University of Nottingham
UK | CHINA | MALAYSIA

FEM Algorithms in C++

G14DIS

Mathematics 4th Year Dissertation
2019/20

School of Mathematical Sciences, University of Nottingham

Adam Matthew Blakey

14286792

Supervisor: Prof Paul Houston

Project code: PH D2

Assessment type: Research-Informed Investigation

I have read and understood the School and University guidelines on plagiarism. I confirm that this work is my own, apart from the acknowledged references.

Abstract

This report investigates efficient algorithms used in approximating solutions to both linear and nonlinear differential equations with the finite element method.

The report begins by laying the analytical framework, setting up some model problems, and (for the linear case) proving existence and uniqueness of solutions to these model problems.

All major results in this report are produced using a bespoke software package, titled Blakey FEM, which applies a posteriori error estimates to certain classes of linear and nonlinear model problems; this application leads to efficient adaptivity strategies of both meshing and choice of interpolating functions.

This report found that there exist hp-adaptive strategies that yield exponential convergence rates between finite element solutions and true solutions for various model problems.

Acknowledgements

I am very grateful to my supervisor, Professor Paul Houston, for his invaluable guidance and advice throughout this project — and for introducing me to such a fascinating area of mathematics that I did not know existed!

A special thank you goes to Eleanor, Andy, Chris, and my mam, who have all helped through the vital proofreading process.

Digital Copy

Visit `https://github.com/JustAdamHere/G14DIS` for a copy of the code used for results in this report. You can also visit `https://r.blakey.family/BlakeyFEMPresentation` for a copy of the presentation slides given on 12th March 2020.

Contents

Section 1

Introduction

Differential equations are often used to explain and predict new facts about everything that changes continuously [13] such as weather prediction, planetary orbits, and the best way to design an aeroplane. It is vital, therefore, that we are able to compute solutions to these governing equations to some reasonable accuracy.

Many differential equations do not have analytical solutions, so one of two approaches is usually taken instead: solve a modified simpler equation to approximate the original solution, or approximate solutions directly [8, p. 260]; the latter is the focus of this report, specifically using numerical methods, as these are the principal choice for those approximating solutions of differential equations.

Various options for numerically approximating solutions to differential equations exist including finite difference methods [34], finite volume methods [26], and finite element methods [10] which is the method of choice for this report. Finite element methods are advantageous over other methods as they can express complicated geometries (for example the work of Giani et. al [14]) much more easily than finite difference methods [8, p. 746]. One can also derive reliable and efficient solution-independent error bounds, that locally give indications of the size of the error [42, p. 2733].

Finite element methods (FEMs) also permit high orders of convergence under the right conditions, through so-called p-refinement [27, p. 228]; however these high convergence rates can rely on certain amounts of regularity [10, p. 125]. There also exist h-adaptive techniques, where one can sequentially change the domain upon which the solution is approximated [41], but these techniques can only attain polynomial convergence rates at best [27, p. 228]. Therefore, in recent years, techniques concerned with a combination of the two (called hp-adaptive

methods) have become increasingly sought and applied effectively [42, p. 2731]. These hp-adaptive methods require two main ingredients: computable local error estimators, and steering criterion (presented in this report with a smoothness indicator) [22], which will be discussed in detail in Section 4.

The motivation for hp-adaptive FEM originates in the numerical solution of practical problems of physics or engineering, where one often encounters the difficulty that the overall accuracy of the numerical approximation is degraded by local singularities [39][p. 67]. By calculating local error estimators, one can subsequently use these to enrich the underlying approximation space in an adaptive manner [23, p. 2642].

This report found that h-adaptive algorithms could yield high (polynomial) convergence rates for some problems, and p-adaptive algorithms could yield high (exponential) convergence rates. Whilst these rates were often found to be more beneficial than their respective global refinement algorithms, the hp-adaptive algorithm appeared to perform the best at minimising both the error (in an appropriate norm) and the degrees of freedom.

Starting with a basic introduction in Section 2, we will introduce the common notations used in FEM analysis, as well as introducing some one-dimensional model problems which inform the numerical results in later sections. Section 3 discusses the specific implementation features of a bespoke software package written for this project, in particular highlighting the design choices made to benefit performance. We then derive local error estimators for a one-dimensional problem in Section 4, paving the way for the h-, p-, and hp-adaptive algorithms, as well as a comparison of their performance through some numerical experiments. Section 5 introduces a nonlinear problem from which we derive similar error estimators, and apply these to more numerical experiments. Concluding the report, Section 6 highlights the results of this project, what problems were encountered, and some suggestions for further work in the area.

Section 2

Background

In this section, the mathematical foundations for this report are laid out; these inform much of the implementation in Section 3 as well as the error analysis in Section 4.

2.1 FEM Notation

We will introduce some preliminary notions needed for understanding the mathematics to finite element methods.

2.1.1 L^p Spaces

In general, it does not make sense to ask what the length of a vector is in a vector space, but a norm is a concept designed to address this [11, p. 8]. There are many choices of norms in different vector spaces, as long as they satisfy some conditions.

Definition 2.1 (Norm, similar to [11, def. 1.3.1])**.**
We define a norm as a function $x \to \|x\|$ some vector space, Ω, to $\mathbb{R}$, provided that it satisfies the following conditions:

- $\|x\| = 0 \Rightarrow x = 0$

- $\|\lambda x\| = |\lambda|\|x\|, \forall x \in E, \forall \lambda \in \mathbb{C}$

- $\|x+y\| \leq \|x\| + \|y\|, \forall x, y \in \Omega$

We may also introduce the concept of an inner product, from which we may induce a norm.

Definition 2.2 (Inner product, similar to [11, def 3.2.1])**.**

We define an inner product on a vector space, V, as a mapping $(\cdot, \cdot) : V \times V \to \mathbb{C}$ that satisfies:

- $(x, y) = \overline{(y, x)}, \forall x, y \in V$

- $(\alpha x + \beta y, z) = \alpha(x, z) + \beta(y, z), \forall x, y, z \in V; \forall \alpha, \beta \in \mathbb{C}$

- $(x, x) \geq 0$*, with* $(x, x) = 0 \Leftrightarrow x = 0$

Note that we may induce a norm from the inner product by

$$\|x\|_V := \sqrt{(x, x)},$$

where $x \in V$.

For ease of notation, we also make the following two definitions:

Definition 2.3 (Range of integers)**.**

We define

$$[a, b]_{\mathbb{N}} := [a, b] \cap \mathbb{N},$$

where $[a, b] = \{x : a \leq x \wedge x \leq b, a, b \in \mathbb{R}\}$. We do this similarly for (a, b), $[a, b)$, and $(a, b]$ noting that we do allow slightly abusive notation with $b = \infty$.

Definition 2.4 (Inner product notation)**.**

We define the inner product, for $u, v \in V$ as

$$(u, v) := \int_{\Omega} uv \, dx.$$

L^p spaces, commonly referred to as Lebesgue spaces, refer to the space of functions where some integral of a function is finite, as shown in the following definition.

Definition 2.5 (L^p space, [9, p. 409])**.**

For u a complex-valued, locally integrable function, the L^p space is defined as:

$$L^2(\Omega) := \{u : \|u\|_{L^2(\Omega)} < \infty\},$$

with norm

$$\|u\|_{L^2(\Omega)} := \left(\int_\Omega |u(x)|^p dx\right)^{\frac{1}{p}}.$$

2.1.2 Weak Derivatives and FEMs

When dealing with finite element methods, we will often come across functions such as

$$u(x) = \begin{cases} x & \text{if } x > 0 \\ -x & \text{if } x \leq 0, \end{cases}$$

where the function may be continuous but the derivative may be discontinuous at a point. We therefore may introduce the concept of a weak derivative, as well as some other notation that we will use.

Definition 2.6 (Multi-index)**.**

If $\alpha = (\alpha_1, ..., \alpha_m)$ is an m-tuple where each $\alpha_i \in \mathbb{N}_0$, then we call α a multi index and denote x^α as the monomial

$$x^{\alpha_1}...x^{\alpha_m}$$

and similarly D^α as

$$D_1^{\alpha_1}...D_m^{\alpha_m}$$

where $D_i = \frac{\partial}{\partial x_j}$ [2, p. 1]. We also denote $|\alpha| = \alpha_1 + ... + \alpha_m$.

Definition 2.7 (Weak derivative)**.**

For $u \in V$, where $\int_\Omega |u|\,dx$, we define w as the αth weak derivative u provided that it satisfies

$$\int_\Omega u(x) D^\alpha v(x)\,dx = (-1)^{|\alpha|} \int_\Omega w(x) v(x)\,dx,$$

for all $v \in C_0^\infty(\Omega)$, where $C_0^\infty(\Omega)$ is the space of infinitely-differentiable functions with compact support on Ω and α defines a multi-index.

Sobolev spaces act as an extension to the L^p spaces with the inclusion of some regularity on weak derivatives of functions.

Definition 2.8 (Sobolev space, [33, p. 2])**.**
For $k \in [0, \infty)_\mathbb{N}$ and $\Omega \subseteq \mathbb{R}^m$, with Lipschitz-continuous boundary, we define:

$$H^k(\Omega) := \{u \in L^2(\Omega) : D^\alpha u \in L^2(\Omega), \forall |\alpha| \leq k\},$$

which is equipped with norm

$$\|u\|_{H^k(\Omega)} := \left(\sum_{|\alpha| \leq k} \|D^\alpha u\|^2_{L^2(\Omega)} \right)^{\frac{1}{2}},$$

and a semi-norm of

$$|u|_{H^k(\Omega)} := \left(\sum_{|\alpha| = k} \|D^\alpha u\|^2_{L^2(\Omega)} \right)^{\frac{1}{2}}.$$

We note that $H^k(\Omega)$ is a Hilbert space, and is often referred to as the one-dimensional Sobolev space of kth order on Ω.

The Cauchy-Schwarz inequality is very useful for the analysis of the a posteriori error bound in Section 4, so we define it here: For $u, v \in V$, V a vector space with some norm:

$$|(u, v)_V| \leq \|u\|_V \|v\|_V, \tag{2.1}$$

where $(\cdot, \cdot)_V$ is an inner product on V, and $\|\cdot\|_V$ is a norm on V.

We also have the triangle inequality: For $x, y \in V$, where V is a vector space with an inner product, we have

$$\|u + v\| \leq \|u\| + \|v\|, \tag{2.2}$$

as stated by Debnath et. al [11, co. 3.2.10].

We also state the following lemma, which follows directly from the definition of $\|\cdot\|_{H^1(\Omega)}$.

Lemma 2.1.

For any $u \in H^1(\Omega)$ we have

$$\|u\|_{L^2(\Omega)} \leq \|u\|_{H^1(\Omega)}.$$

We now have the mathematical tools to define what a finite element space describes. We state it here in its most general form, but in practical terms we will deal directly with these individual aspects of the finite element without mention of this formal definition.

Definition 2.9 (Finite element, [10, p. 78])**.**

A finite element in $\mathbb{R}^d$ is a triple $\mathcal{K} = (\Omega, P, \Sigma)$, where:

- *Ω is a closed subset of $\mathbb{R}^d$, int $\Omega \neq \emptyset$ and $\partial\Omega$ is Lipschitz-continuous;*
- *P is a space of functions from K to $\mathbb{R}$;*
- *$\Sigma := \{\phi_i\}_i^N$ is a finite set of linearly independent linear forms, ϕ_i, defined over P; it is also assumed that Σ is P-unisolvent. That is: $\forall \alpha_i \in \mathbb{R}, \exists! p \in P$ s.t. $\phi_i(p) = \alpha_i \forall i \in [1, N]_{\mathbb{N}}$.*

We call Σ the degrees of freedom (DoF).

Finite element methods rely heavily on the use of integration by parts; we will state the divergence theorem here, then state prove integration by parts.

Theorem 2.1 (Divergence theorem)**.**

For $\Omega \subseteq \mathbb{R}^n$ compact and $\partial\Omega$ piecewise smooth, u sufficiently differentiable in Ω, and n the

outward-pointing unit normal at each point on $\partial\Omega$ we have

$$\int_\Omega \nabla u \, dx \equiv \oint_{\partial\Omega} un \, dS.$$

From the divergence theorem, given in Theorem 2.1, we can now construct a proof of integration by parts.

Theorem 2.2 (Integration by parts)**.**

Taking Ω and n as in Theorem 2.1, we have

$$\int_\Omega u\nabla u \, dx \equiv \int_{\partial\Omega} uvn dS - \int_\Omega v\nabla u \, dx,$$

where u and v are sufficiently differentiable in Ω.

Proof. From Theorem 2.1 we may take $u \to uv$. This yields

$$\int_\Omega \nabla uv \, dx = \oint_{\partial\Omega} (uv)n \, dS$$

.

It is convenient to now write each vector in terms of its individual components, denoted by index i:

$$\int_\Omega \frac{\partial}{\partial x_i}(u_i v_i) \, dx = \oint_{\partial\Omega} (u_i v_i) n_i \, dS, \ \forall i \in [1, n]_{\mathbb{N}}.$$

By the product rule for differentiation we have

$$\int_\Omega \frac{\partial u_i}{\partial x_i} v_i \, dx + \int_\Omega u_i \frac{\partial v_i}{\partial x_i} \, dx = \oint_{\partial\Omega} (u_i v_i) n_i \, dS, \ \forall i \in [1, n]_{\mathbb{N}},$$

and returning to the original notation we have

$$\int_\Omega (\nabla u) v \, dx + \int_\Omega u(\nabla v) \, dx = \oint_{\partial\Omega} uvn \, dS.$$

After some rearrangement we have the result:

$$\int_\Omega u\nabla v\,dx = \int_{\partial\Omega} uvn\,dS - \int_\Omega v\nabla u\,dx.$$

□

2.2 Weak Solutions of PDEs in $\mathbb{R}^d$

For solving partial differential equations we ultimately want to find the solution, u, of the equation in some space, V. Mathematically we want to find $u \in V$ s.t.

$$\mathcal{L}u = f, \text{ in } \Omega \tag{2.3}$$

where $\mathcal{L}$ is some differential operator and f is some forcing function independent of u. We may instead consider a similar problem — which we refer to as the weak formulation — which is roughly constructed through the following steps:

1. Multiply Equation (2.3) by a test function, $v \in V$;
2. Integrate the resulting equation over the domain, Ω;
3. Apply integration by parts to reduce the highest order of derivation on u and v;
4. Apply appropriate boundary conditions to u and v.

This weak formulation will yield solutions for which not all derivatives may exist (or may have weak derivatives). We call these solutions weak solutions (opposed to strong solutions, which satisfy the criteria in Definition 2.7). A specific example of a weak formulation is calculated in the following section.

2.2.1 Model Problem

We restrict ourselves for the remainder of the report to consider only two model problems: a linear and a nonlinear problem. In this section, we will only state the linear PDE problem and derive its weak formulation, and then prove the existence and uniqueness of solutions. The nonlinear problem will be introduced in Section 5.

