for j = 1:noelms1
    index = (2*noelms2)*(j-1)+1;
    A = (1+noelms2)*j+(1:noelms2)';
    B = (1+noelms2)*(j-1)+(1:noelms2)';
    EToV(index:2:(index + 2*noelms2-1),1:3) = [A , B , A + 1];
    EToV((index+1):2:(index+2*noelms2),1:3)   = [B+1 , A+1 , B];
```

```
end
end
```

Assemble

```
function [A, l] = Assemble(VX,VY, EToV, ffun, f)
% Purpose: Assembles stifness matrix A and load vector l.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%   EToV: Element-to-Vertex table.
%   ffun: Right-hand side function.
%   f: Right-hand side as a function handle.
%
% OUTPUT PARAMETERS:
%   A: Stiffness matrix.
%   l: Load vector.

% Initializing
M = length(VX);
nmax = 4*M;

l = zeros(M,1);

ai = ones(nmax,1);
aj = ones(nmax,1);
as = zeros(nmax,1);

count = 1;

% Fills in the stiffness matrix and load vector.
for i = 1:length(EToV(:,1))
    n = EToV(i,1:3);
    %keyboard
%     centerpoint = [1/3*(VX(n(1))+VX(n(2))+VX(n(3))),...
%                    1/3*(VY(n(1))+VY(n(2))+VY(n(3)))];
%     fcp = f(centerpoint(1),centerpoint(2));
%     ftilde = (ffun(n(1))+ffun(n(2))+ffun(n(3))+2*fcp)/5;
    ftilde = (ffun(n(1))+ffun(n(2))+ffun(n(3)))/3;
    [Δ,abc] = ElementQuantities(i,VX,VY,EToV);

    for r = 1:3
        l(n(r)) = l(n(r)) + abs(Δ)/3*ftilde;
        for s = 1:3
            k = 1/(4*abs(Δ))*(abc(r,2)*abc(s,2)+abc(r,3)*abc(s,3));
            ai(count) = n(r);
            aj(count) = n(s);
            as(count) = k;
            count = count + 1;

        end
    end
end

A = sparse(ai,aj,as);
end
```

Find Boundary Nodes

```
function bnodes = BoundaryNodesRectangle(VX,VY)
% Purpose: Finds boundary nodes of rectangular domain.
```

```
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%
% OUTPUT PARAMETERS:
%   bnodes: Boundary nodes.

xmin = min(VX);
xmax = max(VX);
ymin = min(VY);
ymax = max(VY);

count = 1;

for i = 1:length(VX)
    if VX(i) == xmin || VX(i) == xmax || VY(i) == ymin || VY(i) == ymax
        bnodes(count) = i;
        count = count + 1;
    end
end
end
```

Impose Boundary Conditions

```
function [A,l] = DirichletBC(bnodes,bf,A,l)
% Purpose: Imposes boundary conditions on the stiffness matrix A and and load vector b.
%
% INPUT PARAMETERS:
%   bnodes: Boundary nodes within Gamma_2
%   f: Dirichlet boundary conditions
%   A: LHS-matrix for the problem
%   b: RHS-vector for the problem
%
% OUTPUT PARAMETERS:
%   A: Modified stiffness matrix.
%   l: Modified load vector.

N = length(bnodes);


% Imposes boundary condition on the stiffness matrix and the load vector.
for n = 1:N
    i = bnodes(n);
    A(i,:) = 0;
    l(:) = l(:) - A(:,i)*bf(i);
    A(:,i) = 0;
    A(i,i) = 1;
end

% Modifies the RHS for the nodes in the boundary
for i = 1:N
    n = bnodes(i);
    l(n) = bf(n);
end


end
```

Solve System

```
function u = SolveSystem(VX,VY,EToV,bfun,ffun,f)
```

```
% Purpose: Sets up the system of linear equations, imposes
%          Dirichlet boundary conditions. Then solves the system.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%   EToV: Element-to-Vertex table.
%   bfun: Dirichlet boundary function.
%   ffun: Right-hand side function.
%   f: Right-hand side function as a function handle.
%
% OUTPUT PARAMETERS:
%   u: Solution to the linear system.