We consider the partial differential equation stated in Equation (2.4), i.e., given a bounded Lipschitz domain $\Omega \subseteq \mathbb{R}^d, d \geq 1$, we seek u such that

$$-\epsilon \Delta u + cu = f(x), x \in \Omega, \tag{2.4a}$$

$$u = 0, \text{ on } \partial\Omega. \tag{2.4b}$$

Here, $\epsilon > 0$ and $c \geq 0$ and f represent the reaction and forcing terms, respectively. This is a relatively standard model equation and similar problems are also chosen by Wihler [42], Houston et. al [21], and Mitchell et. al [27].

For this problem, we seek u in the function space $H_0^1(\Omega) =: V$; note that this imposes our boundary conditions, and assumes sufficient regularity of the solution, u.

To derive the weak formulation of Equation (2.4) we first multiply Equation (2.4a) by $v \in H_0^1(\Omega)$ and integrate over Ω, which yields:

$$-\epsilon \int_\Omega \Delta u v \, dx + \int_\Omega cuv \, dx = \int_\Omega fv \, dx, \; \forall v \in V.$$

We notice now that u has two derivatives (from the Laplacian) in the first term of this expression and v has zero. We can therefore apply integration by parts, as given in Theorem 2.2

to give

$$\epsilon \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} (\nabla u \cdot n) v \, ds + \int_\Omega cuv \, dx = \int_\Omega fv \, dx, \forall v \in V,$$

where n denotes the unit outward normal vector to the boundary, $\partial\Omega$. Noting that $v \in V$ and hence $v = 0$ on $\partial\Omega$ we get: find $u \in V$ such that

$$\epsilon \int_\Omega \nabla u \cdot \nabla v \, dx + \int_\Omega cuv \, dx = \int_\Omega fv \, dx, \forall v \in V.$$

Rewriting the above equation in inner product notation, the weak formulation of Equation (2.4) is given by: find $u \in V$ such that

$$\epsilon(\nabla u, \nabla v) + (cu, v) = (f, v), \ \forall v \in V. \tag{2.5}$$

Notice that this is still an infinite-dimensional problem which will be projected to a finite-dimensional space later.

We will see in Section 2.3 that this model problem has a solution that is unique.

2.2.2 Lax-Milgram

The Lax-Milgram theorem, named after the pair that solved it in 1954, is a pivotal theorem that guarantees existence and uniqueness of solutions to the problems described in Section 2.2.1.

Theorem 2.3 (Lax-Milgram, [11, p. 157])**.**
Let a be a bounded, coercive, bilinear functional on a Hilbert space, V. For every bounded linear functional l on V, there exists a unique $u \in V$ such that

$$a(u, v) = l(v), \ \forall v \in V.$$

We may note that to show boundedness, we can instead show that a and l are continuous. Therefore to satisfy Theorem 2.3 for a bilinear functional, a, and a linear functional, l, we just need to find $c_0, c_1, c_2 \in \mathbb{R}$ s.t.

- Coercivity of a: $a(u,u) \leq c_0 \|u\|^2_{H^1(\Omega)}$;
- Continuity of a: $|a(u,v)| \leq c_1 \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)}$;
- Continuity of l: $|l(v)| \leq c_2 \|v\|_{H^1(\Omega)}$.

We will now use the Lax-Milgram theorem to prove uniqueness and existence of the linear model problem outlined in Section 2.2.1.

Lemma 2.2 (Uniqueness and existence of linear model problem)**.**
Equation (2.4) *admits a solution that is unique.*

Proof. We begin by taking

$$a(u,v) := \epsilon(\nabla u, \nabla v) + (cu, v)$$

and

$$l(v) := (f, v)$$

from Equation (2.5), where $u, v \in H^1(\Omega)$.

From the definitions of a and l we see immediately that they are respectively bilinear and linear in their arguments.

To show coercivity of a, we see that $a(u,u) = \epsilon \|u\|_{L^2(\Omega)} + \|\sqrt{c}u\|_{L^2(\Omega)}$.

Let $c_s := \max_x(\sqrt{c(x)})$. Then:

$$\begin{aligned} a(u,u) &= \epsilon \|u\|^2_{L^2(\Omega)} + \left\|\sqrt{c}u\right\|^2_{L^2(\Omega)} \\ &\leq \epsilon \|u\|^2_{L^2(\Omega)} + c_s^2 \|u\|^2_{L^2(\Omega)} \\ &\leq c_0/2 \|u\|^2_{L^2(\Omega)} + c_0/2 \|u\|^2_{L^2(\Omega)}, \end{aligned}$$

where we have now defined $c_0 := 2\max(\epsilon, c_s^2)$. Noting Lemma 2.1, we have coercivity:

$$a(u,v) \leq c_0 \|u\|^2_{H^1(\Omega)}.$$

For continuity of a, let us recall that $a(u,v) \equiv \epsilon(\nabla u, \nabla v) + (cu, v)$.

By the triangle inequality from Equation (2.2) we have

$$\begin{aligned} |a(u,v)| &= |\epsilon(\nabla u, \nabla v) + (cu, v)| \\ &\leq |\epsilon(\nabla u, \nabla v)| + |(cu, v)| \\ &= \epsilon|(\nabla u, \nabla v)| + c_m|(cu, v)|, \end{aligned}$$

noting that $\epsilon > 0$ and $c_m := \max_x |c(x)|$.

By employing the Cauchy-Schwarz inequality from Equation (2.1) we get

$$|a(u,v)| \leq \epsilon \|\nabla u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} + c_m \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)}.$$

By letting $c_1 := \max_x(\epsilon, c_m)$ we have continuity:

$$\begin{aligned} |a(u,v)| &\leq \epsilon \|\nabla u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} + c_m \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \\ &\leq c_1 \|\nabla u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} + c_1 \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \\ &\leq c_1 \|\nabla u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} + c_1 \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \\ &\quad + c_1 \|\nabla u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} + c_1 \|u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} \\ &\leq c_1 (\|u\|^2_{L^2(\Omega)} + \|\nabla u\|^2_{L^2(\Omega)})^{\frac{1}{2}} (\|v\|^2_{L^2(\Omega)} + \|\nabla v\|^2_{L^2(\Omega)}) \\ &= c_1 \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)}. \end{aligned}$$

To show continuity of l, we first recall the definition of l as

$$l(v) \equiv (f, v).$$

Employing again the Cauchy-Schwarz inequality gives

$$\begin{aligned} |l(v)| &= |(f, v)| \\ &\leq \|f\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)}. \end{aligned}$$

Noting Lemma 2.1 gives

$$|l(v)| \leq \|f\|_{L^2(\Omega)} \|v\|_{H^1(\Omega)},$$

and defining $c_2 := \|f\|_{L^2(\Omega)}$ we have

$$|l(v)| \leq c_2 \|v\|_{H^1(\Omega)},$$

which shows that $l(v)$ is a continuous functional.

We have shown that the bilinear functional, a, is coercive and continuous in $H^1(\Omega)$, and we have shown that the linear functional, l, is continuous in $H^1(\Omega)$. By Theorem 2.3 we have that $\exists! u \in H^1(\Omega)$ s.t.

$$a(u, v) = l(v), \ \forall v \in H^1(\Omega),$$

which is the same as saying that Equation (2.4) admits a unique solution. □

2.3 hp-FEM

When setting up our FEM version of the problem, it is helpful to set it up in such a way that the size, shape, and degree of the polynomial interpolant on each element may vary independently of any other element. We make the following definitions.

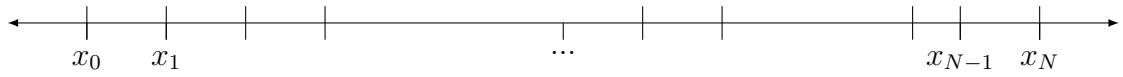
Definition 2.10 (Polynomial space)**.**

Akin to the definition in [10], $\mathcal{P}_P(\Omega) = \{u(x) : u$ *is a polynomial of degree* $r \leq P$ *on* $\Omega\}$*.*

We could have chosen, for example, B-splines or Fourier series to express our solutions, but we have chosen polynomials here.

Now we have a general definition of an FEM where we may freely choose the width of elements (h) and the polynomial degrees of the interpolants (p), and hence the name of hp-FEMs.

We create a mesh in one dimension which is the set of coordinates, $\{x_i\}_{i=0}^N$, that span Ω. We note that the coordinates do not have to be evenly spaced, but we do require that they are ordered $x_0 < ... < x_N$.



We take Equation 2.5, which describes an infinite-dimensional problem, and we now restrict to a finite-dimensional problem. For some set of basis functions, $\{\phi_j\}_{j=0}^M$ that lie in $\mathcal{P}_P(x_{i-1}, x_i)$ for some $i \in [1, N]$, we now define

$$V_h := \{u \in H^1(\Omega) : u|_{(x_{i-1},x_i)} \in \mathcal{P}_P(x_{i-1}, x_i), i \in [1, N]_{\mathbb{N}}\},$$

so that the solution restricted to each element is a polynomial. Our new problem, which we refer to as the finite element approximation, is: find $u_{h,p} \in V_h$ such that

$$\epsilon(\nabla u_{h,p}, \nabla v_h) + (cu_{h,p}, v_h) = (f, v_h),\ \forall v_h \in V_h. \tag{2.6}$$

We stress here that this is now a finite-dimensional problem, which can be approximated computationally.

Section 3

Implementation: Blakey FEM

Many FEM solvers already exist such as Autodesk Simulation, FEFLOW, Deal II, FEniCS, and Goma; these software packages are written in a range of different languages, all set up to solve slightly different problems, making use of specific language features. However many of these software packages fail to make use of hp-adaptive methods, which can lead to very high orders of convergence if used correctly [42].

Our implementation is called Blakey FEM and is written in C++. C++ is low-level and gives the programmer control of memory management, which allows for more efficient algorithms [24]. It is also an object-oriented language: the object-oriented paradigm is convenient for us humans, as its primary focus is to define the structure of how the data will be organised and is often likened to real life examples. The intuitiveness of this organisation would allow a dog class, say, to have a method `Dog.bark()`. This intuitiveness is also often extended through a technique called abstraction, which allows other programmers or users to deal only with interfaces of classes, and not the specific implementation; this may mean to say that we don't really care how a dog barks, as long as it barks.

In real life we may encounter objects that are very similar and share some common logic; the object-oriented paradigm gives us a mechanism called inheritance which allows for some methods and data to be written in the super-class and be reused in the sub-classes. This can be best described by a specific example. Inspired from [29], we may make an inheritance diagram of structures that we may encounter in our everyday lives; this idea then extends to how we may construct classes in an object-oriented language. Take Figure 3.1, for example, which shows an inheritance diagram of some animals along with some methods. Notice that the middle portions of the nodes indicate structural properties (including data), and the bottom

portions indicate methods (manipulators to the data).

We see from Figure 3.1 that all derived classes of `Animal` (i.e. `Mammal`, `Cat`, `Reptile`, `Human`, and `Snake`) all share, among others, the property `hungerLevel` (so every animal can be hungry). However it only makes sense for `Cat` to have the method `meow()`, for obvious cat-related reasons.

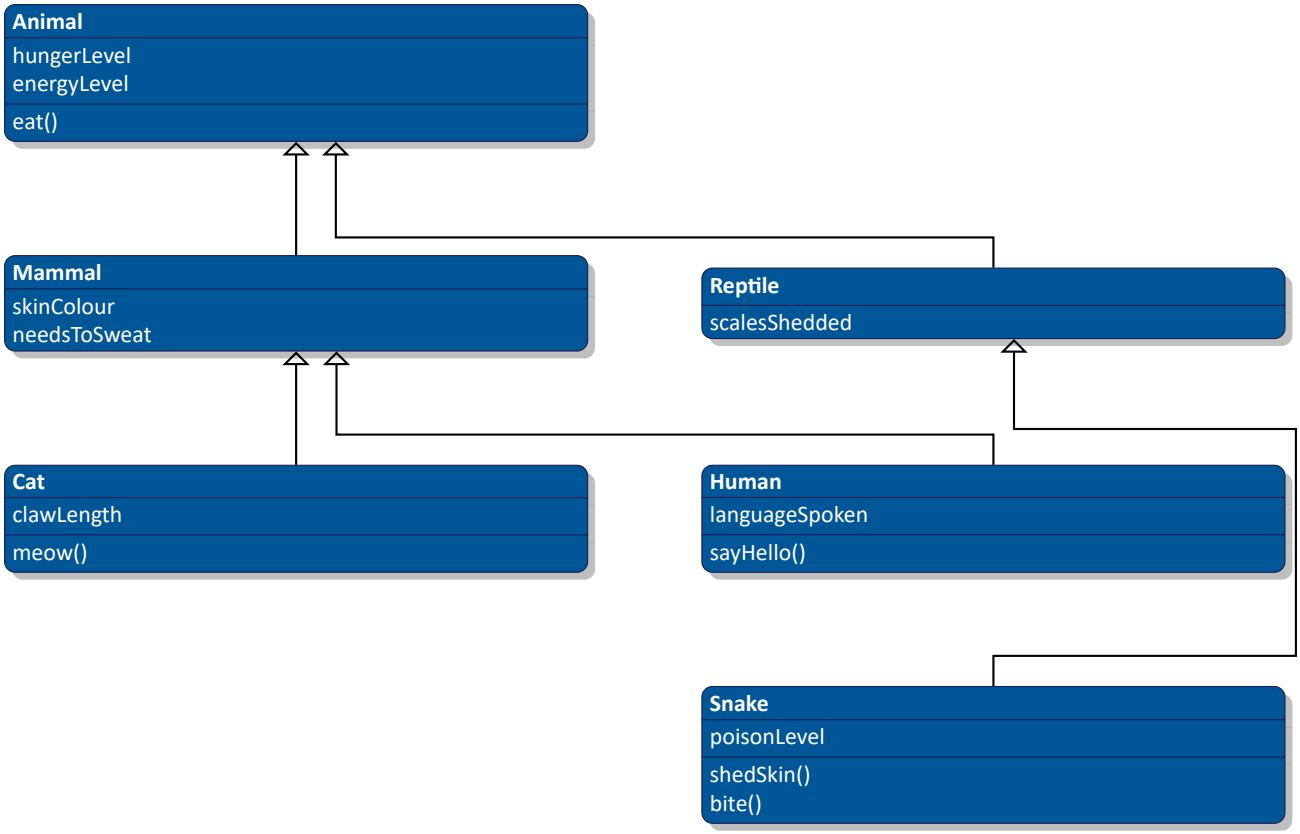


Figure 3.1: *An example of an inheritance diagram with different animals and categories of animals.*

The code structure and implementation choices of Blakey FEM are heavily influenced by [33], which details choices for all aspects needed to be considered in an FEM program. The following sections describe the choices made for implementing the code including details regarding linear solvers, quadrature calculation, and the specific object-oriented design structure. We note, however, that Blakey FEM is currently a one-dimensional FEM package — but design choices have been made that would support easy extensibility to higher dimensions.