% Find boundary nodes
bnodes = BoundaryNodesRectangle(VX,VY);

% Assemble system
[A,l] = Assemble(VX,VY,EToV,ffun,f);

% Impose Dirichlet boundary conditions.
[A,l] = DirichletBC(bnodes,bfun,A,l);

% Solve system.
u = A \ l;
end
```

Element Quantities

```
function [Δ,abc] = ElementQuantities(n, VX, VY, EToV)
% Purpose: Computes various quantities corresponding for a given element.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   n: Element number.
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%   EToV: Element-to-vertex table.
%
% OUTPUT PARAMETERS:
%   Δ: Area of element.
%   abc: Coefficients in 1/(2*Δ)*(a+bx+cy).

abc = zeros(3,3);

% Find the x-vertices for element n
x(1) = VX(EToV(n,1));
x(2) = VX(EToV(n,2));
x(3) = VX(EToV(n,3));
% Find the y-vertices for element n
y(1) = VY(EToV(n,1));
y(2) = VY(EToV(n,2));
y(3) = VY(EToV(n,3));

% Calculate area of element.
Δ = 1/2*(x(2)*y(3)-y(2)*x(3)-(x(1)*y(3)-y(1)*x(3))+x(1)*y(2)-y(1)*x(2));
for i=1:3
    j = mod(i,3)+1;
    k = mod(j,3)+1;
    abc(i,1) = x(j)*y(k)-x(k)*y(j);
    abc(i,2) = y(j)-y(k);
    abc(i,3) = x(k)-x(j);
end
end
```

Element Bisection

```
function [EToV,VX,VY,idxmarked,EToE,Old2New] = ElementBisection(EToV,VX,VY,k,idxmarked,EToE,Old2New)
% Purpose: Recursively bisects elements into smaller elements.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   EToV: Element-to-vertex table.
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%
%   idxmarked: Indices of marked elements.
%   EToE: Element-to-Element table.
%   Old2New: Data structure that links together the two meshes.
%
% OUTPUT PARAMETERS:
%   EToV: Element-to-vertex table.
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%   idxmarked: Indices of marked elements.
%   EToE: Element-to-Element table.
%   Old2New: Data structure that links together the two meshes.

if EToE(k,2) == 0
    in = length(VX)+1;
    ie = length(EToE)+1;
    ki = EToV(k,:);
    ei = EToE(k,:);
    % Update EToV
    EToV(k,:) = [in, ki(3), ki(1)];
    EToV(end+1,:) = [in, ki(1), ki(2)];
    % Update EToE
    EToE(k,:) = [ei(2) ei(3) ie];
    EToE(ie,:) = [k ei(1) ei(2)];
    for i = 1:3
        if EToE(ei(1),i) == k
            EToE(ei(1),i) = ie;
        end
    end
    % Update node vector
    VX(in) = (VX(ki(2))+VX(ki(3)))/2;
    VY(in) = (VY(ki(2))+VY(ki(3)))/2;
    idxmarked(k) = 0;
    Old2New(k) = ie;
else
    %Find neighbour at base
    j = EToE(k,2);
    if EToE(j,2) == k
        ki = EToV(k,:);
        ji = EToV(j,:);
        eki = EToE(k,:);
        eji = EToE(j,:);
        in = length(VX)+1;
        ie = length(EToE)+1;
        % Update EToV
        EToV(k,:) = [in ki(1) ki(2)];
        EToV(j,:) = [in ji(3) ji(1)];
        EToV(end+1:end+2,:) = [in ki(3) ki(1); ...
                               in ji(1) ji(2)];
        % Update EToE
        EToE(k,:) = [ie eki(1) j];
        EToE(j,:) = [k eji(3) ie+1];
        EToE(ie:ie+1,:) = [ie+1 eki(3) k; ...
            j eji(1) ie];
        if eki(3) ≠ 0
```