3.1 Meshes

A mesh describes the discretised version of the domain upon which we are solving and — depending upon how the domain is discretised — can actually have a huge impact upon on our numerical approximation of the solution (as demonstrated in the work of Wilbraham [43], but now commonly referred to as Gibbs phenomenon).

With our implementation we decided to make a class `Mesh` which basically acts as a container for the elements, for which a class structure is displayed in Figure 3.2. The declarations for this class can be found at `./src/mesh.hpp` and the definitions can be found at `./src/mesh.cpp`, through the GitHub repository give on Page 2.

Figure 3.2: *Class structure for Mesh.*

There are two constructors: `Mesh(int a_noElements)` and `Mesh(Elements a_elements)`. The former constructs a mesh on $[0, 1]$ with elements of equal width; and the latter constructs a mesh with the provided elements, which the user should populate manually for more control over the mesh.

The `Elements* elements` property is a pointer to an instance of the `Elements` class described in the next section. These are populated either through the first constructor or referenced by the second constructor.

The other properties are relatively self-explanatory, with the relevant getters: `int noElements`

is the number of elements in the mesh, `int noNodes` is the number of nodes in the mesh, and `int dimProblem` is the dimension of the problem. We note that the dimension of the problem for the implementation in this report is fixed as 1; however, due to the object-oriented techniques and careful design of the implementation, it would be relatively easy to implement meshes on higher dimensions — and this parameter in the class would help to facilitate this.

3.2 Polynomial Spaces

With the hp-FEMs that we are implementing, we have chosen the shape functions as polynomials — and we need to be able to describe these polynomial basis functions in general for any order exponent and for any order derivative. We could have also chosen functions besides polynomials, such as Fourier shape functions [19] or B-splines (although technically polynomials, they don't behave in the same way with degrees of freedom) [16, 17]; however we have chosen polynomials due to their high convergence rates and resulting reduction in the number of degrees of freedom [17, p. 1].

As well as the choice of the type of functions, we now have a further choice to make: what kinds of polynomials we want, remembering that a certain amount of regularity will already be imposed, depending upon the space in which we are seeking solutions. An obvious choice of polynomials would be Lagrange or Legendre polynomials, but we have instead chosen to use Lobatto shape functions, employed in [33, p. 25] and [40, p. 48]; this choice of basis functions gives us a hierarchical basis set.

We begin by introducing the Legendre polynomials in one dimension as given by

$$L_0(x) = 1, \tag{3.1a}$$

$$L_1(x) = x, \tag{3.1b}$$

$$L_n(x) = \frac{2n-1}{n} x L_{n-1}(x) - \frac{n-1}{n} L_{n-2}(x),\ n \geq 2, \tag{3.1c}$$

cf. [33, p. 22]. We can also define the Lobatto shape functions by

$$l_0(x) = \frac{1-x}{2}, \tag{3.2a}$$

$$l_1(x) = \frac{1+x}{2}, \tag{3.2b}$$

$$l_n(x) = \sqrt{n-1/2}\int_{-1}^{x} L_{n-1}(\xi)\,d\xi,\ n \geq 2, \tag{3.2c}$$

cf. [33, p. 25]. We notice that Legendre polynomials are orthogonal, meaning that we have $\int_{-1}^{1} L_n(x)\,dx = 0, n \geq 1$. This helpfully means that $l_n(1) = 0, \forall n \geq 2$. We also notice by definition that $l_n(-1) = 0, \forall n \geq 2$, so the Lobatto shape functions vanish at both sides of the domain for $n \geq 2$.

We have chosen to implement these Lobatto shape functions as our basis functions within the `Element` class, which is described more in Section 3.3.

We note that these functions are defined on $[-1, 1]$, which will be the domain for our one-dimensional reference element. For actual implementation, we can define the Legendre polynomials relatively easily with the recursive formula given in Equation (3.1), and we do so within the `quadrature` namespace (mainly for ease when defining the Gauss-Legendre quadrature). However, the definitions of the Lobatto functions are a little more tricky because of the integral that appears in the definition. With this in mind we state the following result.

Lemma 3.1.

The nth, $n \geq 2$, Lobatto shape function can be written as

$$l_n(x) \equiv \sqrt{\frac{2n-1}{2}}\left(L_{n+1}(x) - L_{n-1}(x)\right).$$

Proof. By [33, eq. 1.43], we know that

$$L_n(\xi) \equiv \frac{\mathsf{d}}{\mathsf{d}\xi}\Big(L_{n+1}(\xi) - L_{n-1}(\xi)\Big).$$

Integrating this on $[-1, x]$ gives

$$\begin{aligned}\int_{-1}^{x} L_n(\xi)\, d\xi &= \Big(L_{n+1}(x) - L_{n-1}(x)\Big) - \Big(L_{n+1}(-1) - L_{n-1}(-1)\Big) \\ &= \Big(L_{n+1}(x) - L_{n-1}(x)\Big) - \Big((-1)^{n+1} - (-1)^{n-1}\Big) \\ &= \Big(L_{n+1}(x) - L_{n-1}(x)\Big),\end{aligned}$$

which, by our definition in Equation (3.2), is simply

$$l_n(x) = \sqrt{n - 1/2}\Big(L_{n+1}(x) - L_{n-1}(x)\Big).$$

Rearranging gives the desired result. □

From Equation (3.1) we can calculate Legendre polynomials generally, but not their derivatives. The derivatives for $n = 0, 1$ are straightforward, but not for $n \geq 2$. Hence, we need to determine how to compute the derivative of the Legendre polynomials for any order; this is required for the proceeding hp-adaptivity algorithm. By following a proof by induction from Lemma 3.2 we can ultimately construct such a method.

Lemma 3.2.

The kth, $k \geq 1$, derivative of a one-dimensional Legendre polynomial of the nth, $n \geq 2$, order is defined by

$$\frac{\mathsf{d}^k}{\mathsf{d}x^k} L_n(x) = \frac{2n-1}{n-1} x L_{n-1}^{(k)}(x) - \frac{n}{n-1} L_{n-2}^{(k)}(x) + (k-1)\frac{2n-1}{n-1} L_{n-1}^{(k-1)}(x).$$

Proof. We first test for the base case ($k = 1$). From this, we get

$$L'_n(x) = \frac{2n-1}{n-1} x L'_{n-1}(x) - \frac{n}{n-1} L'_{n-2}(x),$$

which is true by differentiating and rearranging Equation (3.1c) and employing

$$L_n(x) = \frac{\mathsf{d}}{\mathsf{d}x}\Big(L_{n+1}(x) - L_{n-1}(x)\Big),$$

cf. the proof of Lemma 3.1 (and [33, eq. 1.43]).

Let's now assume that Lemma 3.2 holds for some $k \geq 2$, which is just

$$\frac{\mathsf{d}^k}{\mathsf{d}x^k} L_n(x) = \frac{2n-1}{n-1} x L^{(k)}_{n-1}(x) - \frac{n}{n-1} L^{(k)}_{n-2}(x) + (k-1)\frac{2n-1}{n-1} L^{(k-1)}_{n-1}(x).$$

Differentiating this again gives

$$\frac{\mathsf{d}^{k+1}}{\mathsf{d}x^{k+1}} L_n(x) = \frac{2n-1}{n-1} \frac{\mathsf{d}}{\mathsf{d}x}\Big(x L^{(k)}_{n-1}(x)\Big) - \frac{n}{n-1} L^{(k+1)}_{n-2}(x) + (k-1)\frac{2n-1}{n-1} L^{(k)}_{n-1}(x).$$

By the product rule we may expand the differentiation into the first RHS term, giving

$$\begin{aligned}\frac{\mathsf{d}^{k+1}}{\mathsf{d}x^{k+1}} L_n(x) &= \frac{2n-1}{n-1}\Big(L^{(k)}_{n-1}(x) + x L^{(k+1)}_{n-1}(x)\Big) - \frac{n}{n-1} L^{(k+1)}_{n-2}(x) + (k-1)\frac{2n-1}{n-1} L^{(k)}_{n-1}(x) \\ &= \frac{2n-1}{n-1} x L^{(k+1)}_{n-1}(x) - \frac{n}{n-1} L^{(k+1)}_{n-2}(x) + k\frac{2n-1}{n-1} L^{(k)}_{n-1}(x),\end{aligned}$$

which gives the desired result.

By the principles of mathematical induction, we have shown that the lemma is true for the base case and inductive case, and so we conclude that the lemma is true for all $k \geq 1$. □

Since the Lobatto shape functions are calculated by a linear combination of Legendre polynomials (by Lemma 3.1), we now have a method for calculating our basis functions of any order and any derivation. The method `f_double Element:basisFunction(int n, int i)` in the `Element` class calculates any derivative of any order basis function, making use of the `f_double quadrature::legendrePolynomial(int n, int i)` method in the `quadrature` namespace. Note that the Legendre polynomials and their derivatives calculated here are one dimensional.

3.3 Elements

The element class was originally designed to be an abstract type, where children classes could take various forms in various dimensions (for example intervals in 1D, or triangles or squares in 2D, or tetrahedra or tetrahedra in 3D). Since Blakey FEM has been designed to solve 1D problems only, we have instead decided to make our element class a concrete class that implements only intervals.

As well as creating a class named `Element` we also decided to create a class called `Elements` which is essentially a wrapper for many `Element` instances. Both of these classes are described in Figure 3.3.

There are two constructors for this class: `Element(Element element)` which is a copy constructor (and has same logic as the equals operator), and `Element(int elementNo, int noNodes, vector<int> nodeIndices, vector<double>* nodeCoordinates, int polynomialDegree)` which provides the class with an element number, the number of nodes for the element (although in 1D this will be fixed to 2), the indices of the nodes, a pointer to the node coordinates vector, and the polynomial degree for this element.

The destructor of `Element` does not do anything special: but it certainly does not delete the storage at nodeCoordinates, which belongs to the `Elements` container. N.b. the `Element` and `Elements` classes actually perform very different functions in this implementation.

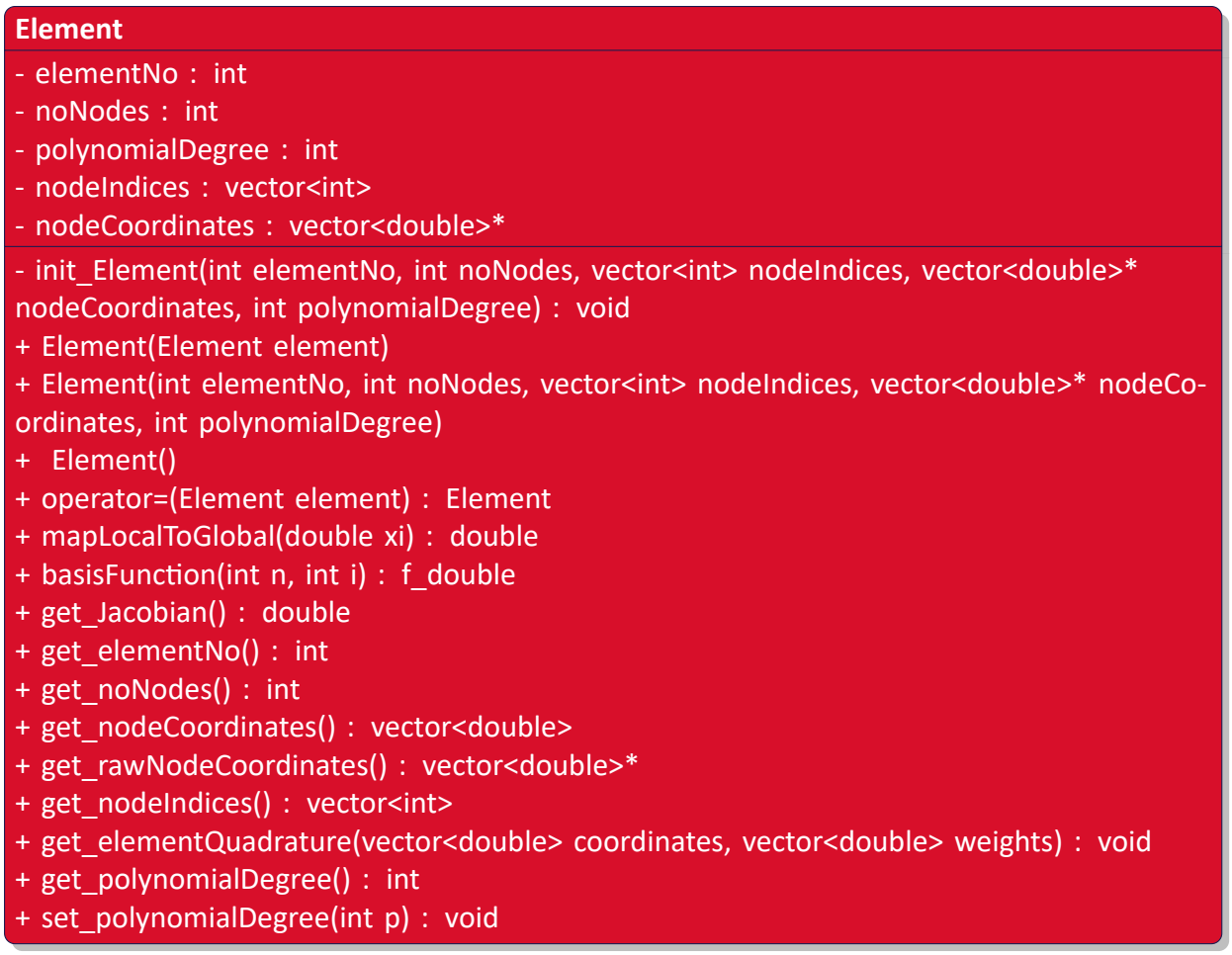


Figure 3.3: *Class structure for Element.*

The method `double mapLocalToGlobal(double xi)` takes a point on the local domain (on $[-1, 1]$) and calculates where that point corresponds to on the global domain; in 1D this is just a simple linear mapping $f : [-1, 1] \to [x_{i-1}, x_i]$, where x_{i-1} and x_i are the node coordinates.

The `basisFunction` method, as discussed in Section 3.2, calculates the basis function on the current element for any given degree and derivative. We note that the basis functions remain on the reference element $[-1, 1]$.

3.4 Linear Solvers

Linear systems arise in various different areas, and are of particular importance computationally thanks to the methods that exist to approximate solutions to the systems. Directly calculating an inverse to an $N \times N$ matrix can take $\mathcal{O}(N^3)$ operations, so for large linear systems this could take a very long time. In practice we don't need to find the explicit inverse and can go straight to seeking a solution to the system provided we have a right-hand-side; however other computational algorithms such as Gaussian elimination also take $\mathcal{O}(N^3)$ operations [8, p. 368]. For diagonal systems we can actually find a solution in $\mathcal{O}(N)$ when we are dealing with linear elements for FEMs in 1D, but higher-order FEMs no longer result in diagonal matrices.