```
            for i = 1:3
                if EToE(eki(3),i) == k
                    EToE(eki(3),i) = ie;
                end
            end
        end
        if eji(1) ~= 0
            for i = 1:3
                if EToE(eji(1),i) == j
                    EToE(eji(1),i) = ie+1;
                end
            end
        end
        % Update node vector
        VX(in) = (VX(ki(2))+VX(ki(3)))/2;
        VY(in) = (VY(ki(2))+VY(ki(3)))/2;
        idxmarked(k) = 0;
        idxmarked(j) = 0;
        Old2New(k) = ie;
        Old2New(j) = ie+1;
    else
        [EToV,VX,VY,idxmarked,EToE,Old2New] = ElementBisection(EToV,VX,VY,j,idxmarked,EToE,Old2New);
        [EToV,VX,VY,idxmarked,EToE,Old2New] = ElementBisection(EToV,VX,VY,k,idxmarked,EToE,Old2New);
    end
end

end
```

Estimate Error

```
function err = EstimateError(EToVc,EToVf,VXc,VYc,VXf,VYf,uhatc,uhatf,Old2New)
% Purpose: Estimates error by comparing solutions of two meshes.
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   EToVc: Element-to-Vertex table of coarse mesh.
%   EToVf: Element-to-Vertex table of refined mesh.
%   VXc: x-position of vertices of coarse mesh.
%   VYc: y-position of vertices of coarse mesh.
%   VXf: x-position of vertices of refined mesh.
%   VYf: y-position of vertices of refined mesh.
%   uhatc: Approximate solution on coarse mesh.
%   uhatc: Approximate solution on refined mesh.
%   Old2New: Data structure that links together the two meshes.
%
% OUTPUT PARAMETERS:
%   err: Estimated error.

err = zeros(length(EToVc),1);

for i = 1:length(EToVc)
    if Old2New(i) ~= 0

        c = EToVc(i,:);
        XYZ = sortrows([VXc(c),VYc(c),uhatc(c)],3);
        xx = XYZ(:,1);
        yy = XYZ(:,2);
        zz = XYZ(:,3);
        %uc = (zz(1)+zz(2)+zz(3))/6*(xx(1)*yy(2)-xx(2)*yy(1)+xx(2)*yy(3)-xx(3)*yy(2)+xx(3)*yy(1)-xx(1)*yy(
        uc = (abs(zz(1))+abs(zz(2))+abs(zz(3)))/6*(xx(1)*yy(2)-xx(2)*yy(1)+xx(2)*yy(3)-xx(3)*yy(2)+xx(3)*y
        n = i;
        m(1) = n;
        in = 2;
        while n ~= 0
            n = Old2New(n);
```

```
            if n ~= 0
                m(in) = n;
                in = in + 1;
                if n > numel(Old2New)
                    n = 0;
                end
            end
        end
        uf = 0;
        for j = 1:length(m)
            f = EToVf(m(j),:);

            XYZ = sortrows([VXf(f),VYf(f),uhatf(f)],3);
            xx = XYZ(:,1);
            yy = XYZ(:,2);
            zz = XYZ(:,3);
            %u = (zz(1)+zz(2)+zz(3))/6*(xx(1)*yy(2)-xx(2)*yy(1)+xx(2)*yy(3)-xx(3)*yy(2)+xx(3)*yy(1)-xx(1)*yy(3))
            u = (abs(zz(1))+abs(zz(2))+abs(zz(3)))/6*(xx(1)*yy(2)-xx(2)*yy(1)+xx(2)*yy(3)-xx(3)*yy(2)+xx(3)*yy(1
            uf = uf + u;
        end
        err(i) = abs(uc-uf);
    end

end


end
```

Adaptive Solve

```
function [u, EToV, VX, VY, DOF, Error, h, time, AllSolutions] = ...
    AdaptiveSolve(ffun,bfun,eTol,method,x0,y0,L1,L2, noelms1,noelms2, maxIter, maxDOF,t)
% Purpose: Solves the 2-dimensional Dirichlet BVP on a rectangle adaptively.
%      -u_xx - u_yy = ffun    in Omega
%                 u = bfun    on Boundary(Omega)
% Author: Matias Fjeldmark
%
% INPUT PARAMETERS:
%   ffun: Right-hand side function.
%   bfun: Dirichlet boundary function.
%   eTol: Error-tolerance (default: 10^(-4)).
%   method: Adaptive method if set to 0, else uniform refinement (default: 0).
%   x0: x-position of the bottom left corner of rectangle (default: 0).
%   y0: y-position of the bottom left corner of rectangle (default: 0).
%   L1: Width of rectangle (default: 1).
%   L2: Height of rectangle (default: 1).
%   noelms1: Initial number of elements the width is divided into (default: 4).
%   noelms2: Initial number of elements the height is divided into (default: 4).
%   maxIter: Maximum number of iterations (default: 20).
%   maxDOF: Maximum number of degrees of freedom (default: 100 000).
%
% OUTPUT PARAMETERS:
%   u: Approximate solution.
%   EToV: Element-to-vertex table.
%   VX: x-position of vertices.
%   VY: y-position of vertices.
%   DOF: Number of degrees of freedom.
%   Error: Approximation of error.
%   h: Mean step-size of each iteration.
%   AllSolutions: Struct containing u, VX, VY, and EToV of all iterations.

% Set default values.
if nargin < 12
    maxDOF = 100000;
end
```

```
if isempty(maxDOF)
    maxDOF = 100000;
end
if nargin < 11
    maxIter = 20;
end
if isempty(maxIter)
    maxIter = 100000;
end
if nargin < 10
    noelms2 = 4;
end
if isempty(noelms2)
    noelms2 = 4;
end
if nargin < 9
    noelms1 = 4;
end
if isempty(noelms1)
    noelms1 = 4;
end
if nargin < 8
    L2 = 1;
end
if isempty(L2)
    L2 = 1;
end
if nargin < 7
    L1 = 1;
end
if isempty(L1)
    L1 = 1;
end
if nargin < 6
    y0 = 0;
end
if isempty(y0)
    y0 = 0;
end
if nargin < 5
    x0 = 0;
end
if isempty(x0)
    x0 = 0;
end
if nargin < 4
    method = 0;
end
if isempty(method)
    method = 0;
end
if nargin < 3
    eTol = 10^(-4);
end
if isempty(eTol)
    eTol = 10^(-4);
end
if nargin < 2
    bfun = @(x,y) 0*x;
end
if isempty(bfun)
    bfun = @(x,y) 0*x;
end
if nargin < 1
    error('Right-hand side function (f) was not specified.')
end
```

```
% Initialize the mesh and find initial solution.
[EToVc,VXc,VYc] = GenerateMesh(x0,y0,L1,L2,noelms1,noelms2);

if nargin == 13
    fc = ffun(VXc,VYc,t);
    bfc = bfun(VXc,VYc,t);
else
    fc = ffun(VXc,VYc);
    bfc = bfun(VXc,VYc);
end

uc = SolveSystem(VXc,VYc,EToVc,bfc,fc,ffun);

% Mark all elements for refinement in first iteration.
idxmarked = ones(length(EToVc),1);

EToEf = ConstructEToE(EToVc);

% Initialize the refined mesh.
EToVf = EToVc;
VXf = VXc;
VYf = VYc;

z=1;
Continue=true;
iter = 0;
stopReason = ' ';
while(Continue)
    tic
    % If non-adaptive, do uniform refinement.
    if method ~= 0
        idxmarked = ones(length(EToVc),1);
    end

    % Bisect marked elements.
    Old2New = zeros(length(EToVc),1);

    for j=1:length(idxmarked)
        if idxmarked(j) == 1
            [EToVf,VXf,VYf,idxmarked,EToEf,Old2New] =...
                ElementBisection(EToVf,VXf,VYf,j,idxmarked,EToEf,Old2New);
        end
    end



    iter = iter + 1;
    if iter >= maxIter
        Continue = false;
        stopReason = sprintf('Maximum number of iterations (%d) reached. \n',maxIter);
    end


    % Solve system for refined mesh.
    if nargin == 13
        bf = bfun(VXf,VYf,t);
        ff = ffun(VXf,VYf,t);
    else
        bf = bfun(VXf,VYf);
        ff = ffun(VXf,VYf);
    end

    uf = SolveSystem(VXf,VYf,EToVf,bf,ff,ffun);
```

```
% Estimate error.
E = EstimateError(EToVc,EToVf,VXc,VYc,VXf,VYf,uc,uf,Old2New);

% Define marked elements.
idxmarked = E > eTol;