We can therefore turn to iterative techniques, which can provide solutions to some given accuracy. For large sparse systems (like the system resulting from our FEM calculations) the conjugate gradient is generally a well-favoured method [8, p. 479]. We have therefore chosen to use a conjugate gradient solver with some tolerance (usually set to 1×10^{-15}) for solving all linear systems that arise and is defined in `vector<double>` `linearSystems::conjugateGradient(Matrix<double> M, vector<double> b, double tolerance)`. We note that we could also make use of the Thomas algorithm for solving diagonal systems arising from 1D linear FEM calculations; however we decided that the performance boost for using a separate algorithm for solving a linear system for perhaps only the first few iterations was seen as too small to justify proper implementation. We have however implemented the Thomas algorithm in `void linearSystems::` `thomasInvert(vector<double> lower, vector<double> diagonal vector<double> upper, vector<double> load, vector<double> solution)` in case this wanted to be developed further in the future. The conjugate gradient algorithm is shown in Equations (3.3), as defined in [6, p. 1605], where we choose $\boldsymbol{d}_0 = \boldsymbol{r}_0 = \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b}$. We note that each $\boldsymbol{x}_k$ is our approximation of the solution, and the terminating condition for the

algorithm is when $\boldsymbol{r}_k \cdot \boldsymbol{r}_k$ is below some given tolerance (usually 1×10^{-15}).

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k \tag{3.3a}$$

$$\alpha_k = -\frac{\boldsymbol{r}_k^\mathsf{T} \boldsymbol{d}_k}{\boldsymbol{d}_k^\mathsf{T} \boldsymbol{A} \boldsymbol{d}_k} \tag{3.3b}$$

$$\boldsymbol{d}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{d}_k \tag{3.3c}$$

$$\beta_k = -\frac{\boldsymbol{r}_{k+1}^\mathsf{T} \boldsymbol{A} \boldsymbol{d}_k}{\boldsymbol{d}_k^\mathsf{T} \boldsymbol{A} \boldsymbol{d}_k} \tag{3.3d}$$

We have implemented the conjugate gradient method described above as done in [20], but we note that we could have computationally implemented a better algorithm: one that permits parallelisation. Computers in recent years have, for one reason or another, been geared to having more cores than a faster processing clock speed [37]; however the standard conjugate gradient algorithm is not suitable for these multi-core processors as each new direction ($\boldsymbol{d}_k$) requires the new residue ($\boldsymbol{r}_k$) to have been calculated [6, p. 1605]. We therefore could have used the cooperative conjugate gradient method provided by Bhaya et. al to make use of the many cores and threads in a central processing unit, but for the purposes of this project we will just use the regular conjugate gradient method as the 1D simulations aren't too computationally demanding.

We note that to solve our linear systems above we need a `Matrix` data structure, as matrices are not built-in to the programming language — this is due to use cases for matrices widely differing from programmer-to-programmer. We have therefore implemented our own class hierarchy to store matrix details relevant to our problems, as shown in Figure 3.4. The `Matrix` class is itself abstract, and its descendant classes implement many of its method. We have implemented a child class called `Matrix_full` which stores the matrix elements in a single vector, `items`, whereby the index at which an element value is stored is calculated relatively easily from two coordinate values; this is akin to the approach taken by the creator of C++,

Bjarne Stroustrup, in his comprehensive guide to the language [35, p. 831] to minimise storage required.

Now that we have this generic structure for our matrix classes, it would be relatively easy to add a sparse matrix data structure, say `Matrix_sparse`, with a method like compressed sparse row format [31, p. 93]. For large sparse matrices, this method aims to reduce the total amount of storage needed in computer memory.

We also notice that the Matrix class and its descendants are implemented generally for a type, `T`, using C++'s templates feature. We have chosen to do this for several reasons: firstly, this allows a single implementation of a matrix for any given type — for example, a matrix of type `double` or `int`; secondly, this approach reduces redundancy of code and allows features for all types of matrix to be added with relative ease; and thirdly, this approach saves on both runtime and space efficiency [35, p. 665].

3.5 Nonlinear Solvers

Later in the report, in Section 5, we introduce a nonlinear model problem (rather than a linear problem). We need a nonlinear solver to solve the resulting nonlinear system. Although not the primary concern for this stage in the report, it's important that we cover the implementation side of this problem. We have chosen to implement a Newton solver for this solving process, thanks to its quadratic convergence rates close to roots [36, p. 119].

By explicitly calculating the necessary function and derivative needed for our nonlinear problem, we use the Newton's method in `void Solution_nonlinear::Solve_single(double cgTolerance, vector<double> uPrev, vector<double> uNext, double difference)`. This performs one Newton step, which we can run multiple times in the `void Solution_nonlinear::Solve(double cgTolerance, double NewtonTolerance, vector<double> u0)` to find a root within a certain specified tolerance.

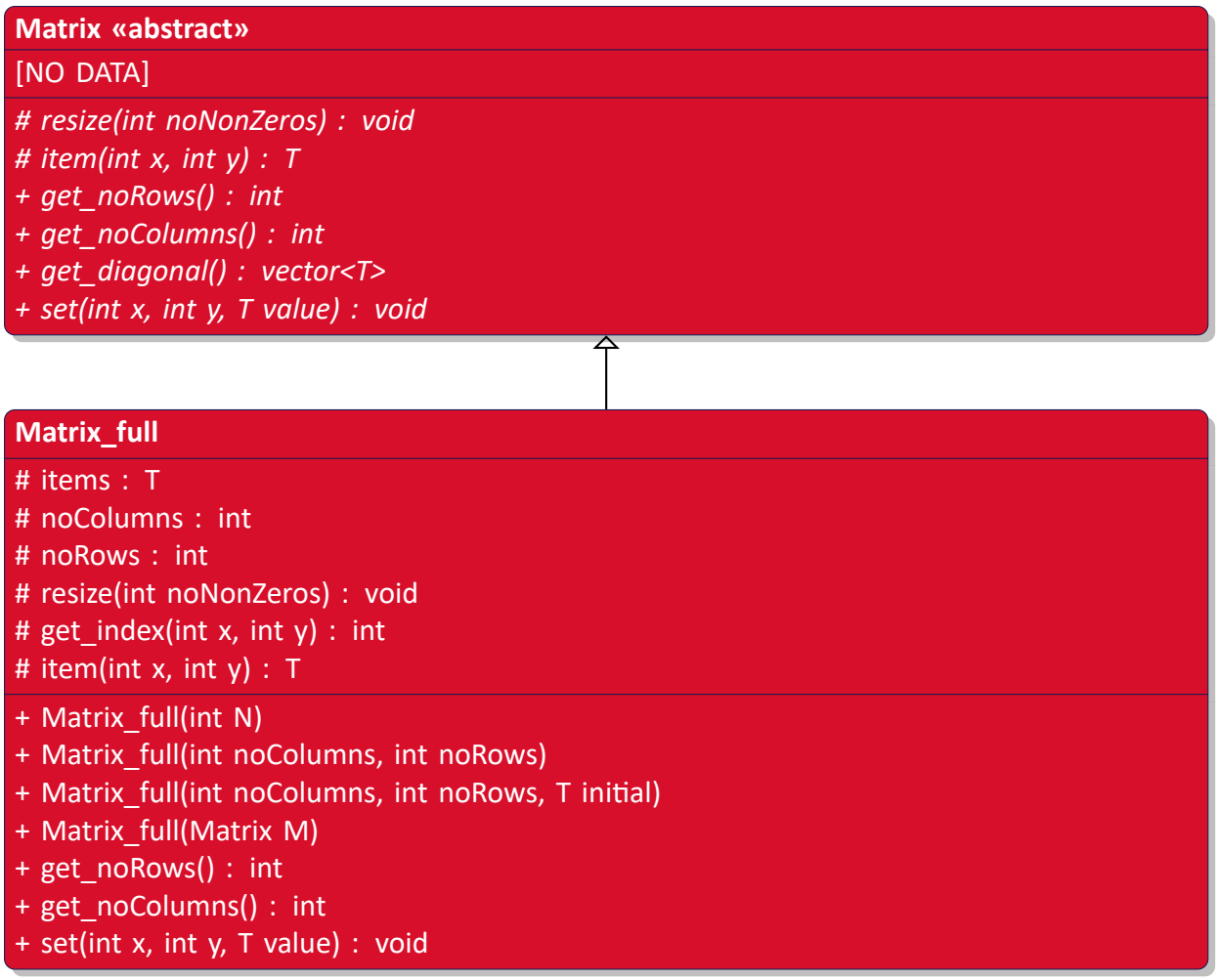


Figure 3.4: *Class structure for Matrix and its descendants.*

3.6 Quadrature

All FEMs will need to (at some point at least) find the value of the one-dimensional definite integral

$$I(f) = \int_a^b f(x)\,dx.$$

We are interested in finding numerical methods that yield accurate approximations to I [12]. We call the numerical approximation to a definite integral a quadrature method, and we will in particular consider quadratures of the form

$$I_n(f) = \sum_{i=1}^{n} w_i f(x_i),$$

where we call w_i the quadrature weights and x_i the quadrature points for $i \in [1, n]_{\mathbb{N}}$.

Gaussian quadrature gives us the best choices of weights and points for approximating the integration of a function of one variable numerically on an interval [30]; Gaussian quadrature can also be derived for integration of functions in more than one dimension (useful in multi-dimensional FEMs), but we will restrict ourselves to considering one dimension.

We define Gaussian quadrature of order n, $n \geq 1$, as done in [33], and more universally known as Gauss-Legendre quadrature. By taking L_n to mean the nth order Legendre polynomial as defined in [33, p. 22] (c.f. Section 3.2), and noticing that L_n has n zeros, we define the ith Gaussian weights and points given by:

$$\xi_{n,i} \text{ s.t. } L_n(\xi_{n,i}) = 0, \tag{3.4a}$$

$$w_{n,i} = \frac{2}{(1-\xi_{n,i}^2)L_n'(\xi)^2},\ i \in [1,n]_{\mathbb{N}}. \tag{3.4b}$$

To allow for exact integration of polynomials of a chosen degree, we need to be able to calculate these points and weights for any given n. Therefore, Blakey FEM implements:

- `f_double quadrature::legendrePolynomial(int n, int i)` — returns a function pointer to the ith derivative of the nth-degree Legendre polynomial;
- `void quadrature::legendrePolynomialRoots(int n, vector<double> roots)` — populates `roots` with the n roots of the nth degree Legendre polynomial by a Newton method within a residual tolerance of 10^{-5};
- `double quadrature::get_gaussLegendrePoint(int n, int i)` — returns the ith Gaussian point of nth order;
- `double quadrature::get_gaussLegendreWeight(int n, int i)` — returns the ith Gaussian weight of nth order.

Since the root finding of the zeros of the Legendre polynomials are relatively expensive, we also have an intelligent cache built-in to the code so that no point or weight is generated more

than once — when calculated for the first time they are stored in a dictionary data structure. Whilst this has an immediate computational penalty with the lookup of various values in the dictionary, it is a much lower cost than computing the points and weights. It is so vital in decreasing the runtime of the program because the roots of these polynomials are found using a Newton method, which may take a large number of iterations to converge.

3.7 Object-Oriented Design

Figure 3.5 highlights the main class diagram of this implementation, in particular highlighting the structure rather than the specific syntax usage — we have therefore omitted arguments and their types, as well as const-ness, as these don't inform the structure of our code too much.

Note that we show inheritance with open triangle-headed arrows, one-to-one association with closed triangle-headed arrows, and one-to-many association with open diamond-headed arrows (in accordance with industry standards [1]). We denote private members with '#', private members with '-', and public members with '+'. These access attributes give our code some protection to illegal usage, whereby a user may only interact with our classes with public members. We also denote abstract methods with italics, and make a note of abstract classes next to their class name.

To calculate a solution to a problem, the user needs only to instantiate instances of `Mesh` and `Solution` (with their relevant arguments), and all instances of other classes are created within these if necessary.

During the creation an instance of `Mesh`, they can either provide an `Elements` instance with various `Element` s pre-populated, or they may simply provide a number of equally-spaced elements they want in that `Mesh`. This `Mesh` can then be then be passed to the `Solution` constructor, with other problem details. To calculate a finite element solution, the user may call the `Solution::Solve` method, from which `Solution::output_solution()` may be called to output the solution to a data file. This data file can then be used to plot the values of the

finite element solution again the x-axis.

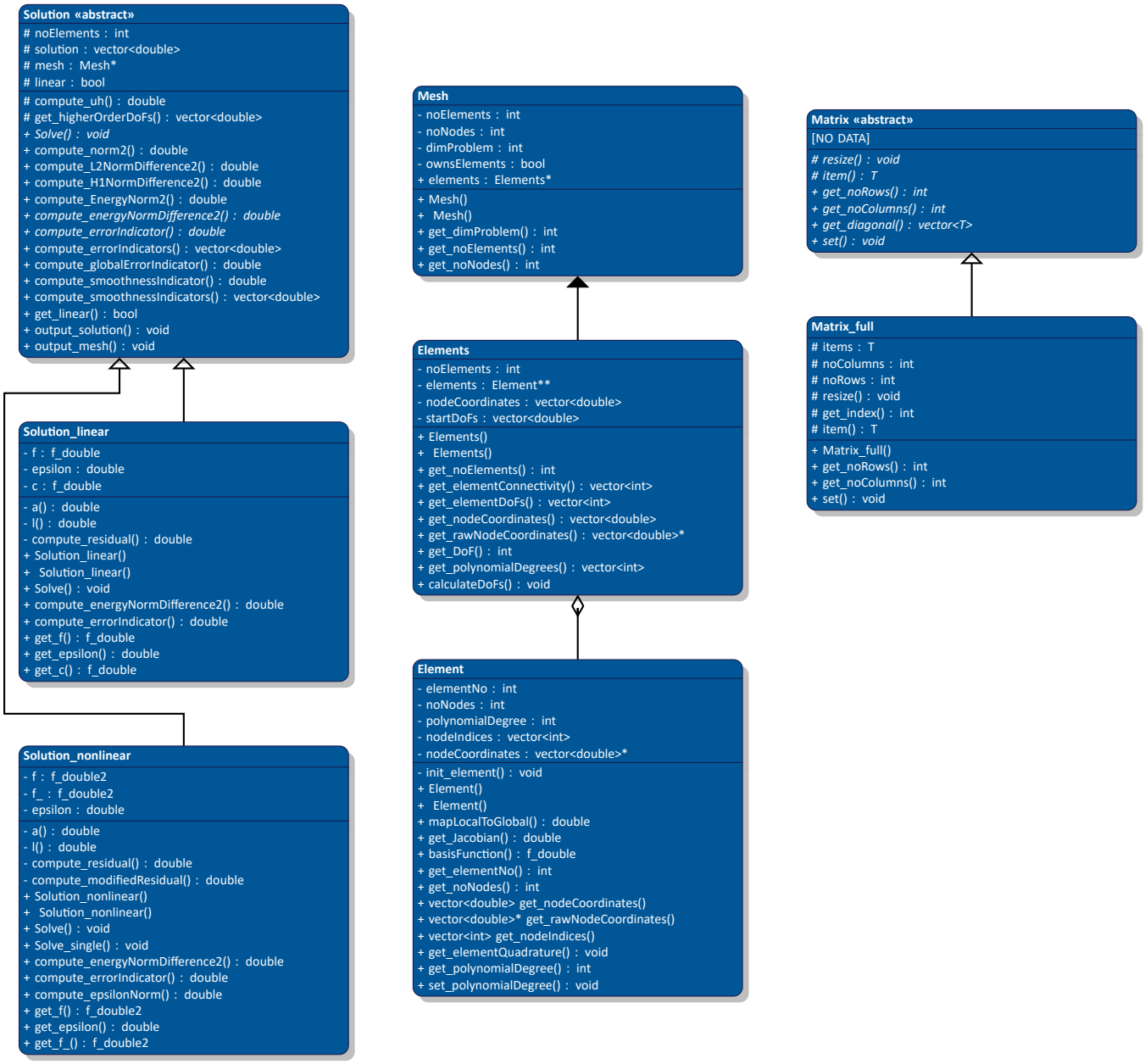


Figure 3.5: *The rough object-oriented design of Blakey FEM.*

Figure 3.6 outlines the structure of the namespaces used in this implementation, which again omit the arguments for simplicity. In general, the namespace names correlate to their purpose.