% Check if there are elements that are marked.
if sum(idxmarked) == 0
    Continue = false;
    stopReason = sprintf('Precision reached after %d iterations.\n',iter);
end

for j=1:length(idxmarked)
    if Old2New(j) ~= 0
        n = Old2New(j);
        while(n~=0)
            idxmarked(n) = idxmarked(j);
            if n > numel(Old2New)
                n = 0;
            else
                n = Old2New(n);
            end

        end
    end
end

% The refined mesh is the coarse mesh for next iteration.
EToVc = EToVf;
VXc = VXf;
VYc = VYf;
uc = uf;

% Information about each iteration stored, if required.
if nargout > 4
    if nargout > 5
        if nargout > 6
            if nargout > 8
                AllSolutions.u{z} = uc;
                AllSolutions.VX{z} = VXc;
                AllSolutions.VY{z} = VYc;
                AllSolutions.EToV{z} = EToVc;
            end
            h(z) = min(sqrt((VXc(EToVc(:,2))-...
                VXc(EToVc(:,3))).^2+(VYc(EToVc(:,2))-VYc(EToVc(:,3))).^2));
        end
        Error(z) = max(E);
    end
    DOF(z) = length(EToVc);
end

% Check if degrees of freedom exceed maximum.
if length(EToVc) > maxDOF
    Continue = false;
    stopReason = sprintf('Maximum number of degrees of freedom (%d) exceeded.\n', maxDOF);
end

%Printed information if computation time is high.
if z>1
    time(z) = time(z-1) + toc;
else
    time(z) = toc;
end
if time(end) > 20
    fprintf('Iteration number %d completed.\n',iter);
    fprintf('Degrees of freedom: %d\n',length(EToVc));
```

```
        fprintf('Time elapsed: %.2f seconds.\n\n',time(end));
    end
    z = z + 1;
end

% Finalize output and print what caused the algorithm to terminate.

u = uc;
EToV = EToVc;
VX = VXc;
VY = VYc;
fprintf(stopReason);
end
```

Assemble Time-dependent Matrix

```
function [A,B] = AssembleMatrixTime(EToV,VX,VY)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

nmax = 4*length(VX);
ai = ones(nmax,1);
aj = ones(nmax,1);
as = zeros(nmax,1);
bs = zeros(nmax,1);

count = 1;
N = size(EToV,1);
for n = 1:N
    [Δ,abc] = ElementQuantities(n,VX,VY,EToV);
    C = abs(Δ)/12*...
        [2, 1, 1;
        1, 2, 1;
        1, 1, 2];
    for r = 1:3
        for c = 1:3
            k = 1/(4*abs(Δ))*(abc(r,2)*abc(c,2)+abc(r,3)*abc(c,3));
            b = C(r,c);
            ai(count) = EToV(n,r);
            aj(count) = EToV(n,c);
            as(count) = k;
            bs(count) = b;
            count = count + 1;
        end
    end
end
A = sparse(ai,aj,as);
B = sparse(ai,aj,bs);
end
```

Assemble Time-dependent Load Vector

```
function [l1, l0] = AssembleLoadTime(EToV, VX, VY, f, t, k)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

l1 = zeros(length(VX),1);
l0 = zeros(length(VX),1);

for i = 1:length(EToV(:,1))
    n = EToV(i,1:3);
    centerpoint = [1/3*(VX(n(1))+VX(n(2))+VX(n(3))),...
        1/3*(VY(n(1))+VY(n(2))+VY(n(3)))];
```

```
    t1 = t+k;
    ftilde1 = 1/5*(f(VX(n(1)),VY(n(1)),t1)+...
        f(VX(n(2)),VY(n(2)),t1) + f(VX(n(3)),VY(n(3)),t1)+...
        2*f(centerpoint(1),centerpoint(2),t1));

    if nargout == 2
        t0 = t;
        ftilde0 = 1/5*(f(VX(n(1)),VY(n(1)),t0)+...
            f(VX(n(2)),VY(n(2)),t0) + f(VX(n(3)),VY(n(3)),t0)+...
            2*f(centerpoint(1),centerpoint(2),t0));
    end