The `common` namespace holds some methods that are included in every implementation file; these methods are methods that need to be used frequently across different classes and namespaces, so it made sense for them to be defined once in their own separate namespace.

Methods concerned with refinement are defined in the `refinement` namespace. In general,

one would make a call to either `refinement::refinement()` or `refinement::refinement_g()`, which respectively describe refinement and global refinement. The choice of refinement method at each step is chosen by the choice of flags in the arguments for these methods. The `refinement::refine_hp()`, `refinement::refine_h()`, and `refinement::refine_p()` respectively define individual steps of the hp-, h-, and p-adaptive refinement processes, and will usually only be called from the former refinement processes. The namespace is structured such that one of the individual refinement steps will take an old `Solution` and old `Mesh`, and give a new, refined `Solution` and new, refined `Mesh`. More of the implementation details for these processes are given later in Algorithms 4.1–4.4.

The implementations of `linearSystems` and `quadrature` are discussed in Sections 3.4 and 3.6, respectively.

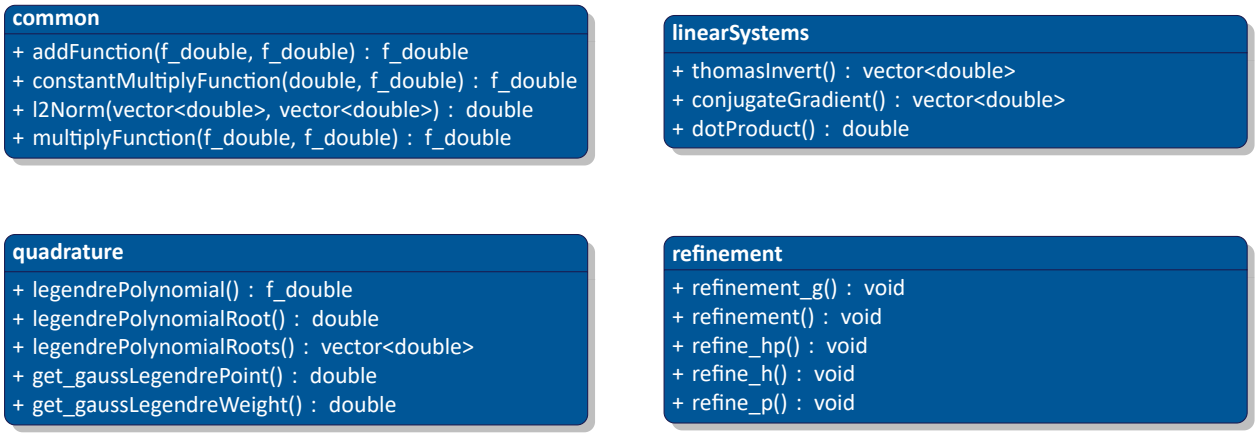


Figure 3.6: *The namespaces available in Blakey FEM.*

We note here that the base class `Solution` and its descendants, `Solution_linear` and `Solution_nonlinear`, are at the heart of the solving process of the finite element algorithms. See Appendix A for a more detailed discussion of how these classes use the data to solve the problems.

3.8 Simple Numerics

Now that we have an implementation of a finite element solver, we may test it with a few example problems and see what results we have. Let us first consider a boundary value problem

on $\Omega = [0, 1]$, where we seek a solution, $u \in H^1(\Omega)$, such that

$$-0.001u'' + u = 1,$$

where $u(0) = u(1) = 0$. This fits our model problem from Section 2.2.1 by setting $\epsilon = 0.001$, $f \equiv 1$, and $c \equiv 1$.

We first solve some simple examples on this domain for equally-sized linear elements across the entire domain, as shown in Figure 3.7.

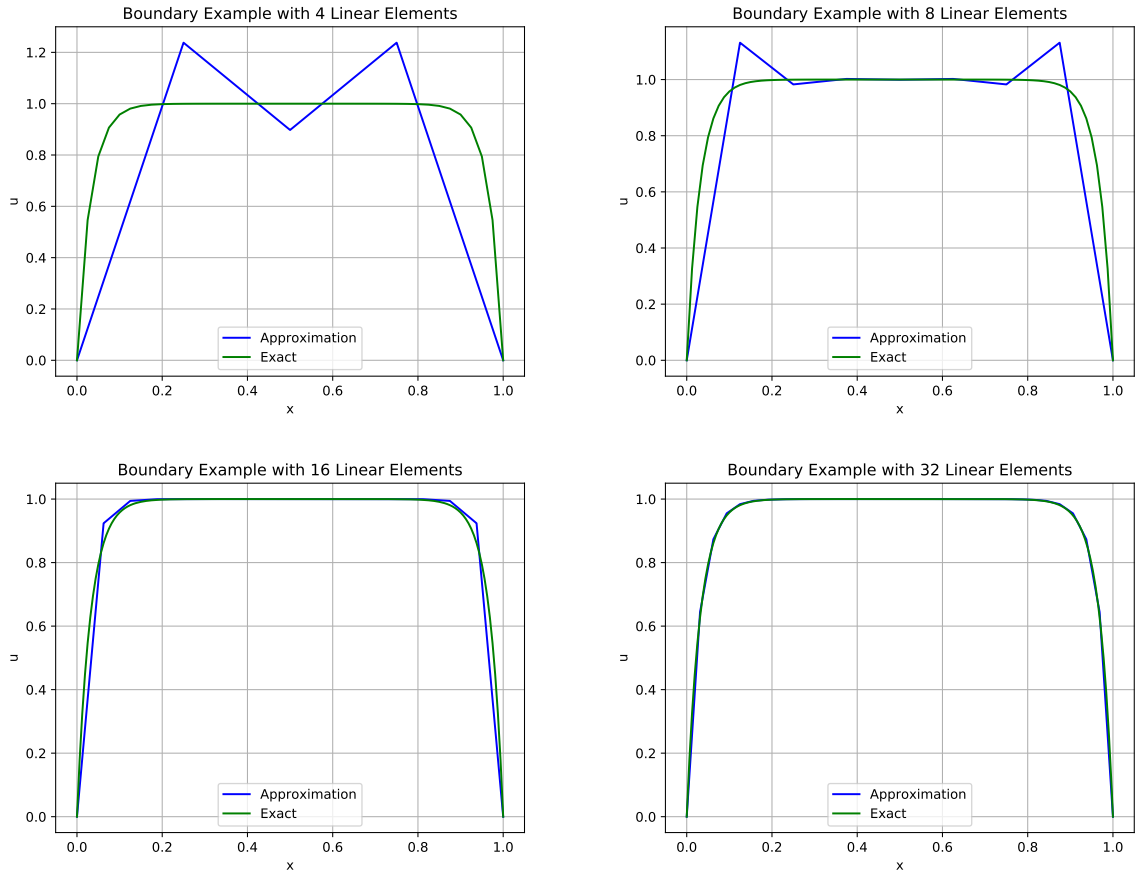


Figure 3.7: *Plots for the example boundary layer problem for varying element sizes.*

We see that, as the number of elements increases, the solution visibly gets more accurate. But there is more to this than that: we notice that with 8 linear elements, there is an undesirable overshoot in the approximation near the boundaries, but the plateau in the centre of the domain is already a good approximation of the solution. As we increase the number of elements we use for the approximation, we are increasing the elements across the plateau for

seemingly no reason. It may be beneficial, for example, to only reduce the size of elements in areas where the higher resolution is needed. We will see in Section 4 that we can compute indicators capable of telling us exactly where these regions are, and allow us to get a better approximation with fewer elements.

We could instead increase the polynomial degree on elements, which could lead to some exponential convergence rates [18]. We've plotted quadratic and cubic elements for 4 and 8 elements in Figure 3.8. We note once again that the higher polynomial degrees across the plateau (where the exact solution is roughly constant) are mostly unnecessary, and better approximations of the solution mostly come from the higher polynomial degrees near the boundary.

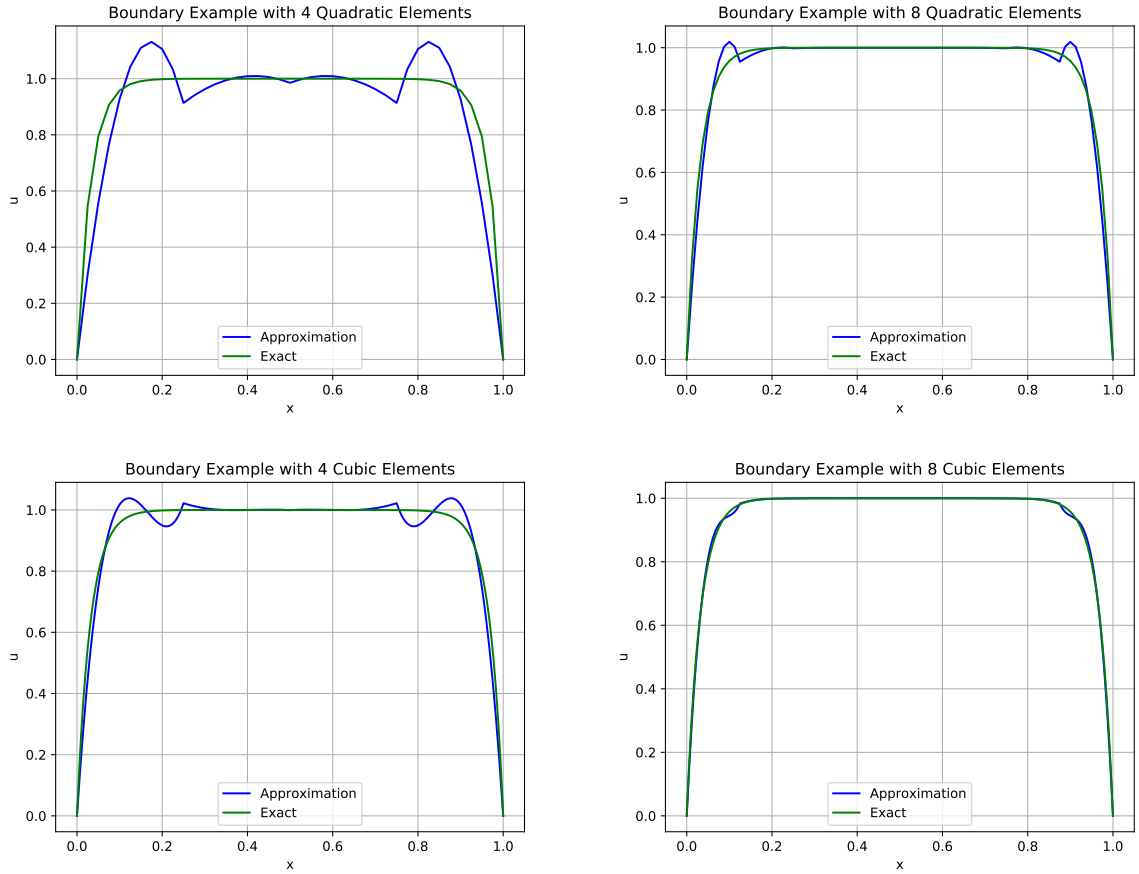


Figure 3.8: *Plots for the example boundary layer problem for varying element sizes and varying polynomial degrees.*

To try out more simple examples, you may visit `https://fem.blakey.family` where a simple version of Blakey FEM is running, allowing you to compute some of your own solutions. In particular you can try 3 different pre-set examples in the bottom-left of the page: one that pro-

duces the boundary layer solution like above, one that produces a sinusoidal solution, and one that produces a quadratic solution. By varying the number of elements we can see improvements in the solution or deteriorations in the solution. A small shortcut allows mixed-sized element meshes by entering a negative number of elements to the input, which can show interesting local convergence properties. This will give roughly twice as many elements to the left-side of the domain as on the right-side. You may also visit the GitHub repository, given on page 2, to compile and run the code yourself.

Section 4

A Posteriori Error Estimation and Adaptivity

A priori error bounds only go so far for giving us error estimations, as in real-world models we're unlikely to know the exact solutions. In some situations we may be able to use optimal solutions [15], or use a priori error bounds on model paramters [4]. However, for real world applications of finite element methods for solving PDEs, it may be that we cannot take reasonable guesses at the solution. After all, why would we be solving for a solution that we already know?

In general for an a priori bound, we have a bound of the form

$$\|u - u_h\| \leq \mathcal{E}_1(h, p, u),$$

where $\|\cdot\|$ denotes a suitable norm and the bound, $\mathcal{E}_1$, depends upon the element size, h, the polynomial degree on that element, p, and — crucially — the actual solution, u. It would be far better to derive computable error bounds that depend instead upon the finite element approximation of the solution, u_h, so that we can calculate the bound even when we don't know the exact solution. We will introduce such bounds in this section, referred to as a posteriori error bounds, which take the form

$$\|u - u_h\| \leq \mathcal{E}_2(h, p, u_h),$$

and therefore depend on our numerical approximation, u_h.

4.1 A Posteriori Error Estimation in 1D

When numerically solving real-life problems (such as computational fluid dynamics, elasticity, or weather prediction problems) we may find that the overall accuracy of the numerical approximation is degraded by local singularities; a remedy to this problem is to locally refine around areas where the approximate and analytical solution differ the most [39]. When working with test problems (e.g. $-u''(x) = sin(x)$) we know the analytical solution and can therefore immediately see where these areas are; however, in most practical situations the analytical solution is not known and we can't do this.