    [Δ] = ElementQuantities(i,VX,VY,EToV);
    for r = 1:3
        l1(n(r)) = l1(n(r)) + abs(Δ)/3*ftilde1;
        if nargout == 2
            l0(n(r)) = l0(n(r)) + abs(Δ)/3*ftilde0;
        end
    end
end

end
```

Dirichlet Boundary Condition Time

```
function [A,B,l0,l1] = DirichletBCTime(bnodes, VX, VY, bf, A, B, l0,l1, t,k) %(bnodes, EToV,VX,VY,bf,g,A,B
% Purpose: Imposes boundary conditions on the stiffness matrix A and and load vector b.
%
% INPUT PARAMETERS:
%   bnodes: Boundary nodes within Gamma_2
%   f: Dirichlet boundary conditions
%   A: LHS-matrix for the problem
%   b: RHS-vector for the problem
%
% OUTPUT PARAMETERS:
%   A: Modified stiffness matrix.
%   l: Modified load vector.

N = length(bnodes);
bfvec1 = bf(VX, VY, t);
bfvec2 = bf(VX, VY, t+0.25*k);
bfvec3 = bf(VX, VY, t+0.5*k);

dbfdt0 = 1/(4*k)*(bfvec2 - bfvec1);
dbfdt1 = 1/(4*k)*(bfvec3 - bfvec2);
for n = 1:N
    i = bnodes(n);
    A(i,:) = 0;
    l0(i) = dbfdt0(i);
    l1(i) = dbfdt1(i);
    B(i,:) = 0;
    B(i,i) = 1;
end
end
```

Solve Time-dependent Problem

```
function [U, EToV, VX, VY] = ...
    TimeSolve(f,u0,g,k,K,t0,x0,y0,L1,L2,NoElms1,NoElms2,theta)
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here
if nargin < 13
```

```
    theta = 1/2;
end
if isempty(theta)
    theta = 1/2;
end
if nargin < 12
    NoElms2 = 64;
end
if isempty(NoElms2)
    NoElms2 = 64;
end
if nargin < 11
    NoElms1 = 64;
end
if isempty(NoElms1)
    NoElms1 = 64;
end
if nargin < 10
    L2 = 1;
end
if isempty(L2)
    L2 = 1;
end
if nargin < 9
    L1 = 1;
end
if isempty(L1)
    L1 = 1;
end
if nargin < 8
    y0 = 0;
end
if isempty(y0)
    y0 = 0;
end
if nargin < 7
    x0 = 0;
end
if isempty(x0)
    x0 = 0;
end
if nargin < 6
    t0 = 0;
end
if isempty(t0)
    t0 = 0;
end
if nargin < 5
    K = 10;
end
if isempty(K)
    K = 10;
end
if nargin < 4
    k = 0.01;
end
if isempty(k)
    k = 0.01;
end
if nargin < 3
    g = @(x,y,t) 0.*x;
end
if isempty(g)
    g = @(x,y,t) 0.*x;
end
if nargin < 2
```

```
    error('Initial value (u0) and right-hand side (f) must be specified.')
end

% Initial assembly
[EToV, VX, VY] = GenerateMesh(x0,y0,L1,L2,NoElms1,NoElms2);
U{1} = u0(VX,VY);
[A, B] = AssembleMatrixTime(EToV,VX,VY);
[l1, l0] = AssembleLoadTime(EToV,VX,VY,f,t0,k);
bnodes = BoundaryNodesRectangle(VX,VY);
[A,B,l0,l1] = DirichletBCTime(bnodes, VX, VY, g, A, B, l0,l1, t0,k);

% Initialize time
t = t0;
u = U{1};
% Time step
for i = 1:K
    uNew = (B+k*theta*A)\((B-k*(1-theta)*A)*u+k*(theta*l1+(1-theta)*l0));

    % Impose boundary conditions on solution.
    for n = 1:length(bnodes)
        j = bnodes(n);
        uNew(j) = g(VX(j),VY(j),t);
    end
    t = t + k;
    U{i+1} = uNew;

    % Prepare for next iteration.
    l0 = l1;
    l1 = AssembleLoadTime(EToV,VX,VY,f,t,k);
    [¬,¬,¬,l1] = DirichletBCTime(bnodes, VX, VY, g, A, B, l0,l1, t,k);
    u = uNew;
end
end
```