For finite element methods, there exist a posteriori error estimates (error estimates without knowing the analytical solution). We will derive such an error estimate for our model problem in Equation (2.4) in this section, mostly following the results from Schwab [32] as a guide.

We firstly define the energy norm, which is the norm in which we will measure our error and error estimates.

Definition 4.1 (Energy norm, [25, p. 55])**.**
The energy norm is defined as

$$\|u\|_E := a(u,u),$$

where a is the usual bilinear functional from our FEM.

In 1D, we may consider domain $\Omega := (a,b)$ and finite element space

$$V_h := \{u \in H^1(\Omega) : u|_{(x_{i-1},x_i)} \in \mathcal{P}_{P_i}(x_{i-1},x_i) \forall i \in [1,N]_{\mathbb{N}}\}.$$

We ultimately want to prove that there exists an a posteriori bound on $\|u-u_h\|_E$, but we first need to prove some preliminary results.

Let $\hat{\Omega} := [-1,1]$ be the one-dimensional reference element. Then we know by [32, eq. 3.3.3]

that, for any $u \in L^2(\hat{\Omega})$, we may write u in the form of an expansion of Legendre polynomials:

$$u(\xi) = \sum_{i=0}^{\infty} a_i L_i(\xi), \tag{4.1}$$

where

$$a_i = \frac{2i+1}{2} \int_{-1}^{1} u(\xi) L_i(\xi)\, d\xi,$$

and $\{L_i\}_{i=0}^{\infty}$ are the family of Legendre polynomials, as defined in Section 3.2. We note that the Legendre polynomials satisfy

$$\int_{-1}^{1} L_i(\xi) L_j(\xi)\, d\xi = \frac{2}{2i+1}\delta_{ij}, \tag{4.2}$$

where δ_{ij} denotes the Kronecker delta.

We may employ Equation (4.2), noting Equation (4.1), to deduce the following Parseval identity, as stated in a similar form by Schwab [32, eq. 3.3.14]:

$$\|u\|^2_{L^2(\hat{\Omega})} = \sum_{i=1}^{\infty} \frac{2}{2i+1} |a_i|^2. \tag{4.2*}$$

More generally, the following lemma holds.

Lemma 4.1 ([32, lem. 3.10])**.**

Given $u \in H^k(\hat{\Omega}), k \geq 0$, defined by Equation (4.1)*, then the following (generalised) Parseval identity holds:*

$$\int_{-1}^{1} |u^{(k)}(\xi)|^2 (1-\xi^2)^k\, d\xi = \sum_{i=k}^{\infty} |a_i|^2 \frac{2}{2i+1} \frac{(i+k)!}{(i-k)!}.$$

Proof. Firstly, we show that the following relation holds:

$$\int_{-1}^{1} (1-\xi^2)^k L_i^{(k)}(\xi) L_j^{(k)}(\xi)\, d\xi = \frac{2}{2i+1} \frac{(i+k)!}{(i-k)!} \delta_{ij}. \tag{4.3}$$

To prove Equation (4.3) we first note that Legendre polynomials are a special case of the

Jacobi polynomials

$$\{P_i(\xi;\alpha,\beta)\}_{i=0}^{\infty}$$

cf. [38, p. 58]. In particular the Jacobi polynomials satisfy the orthogonality property:

$$\begin{aligned}&\int_{-1}^{1}(1-\xi)^{\alpha}(1+\xi)^{\beta}P_i(\xi;\alpha,\beta)P_j(\xi;\alpha,\beta)\,d\xi\\&=\frac{2^{\alpha+\beta+1}}{2i+1+\alpha+\beta}\frac{\Gamma(\alpha+1+i)\Gamma(\beta+1+i)}{\Gamma(i+1)\Gamma(\alpha+\beta+1+i)}\delta_{ij};\alpha,\beta>-1,\end{aligned}\tag{4.4}$$

where $\Gamma(\cdot)$ denotes the gamma function.

Moreover, we note that

$$L_i^{(k)}(\xi)=\frac{(i+k)!}{2^k i!}P_{i-k}(\xi;k,k),k\geq i.\tag{4.5}$$

Hence, using both Equation (4.4) and (4.5) gives

$$\begin{aligned}&\int_{-1}^{1}(1-\xi^2)^k L_i^{(k)}(\xi)L_j^{(k)}(\xi)\,d\xi\\&=\int_{-1}^{1}(1-\xi)^{(k)}(1+\xi)^{(k)}\frac{(i+k)!}{2^k i!}\frac{(j+k)!}{2^k j!}P_{i-k}(\xi;k,k)P_{j-k}(\xi;k,k)\,d\xi\\&=\frac{[(i+k)!]^2}{2^{2k}[i!]^2}\frac{2^{2k+1}}{2i+1}\frac{\Gamma(i+1)\Gamma(i+1)}{\Gamma(i-k+1)\Gamma(i+k+1)}\delta_{ij}\\&=\frac{[(i+k)!]^2}{2^{2k}[i!]^2}\frac{2^{2k+1}}{2i+1}\frac{i!}{(i-k)!}\frac{i!}{(i+k)!}\delta_{ij}\\&=\frac{2}{2i+1}\frac{(i+k)!}{(i-k)!}\delta_{ij},\end{aligned}$$

hence, we deduce that Equation (4.3) holds.

To complete the proof of the lemma, we note that

$$\begin{aligned}
&\int_{-1}^{1}(1-\xi^2)^k|u^{(k)}(\xi)|^2\,d\xi \\
&= \int_{-1}^{1}(1-\xi^2)^k|\sum_{i=1}^{\infty}a_iL_i^{(k)}(\xi)|^2\,d\xi \\
&= \sum_{i,j=k}^{\infty}a_ia_j\int_{-1}^{1}(1-\xi^2)^kL_i^{(k)}(\xi)L_j^{(k)}(\xi)\,d\xi \\
&= \sum_{i=k}^{\infty}\frac{2}{2i+1}\frac{(i+k)!}{(i-k)!}|a_i|^2,
\end{aligned}$$

where we have employed Equation (4.3). □

With these results, we now consider the construction of a suitable projector $\pi_h : L^2(\hat{\Omega}) \rightarrow \mathcal{P}_P(\hat{\Omega})$, where $\mathcal{P}_P(\hat{\Omega})$ denotes the space of polynomials of degree less or equal to P, $P \geq 0$. To this end, we state the following result.

Lemma 4.2 ([32, eq. 3.3.14])**.**

For every $u \in L^2(\hat{\Omega})$ we have that

$$\inf_{v\in\mathcal{P}_P(\hat{\Omega})}\|u-v\|_{L^2(\hat{\Omega})} = \left[\sum_{i=p+1}^{\infty}\frac{2}{2i+1}|a_i|^2\right]^{\frac{1}{2}}.$$

Proof. Let $v \in \mathcal{P}_P(\hat{\Omega})$ be any polynomial of degree P, $P \geq 0$, then

$$v(\xi) = \sum_{i=0}^{P}b_iL_i(\xi)$$

for a given set of coefficients $\{b_i\}_{i=0}^{P}$. Then employing Equation (4.2*) gives

$$\|u-v\|^2_{L^2(\hat{\Omega})} = \sum_{i=0}^{P}\frac{2}{2i+1}|a_i-b_i|^2 + \sum_{i=P+1}^{\infty}\frac{2}{2i+1}|a_i|^2.$$

Hence $\|u-v\|_{L^2(\hat{\Omega})}$ will be minimised when $b_i = a_i$ for $i \in [1,P]_{\mathbb{N}}$, and we have our result.

□

Remark 4.1.

As we would expect, the function $v \in \mathcal{P}_P(\hat{\Omega})$, which minimises the norm $\|u - v\|_{L^2(\hat{\Omega})}$ is in fact the $L^2(\hat{\Omega})$-projection of u onto $\mathcal{P}_P(\hat{\Omega})$.

Based on previous results, we may derive an approximation result. However, we first, for $j \in [0, k]_{\mathbb{N}}$, $k \in \mathbb{N}$, define

$$V_j^k(\hat{\Omega}) := \{u \in L^2(\hat{\Omega}) : |u|_{V_j^k(\hat{\Omega})} < \infty\},$$

where

$$|u|^2_{V_j^k(\hat{\Omega})} \equiv \sum_{i=j}^{k} \int_{-1}^{1} (1 - \xi^2)^i |u^{(i)}(\xi)|^2 \, d\xi,$$

akin to [32, eq. 3.3.10].

Note that for $j = 0$, $|\cdot|_{V_j^k(\hat{\Omega})}$ is a norm, but only a semi-norm for $j > 0$.

Theorem 4.1 (Similar to [32, th. 3.11])**.**

Given $u \in V_0^k(\hat{\Omega})$, $k \geq 1$, the following approximation result holds

$$\inf_{v \in \mathcal{P}_P(\hat{\Omega}) \|u - v\|^2_{L^2(\hat{\Omega})}} \leq \left[\frac{(P + 1 - s)!}{(P + 1 + s)!}\right]^{\frac{1}{2}} |u|^2_{V_s^s(\hat{\Omega})},$$

for $s \in [0, \min(P + 1, k)]_{\mathbb{N}}$.

Proof. Employing Lemma 4.2 gives

$$\begin{aligned}
\inf_{v \in \mathcal{P}_P(\hat{\Omega})} \|u - v\|^2_{L^2(\hat{\Omega})} &= \sum_{i=P+1}^{\infty} \frac{2}{2i+1} |a_i|^2 \\
&= \sum_{i=P+1}^{\infty} \frac{2}{2i+1} |a_i|^2 \frac{(i+s)!}{(i-s)!} \frac{(i-s)!}{(i+s)!} \\
&\leq \frac{(P+1-s)!}{(P+1+s)!} \sum_{i=P+1}^{\infty} \frac{2}{2i+1} \frac{(i-s)!}{(i+s)!} \\
&= \frac{(P+1-s)!}{(P+1+s)!} |u|^2_{V_s^s(\hat{\Omega})},
\end{aligned}$$

by Lemma 4.1, as required. □

For the purposes of the a posteriori error estimation, we require an alternative approximation result, which we will now develop in a similar way.

Theorem 4.2 ([32, th. 3.14])**.**

Given $u \in H^1(\hat{\Omega})$ there exists $\pi_h u \in \mathcal{P}_P(\hat{\Omega})$ such that the following hold:

$$\pi_h u(\pm 1) = u(\pm 1), \tag{4.6a}$$

$$\|u' - (\pi_h u)'\|^2_{L^2(\hat{\Omega})} = \sum_{i=P}^{\infty} \frac{2}{2i+1} |b_i|^2, \tag{4.6b}$$

$$\|u - \pi_h u\|^2_{L^2(\hat{\Omega})} \leq \int_{-1}^{1} \frac{(u - \pi_h u)^2}{1 - \xi^2}\, d\xi \quad = \sum_{i=P}^{\infty} \frac{2}{i(i+1)(2i+1)} |b_i|^2. \tag{4.6c}$$

Here, $\{b_i\}_{i=0}^{\infty}$ are the Legendre coefficients of u', i.e.,

$$b_i = \frac{2i+1}{2} \int_{-1}^{1} u'(\xi) L_i(\xi)\, d\xi, i \in \mathbb{N}.$$

Proof. (4.6a) Firstly, we write $(\pi_h u)'$ to be the truncated Legendre series expansion of u', i.e.,

$$(\pi_h u)' = \sum_{i=0}^{P-1} b_i L_i(\xi),$$

and define

$$\pi_h u(\xi) = \int_{-1}^{\xi} (\pi_h u)'(\eta)\, d\eta + u(-1).$$

Hence, by definition of $\pi_h u(-1) = u(-1)$.

Moreover,

$$\begin{aligned}\pi_h u(1) &= \int_{-1}^{1} (\pi_h u)'(\eta)\, d\eta + u(-1) \\ &= 2b_i + u(-1) \\ &= 2b_i \pi_h u(-1)\end{aligned}$$

Hence,

$$\pi_h u(1) - \pi_h u(-1) = 2b_0.$$

Similarly,

$$\begin{aligned}u(1) - u(-1) &= \int_{-1}^{1} u'(\xi)\, d\xi \\ &= \int_{-1}^{1} \sum_{i=0}^{\infty} b_i L_i(\xi)\, d\xi \\ &= 2b_0.\end{aligned}$$

Thereby, given that $\pi_h u(-1) = u(-1)$, we deduce that $\pi_h u(1) = u(1)$, and hence Equation (4.6a) holds.

(4.6b) This follows immediately with Lemma 4.1.

(4.6c) First consider the following, where we have applied Equation (4.6a):

$$\begin{aligned}u(\xi) - \pi_h u(\xi) &= \int_{-1}^{\xi} u'(\eta)\, d\eta - \int_{-1}^{\xi} (\pi_h u)'(\eta)\, d\eta \\ &= \int_{-1}^{\xi} \sum_{i=P}^{\infty} b_i L_i(\eta)\, d\eta \\ &= \sum_{i=P}^{\infty} b_i \psi_i(\xi), \qquad (4.7)\end{aligned}$$

where $\psi_i(\xi) = \int_{-1}^{\xi} L_i(\eta)\, d\eta, i \in [p, \infty)_{\mathbb{N}}$.

We recall that the Legendre polynomials satisfy the ODE problem:

$$((1-\xi^2)L_i'(\xi))' + i(i+1)L_i(\xi) = 0, \text{ in } \hat{\Omega},$$

for $i \in \mathbb{N}$, by [38, th. 4.2.1].

Rearranging gives

$$L_i = -\frac{((1-\xi^2)L_i')'}{i(i+1)}, i \geq 1,$$

and integrating gives

$$\begin{aligned}\psi(\xi) &\equiv \int_{-1}^{\xi} L_i(\eta)\, d\eta \\ &= -\frac{1}{i(i+1)} \int_{-1}^{\xi} ((1-\eta^2)L_i'(\eta))'\, d\eta \\ &= -\frac{1}{i(i+1)}(1-\xi^2)L_i'(\xi).\end{aligned}$$

Hence,

$$\begin{aligned}\int_{-1}^{1} \frac{1}{1-\xi^2}\psi_i(\xi)\psi_j(\xi)\, d\xi &= \int_{-1}^{1} \frac{1}{i(i+1)}\frac{1}{j(j+1)}(1-\xi^2)L_i'(\xi)L_j'(\xi)\, d\xi \\ &= \frac{1}{[i(i+1)]^2}\frac{2}{2i+1}\frac{(i+1)!}{(i-1)!}\delta_{ij} \\ &= \frac{2}{2i+1}\frac{1}{i^2(i+1)^2}\frac{(i+1)i(i-1)!}{(i-1)!}\delta_{ij} \\ &= \frac{2}{i(i+1)(2i+1)}\delta_{ij}. \qquad (4.8)\end{aligned}$$

Employing Equations (4.7) and (4.8) gives

$$\begin{aligned}\int_{-1}^{1} |u(\xi) - \pi_h u(\xi)|^2\, d\xi &\leq \int_{-1}^{1} \frac{1}{1-\xi^2}|u(\xi) - \pi_h u(\xi)|^2\, d\xi \\ &= \int_{-1}^{1} \left(\sum_{i=P}^{\infty} b_i \psi_i(\xi)\right)^2 d\xi \\ &= \sum_{i=P}^{\infty} \frac{2}{i(i+1)(2i+1)}|b_i|^2,\end{aligned}$$

as required. □

From Theorem 4.6, we now derive the following approximation result.

Corollary 4.1 ([32, co. 3.15])**.**
Let $u \in H^1(\hat{\Omega}) \cap V_0^k(\hat{\Omega}), k \geq 1$*. Then the following bound holds:*

$$\|u' - (\pi_h u)'\|_{L^2(\hat{\Omega})} \leq \left[\frac{(p-s)!}{(p+s)!}\right]^{\frac{1}{2}} |u'|_{V_s^s(\hat{\Omega})},$$

where $s \in [0, \min(P,k)]_{\mathbb{N}}$*.*

Proof. From Equation (4.6b) from Theorem 4.2, together with the proof of Theorem 4.1, gives

$$\begin{aligned}
\|u' - (\pi_h u)'\|^2_{L^2(\hat{\Omega})} &= \sum_{i=P}^{\infty} \frac{2}{2i+1}|b_i|^2 \\
&= \sum_{i=P}^{\infty} \frac{2}{2i+1}\frac{(i+s)!}{(i-s)!}\frac{(i-s)!}{(i+s)!}|b_i|^2 \\
&\leq \frac{(P-s)!}{(P+s)!}\sum_{i=P}^{\infty} \frac{2}{2i+1}\frac{(i+s)!}{(i-s)!}|b_i|^2 \\
&= \frac{(P-s)!}{(P+s)!}\,|u'|^2_{V_s^s(\hat{\Omega})},
\end{aligned}$$

as required. □

For the purposes of the proceeding a posteriori error analysis we consider a particular case of Theorem 4.2, applied to an individual element, $\kappa_i = [x_{i-1}, x_i]$, $i \in [1, N]_{\mathbb{N}}$. To this end, consider the following element mapping:

$$\begin{aligned}
F_{\kappa_i} :&[-1,1] \to [x_{i-1}, x_i]; \\
F_{\kappa_i}(\xi) \equiv x &= \frac{1}{2}(1-\xi)x_{i-1} + \frac{1}{2}(1+\xi)x_i.
\end{aligned}$$

Note the slight change of notation, where we can write

$$\hat{u}(\xi) = u \circ F(\xi).$$

We may now introduce the following theorem as the linear approximation result that we will use.

Theorem 4.3 (1D linear approximation result, [32, p. 145])**.**
If $u \in H^1(\kappa_i), i \in [1,n]_{\mathbb{N}}$*, and* $\pi_h u(x_i) = u(x_i)$ *for* $i \in [0,n]_{\mathbb{N}}$*, then* $\Pi_h u \in V_h$ *s.t.*

$$\int_{x_{i-1}}^{x_i} w_i^{-1}(u - \pi_h u)^2 dx \leq \frac{1}{P_i(P_i+1)} \|u'\|^2_{L^2(\kappa_i)},$$

where $w_i = (x_i - x)(x - x_{i-1})$*.*

Proof. From Equation (4.6c) from Theorem 4.2, we have

$$\begin{aligned}
\int_{-1}^{1} \frac{(\hat{u} - \pi_h \hat{u})^2}{1-\xi^2}\, d\xi &= \sum_{i=P}^{\infty} \frac{2}{i(i+1)(2i+1)} |b_i|^2 \\
&= \sum_{i=P}^{\infty} \frac{2}{i(i+1)(2i+1)} \frac{(i-s)!}{(i-s)!} \frac{(i+s)!}{(i+s)!} |b_i|^2 \\
&\leq \frac{(P-s)!}{(P+s)!} \frac{1}{P(P+1)} \sum_{i=P}^{\infty} \frac{2}{2i+1} \frac{(i+s)!}{(i-s)!} |b_i|^2.
\end{aligned}$$

Employing Lemma 4.1 gives

$$\int_{-1}^{1} \frac{(\hat{u} - \pi_h \hat{u})^2}{1-\xi^2}\, d\xi \leq \frac{(P-s)!}{(P+s)!} \frac{1}{P(P+1)} \int_{-1}^{1} |\hat{u}^{(s+1)}|^2 (1-\xi^2)^s \, d\xi.$$

In order to scale from the reference element on $\hat{\Omega}$ to κ_i (on $[x_{i-1}, x_i]$), we first note that

$$\frac{\mathsf{d}x}{\mathsf{d}\xi} = \frac{1}{2}(x_i - x_{i-1})$$

and hence

$$\frac{1}{1-\xi^2} \left(\frac{\mathsf{d}x}{\mathsf{d}\xi} \right)^{-2} = \frac{1}{w_i}.$$

To see this, we note that

$$\xi = \frac{2x - (x_i + x_{i-1})}{x_i - x_{i-1}}$$

and therefore

$$\begin{aligned}
1-\xi^2 &= 1-\frac{(2x-(x_i+x_{i-1}))^2}{(x_i-x_{i-1})^2} \\
&= \frac{(x_i-x_{i-1})^2-(2x-(x_i+x_{i-1}))^2}{(x_i-x_{i-1})^2} \\
&= \frac{4(x_i-x)(x-x_{i-1})}{(x_i-x_{i-1})^2}.
\end{aligned}$$

We may now set $s=0$, which gives

$$\begin{aligned}
\int_{x_{i-1}}^{x_i} w_i^{-1}(u-\pi_h u)^2\,dx &= \int_{-1}^{1}(\hat{u}-\pi_h\hat{u})^2 w_i^{-1}\frac{\mathsf{d}x}{\mathsf{d}\xi}\,d\xi \\
&= \int_{-1}^{1}\frac{1}{1-\xi^2}(\hat{u}-\pi_h\hat{u})^2\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^{-1}d\xi \\
&\leq \frac{1}{P(P+1)}\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^{-1}\int_{-1}^{1}(\hat{u}')^2\,d\xi.
\end{aligned}$$

Now

$$\begin{aligned}
u' &\equiv u_x \\
&= \hat{u}_\xi\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^{-1} \\
&\equiv \hat{u}'\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^{-1}.
\end{aligned}$$

Thus,

$$\begin{aligned}
\int_{x_{i-1}}^{x_i} w_i^{-1}(u-\pi_h u)^2\,dx &\leq \frac{1}{P(P+1)}\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^{-1}\int_{x_{i-1}}^{x_i}(u')^2\left[\frac{\mathsf{d}x}{\mathsf{d}\xi}\right]^2\frac{\mathsf{d}\xi}{\mathsf{d}x}\,dx \\
&= \frac{1}{P(P+1)}\|u'\|^2_{L^2(x_{i-1},x_i)},
\end{aligned}$$

and noting $\kappa_i=[x_{i-1},x_i]$ we are done.

□

For the following theorem's proof we also make note of the Galerkin orthogonality property, as stated in [25][eq. 1.35].

$$a(u - u_h, v_h) = 0, \forall v_h \in V_h. \tag{4.9}$$

We also make the following definition of the residual.

Definition 4.2 (Residual)**.**

For one-dimensional version of the model problem given in Equation (2.4)*, we have the residual:*

$$R(u)|_{(x_{i-1},x_i)} := f + \epsilon u'' - cu,\ i \in [1, n]_{\mathbb{N}}.$$

We will now state and prove the main theorem providing an a posteriori error bound.

Theorem 4.4 (1D a posteriori error bound)**.**

If $u \in H^1(\Omega)$ *and* u *satisfies Equation* (2.4)*, then*

$$\|u - u_h\|_E \leq \sqrt{\sum_{i=1}^{N} \frac{1}{P_i(P_i+1)} \frac{1}{\epsilon} \left\| w_i^{1/2} R(u_h) \right\|^2_{L^2(x_{i-1},x_i)}},$$

where $\|\cdot\|_E$ *denotes the energy norm,* P_i *is the element's polynomial degree,* w_i *is defined as in Theorem 4.3, and* u_h *is our approximation of the solution defined in Equation* (2.6)*.*

Proof. We note that Equation (2.4) tells us that

$$-\epsilon u''(x) + c(x)u(x) = f(x), x \in (0, 1)$$

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

The problem's weak formulation is, as written in Equation (5.2), find $u \in H_0^1(0,1)$ s.t.

$$a(u,v) = l(v),\ \forall v \in H_0^1(0,1),$$

Projecting our problem to the finite-dimensional space, as shown in Equation (2.6), we have to find $u_h \in V_h$ s.t.

$$a(u_h, v_h) = l(v_h) \forall v_h \in V_h.$$

By working from the definition of the energy norm, we get:

$$\|u - u_h\|_E^2 = a(u - u_h, u - u_h).$$

By defining $e := u - u_h$ and using Galerkin orthogonality (Equation (4.9)) in the second argument, we get

$$\begin{aligned}\|u - u_h\|_E^2 &= a(u - u_h, e) \\ &= a(u - u_h, e - \pi_h e).\end{aligned}$$

By linearity we may split up the terms in the first argument to give

$$\|u - u_h\|_E^2 = a(u, e - \pi_h e) - a(u_h, e - \pi_h e).$$

We now substitute the model equation (Equation (2.4)) into the first term to give

$$\begin{aligned}\|u-u_h\|_E^2 &= l(e-\pi_h e) - a(u_h, e-\pi_h e)\\ &= \int_0^1 \left[f(e-\pi_h e) - \epsilon u_h'(e-\pi_h e)' - cu_h(e-\pi_h e)\right] dx.\end{aligned}$$

Applying integration by parts elementwise and noting the vanishing boundary conditions by Theorem 4.3 gives

$$\begin{aligned}\|u-u_h\|_E^2 &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} \left[f(e-\pi_h e) + \epsilon u_h''(e-\pi_h e) - cu_h(e-\pi_h e)\, d\xi\right]\\ &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} (f + \epsilon u_h'' - cu_h)(e-\pi_h e)\, d\xi.\end{aligned}$$

By using the definition of the residual in Equation (4.2), we have

$$\begin{aligned}\|u-u_h\|_E^2 &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} (R(u_h))(e-\pi_h e) dx\\ &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} w_i^{1/2}(R(u_h)) w_i^{-1/2}(e-\pi_h e) dx,\end{aligned}$$

where $w_i := (x_i - x)(x - x_{i-1})$.

Using the Cauchy-Schwarz inequality gives

$$\|u-u_h\|_E^2 \le \sum_{i=1}^{N} \sqrt{\int_{x_{i-1}}^{x_i} w_i (R(u_h))^2 dx} \sqrt{\int_{x_{i-1}}^{x_i} w_i^{-1}(e-\pi_h e)^2 dx}$$
$$= \sum_{i=1}^{N} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}} \sqrt{\int_{x_{i-1}}^{x_i} w_i^{-1}(e-\pi_h e)^2 dx}.$$

Applying our error bound in Theorem 4.3 we have

$$\|u-u_h\|_E^2 \le \sum_{i=1}^{N} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}} \sqrt{\frac{1}{P_i(P_i+1)} \|e'\|^2_{L^2(x_{i-1},x_i)}}$$
$$= \sum_{i=1}^{N} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}} \sqrt{\frac{1}{\epsilon P_i(P_i+1)} \|\epsilon^{1/2} e'\|^2_{L^2(x_{i-1},x_i)}}$$
$$= \sum_{i=1}^{N} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}} \left\| \epsilon^{1/2} e' \right\|_{L^2(x_{i-1},x_i)} \sqrt{\frac{1}{\epsilon P_i(P_i+1)}}.$$

Applying the Cauchy-Schwarz inequality, we have

$$\|u-u_h\|_E^2 \le \sqrt{\sum_{i=1}^{N} \|\epsilon^{1/2} e'\|^2_{L^2_{(x_{i-1},x_i)}}} \sqrt{\sum_{i=1}^{N} \frac{1}{\epsilon p_i(p_i+1)} \left\| w_i^{1/2} R(u_h) \right\|^2_{L^2_{(x_{i-1},x_i)}}}.$$
$$= \left\| \epsilon^{1/2} e' \right\|_{L^2(0,1)} \sqrt{\sum_{i=1}^{N} \frac{1}{\epsilon p_i(p_i+1)} \left\| w_i^{1/2} R(u_h) \right\|^2_{L^2_{(x_{i-1},x_i)}}}.$$

Noticing that $\|e\|_E^2 \equiv \left\|\epsilon^{1/2} e'\right\|^2_{L^2(0,1)} + \left\|c^{1/2} e\right\|^2_{L^2(0,1)} \ge \left\|\epsilon^{1/2} e'\right\|^2_{L^2(0,1)}$, we have

$$\|u-u_h\|_E^2 \leq \|e\|_E \sqrt{\sum_{i=1}^{N} \frac{1}{\epsilon p_i(p_i+1)} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}}^2}$$
$$= \|u-u_h\|_E \sqrt{\sum_{i=1}^{N} \frac{1}{\epsilon p_i(p_i+1)} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}}^2}$$

By dividing through by the norm of the error in the energy norm, we further simplify to

$$\|u-u_h\|_E \leq \sqrt{\sum_{i=1}^{N} \frac{1}{\epsilon p_i(p_i+1)} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}}^2},$$

which gives the desired result.

□

Since each term in the sum is dependant only on the properties of a single element, say κ_i, we also make the further definition of

$$\eta_{\kappa_i} := \frac{1}{\sqrt{\epsilon p_i(p_i+1)}} \left\| w_i^{1/2} R(u_h) \right\|_{L^2_{(x_{i-1},x_i)}}$$

as the element or local error indicator so that we can further write

$$\|u-u_h\|_E \leq \sqrt{\sum_{i=1}^{N} \eta_{\kappa_i}^2}.$$

We will then write this upper bound as

$$\mathcal{E}(u_h, h, p) := \sqrt{\sum_{i=1}^{N} \eta_{\kappa_i}^2}$$

which will be referred to as the global error indicator.

4.2 Example Problems

We introduce here some model problems that will be used for numerical experiments with the h-, p-, and hp-adaptive algorithms in Sections 4.3–4.5.

Problem 4.1.

A sinusoidal example, for which the solution the solution is very smooth. The problem has the exact solution

$$u(x) = \sin(2\pi x).$$

The data is set on Equation (2.4a) *as* $\epsilon = 1$, $f = 4\pi^2 \sin(2\pi x)$, *and* $c \equiv 0$.

Problem 4.2.

A boundary layer problem, exhibiting boundaries near $x = 0$ *and* $x = 1$, *as given in [42, ex. 2] with the exact solution*

$$u(x) = -\frac{\exp(x/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} - \frac{\exp(-x/\sqrt{\epsilon})\exp(1/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} + 1.$$

The data is set on Equation (2.4a) *as* $\epsilon = 10^{-3}$, $f \equiv 1$, *and* $c \equiv 1$.

Problem 4.3.

A problem exhibiting a shock, as given in [42, ex. 4] with the exact solution

$$u(x) = \arctan(100(x - 1/3)) + (1 - x)\arctan(100/3) - x\arctan(200/3).$$

The data is set on Equation (2.4a) *as* $\epsilon = 1$, $f = \frac{4\times10^6(x-1/3)}{(10^5(x-1/3)^2+1)^2}$, *and* $c \equiv 1$. *Again, we note that* f *is just* $-u'' + u$.

4.3 h-adaptivity

This section will be dedicated to looking at strategies to adaptively change the mesh locally in order to reduce the energy norm error of the solution.

We need an algorithm that will instruct us on how we will construct successive meshes, each more refined than the previous. We will use a modified version of the algorithm described in [39, p .68] by Verfürth, which is given in Algorithm 4.1; this algorithm has τ_h^k as the kth mesh, u_h^k as the finite element solution on τ_h^k, and η_κ as the individual element indicators and $\mathcal{E}$ as the global error indicator, c.f. the end of Section 4.1. The underline on the word ”refine” is because we need to do something further — how do we deem elements as big contributors to the global error, and then how do we construct the subsequent refinement to the mesh and solution?

Algorithm 4.1: Refinement

Create initial mesh, τ_h^0;
Compute initial solution, u_h^0;
Compute all η_κ and $\mathcal{E}$;
$k \leftarrow 0$;
while $\mathcal{E} \geq$ *TOL* **do**
  Refine mesh and solution, giving τ_h^{k+1} and u_h^{k+1};
  Solve u_h^k on τ_h^k;
  Compute all η_κ and $\mathcal{E}$;
  $k \leftarrow k + 1$

For h-refinement, as described in [5, p. 748] and [41, p. 772], our aim is to make the error uniform across all elements. By the strategy outlined in [23, p. 18] we will choose to refine all those elements with local error indicators that are greater or equal to one third of the largest local error indicator. Algorithm 4.2 outlines how we have implemented this, where the algorithm takes the current mesh, current solution, and error indicators for all elements as inputs; and then returns a refined mesh and solution.

Algorithm 4.2: h-refinement

Input : $\tau_h, u_h, \boldsymbol{\eta_\kappa}$
Output: $\tau_h^{\text{new}}, u_h^{\text{new}}$
forall κ **do**
  if $\eta_\kappa \geq \max \boldsymbol{\eta_\kappa}/3$ **then**
    Split element in half and add both these elements to τ_h^{new}
  else
    Copy element from τ_h to τ_h^{new}

4.3.1 Test Problem 1

For Problem 4.1, we may apply the above algorithms adaptively refine the mesh to provide more accurate solutions.

By initialising τ_0 to have 4 linear elements, and allowing the algorithm to run, we get the results shown in Figure 4.1 for the first 3 h-adaptive steps.

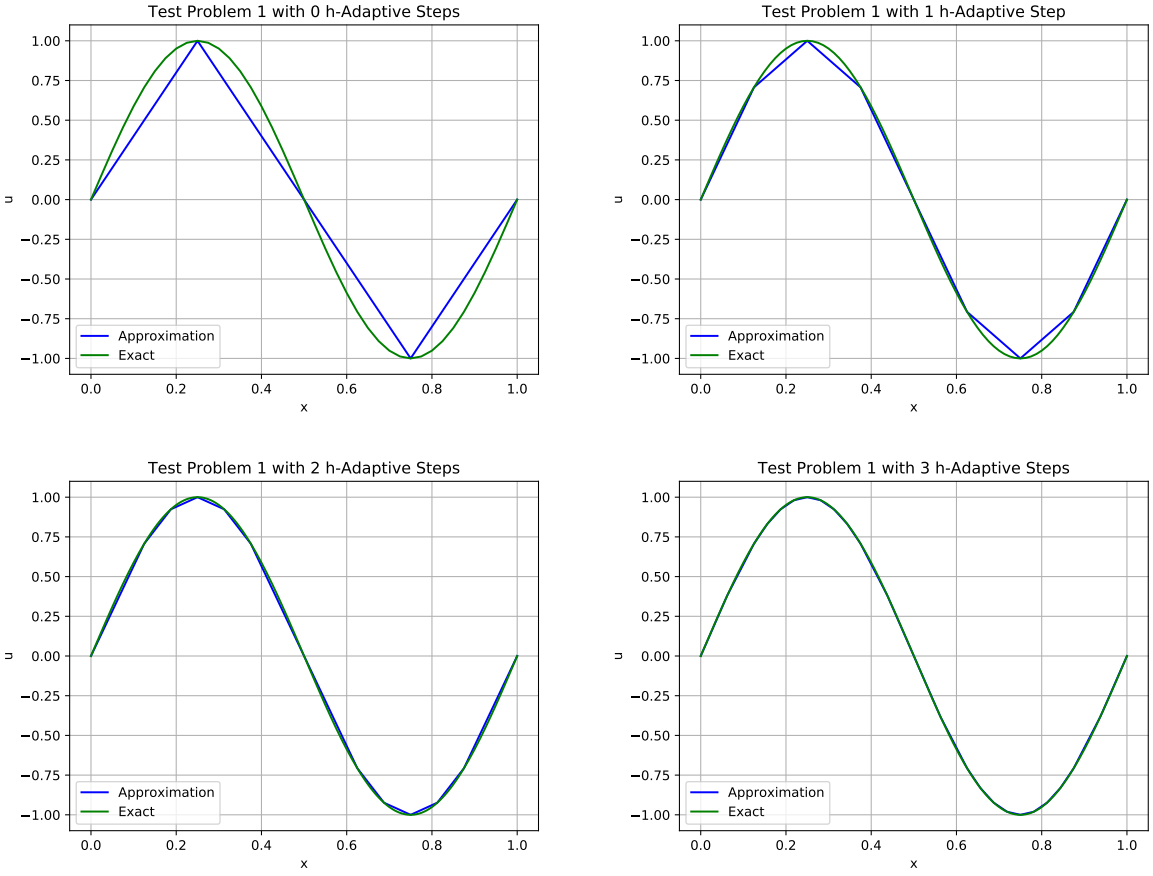


Figure 4.1: *h-adaptivity on Problem 4.1.*

We see that the initial mesh with four linear elements is not very good at approximating the exact solution (and in fact $\|u - u_h\|_E \approx 1.93$ here). The first step of the refinement algorithm automatically marks all elements for refinement, resulting in a mesh with eight elements, which is again a better approximation of the exact solution. Figure 4.2 shows both the energy norm between the exact and approximate solutions ($\|u - u_h\|_E$) as well as the error indicator ($\mathcal{E}(u_h, h, p)$) against the degrees of freedom for the first 15 h-refinement steps. We can see that the error is reducing at some polynomial rate — and is importantly staying under the error estimator, since it is an upper bound.

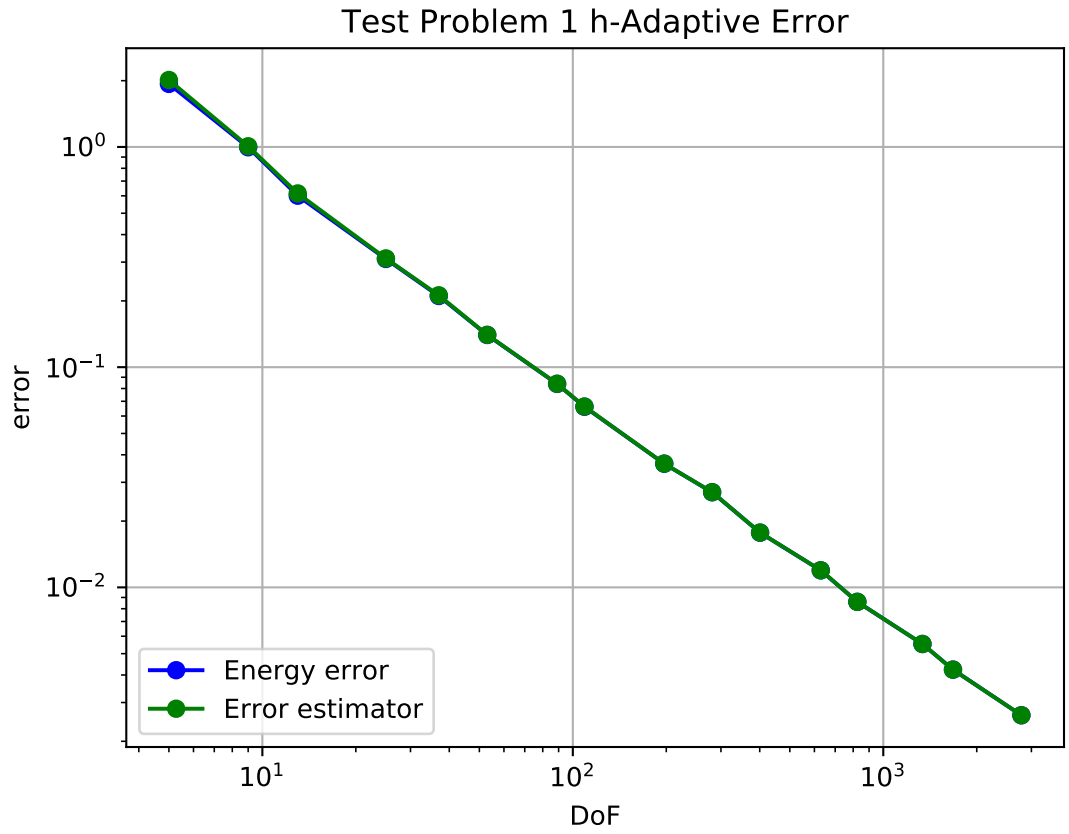


Figure 4.2: *h-adaptivity convergence rates on Problem 4.1.*

This means that refinement is working at reducing the error, but how does the efficiency of this compare to global refinement? We can see in Figure 4.3 that the h-adaptive version, as well as taking more steps, performs very slightly better. It may seem at this point that there is no real advantage to h-adaptivity, but we will see in Section 4.4 that this particular problem performs significantly better with p-adaptivity.

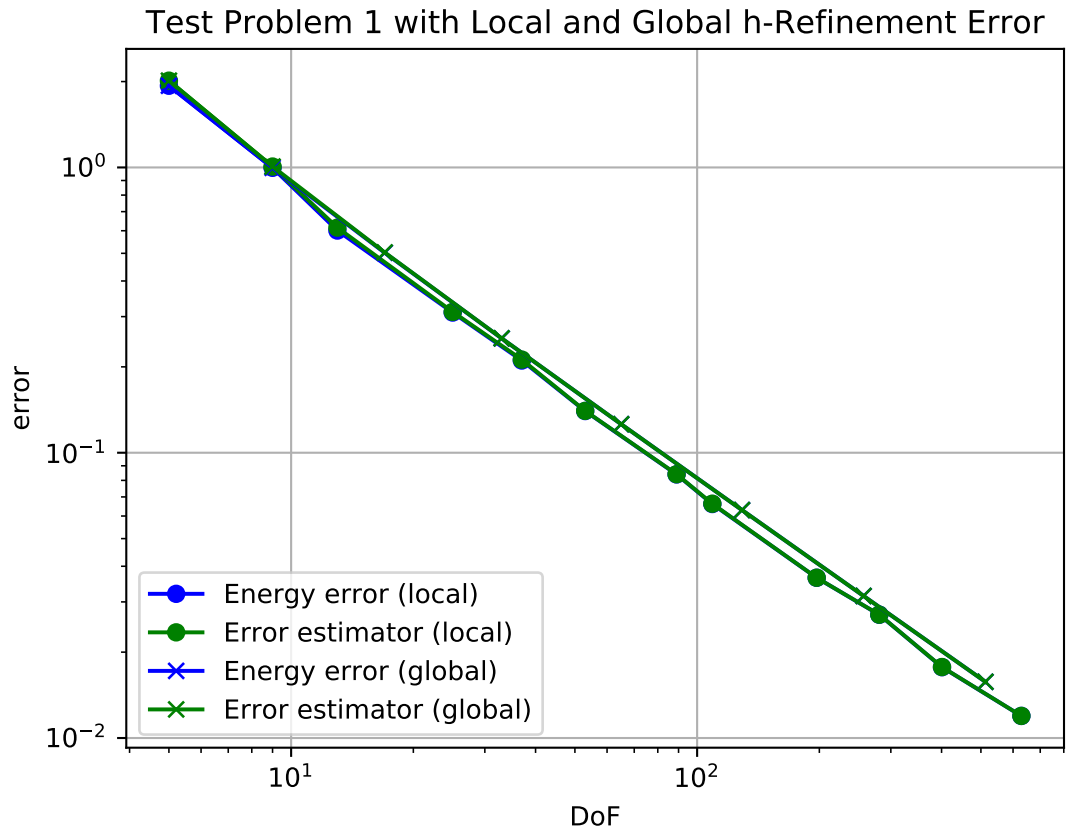


Figure 4.3: *h-adaptivity (local refinement) and global h-refinement convergence rates on Problem 4.1.*

We note that for this problem we can calculate the efficiency indices of our error bounds with

$$\Theta = \frac{\mathcal{E}(u_h, h, p)}{\|u - u_h\|_E},$$

which gives us an indication of how close our error estimate is compared to the actual error in the solution. Note that we can only calculate this quantity here because we know the exact solution. The first ten steps of the h-adaptive algorithm produce the results in Table 4.1. We see that the error estimator is initially about 4% inefficient for the initial condition of 4 elements, but this decreases down to 0.1% inefficiency by the time that the algorithm has split the mesh into 280 elements. This means that our bound for this problem is very tight and we aren't doing too many unnecessary refinements.

N	Θ
4	1.042
8	1.010
12	1.024
24	1.006
36	1.007
52	1.001
88	1.002
108	1.002
196	1.001
280	1.001

Table 4.1

4.3.2 Test Problem 2

Recall that Problem 4.2 is a boundary layer problem, exhibiting boundaries near $x = 0$ and $x = 1$, with the exact solution

$$u(x) = -\frac{\exp(x/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} - \frac{\exp(-x/\sqrt{\epsilon})\exp(1/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} + 1,$$

where $\epsilon = 10^{-3}$.

We initialise the h-adaptive algorithm with a mesh of 4 linear elements, and get the results in Figure 4.4 after 3 h-adaptive steps.

We see that the initial mesh is very bad at approximating the solution: the approximate solution does not capture the plateau through the centre of the domain, it does not capture the sharp derivative near the boundaries, and there is an erroneous overshoot in the solution's maximum value. However, by 2 and 3 refinement steps we see the features of the true solution becoming apparent in the approximate solution.

We notice that, unlike Problem 4.1, that the h-adaptive algorithm produces very different results to global refinement. Figure 4.5 shows, for the same initial condition, global refinement having a much higher error for the same number of degrees of freedom. Interestingly, we see some very high convergence rates for local refinement when the algorithm is starting. This is

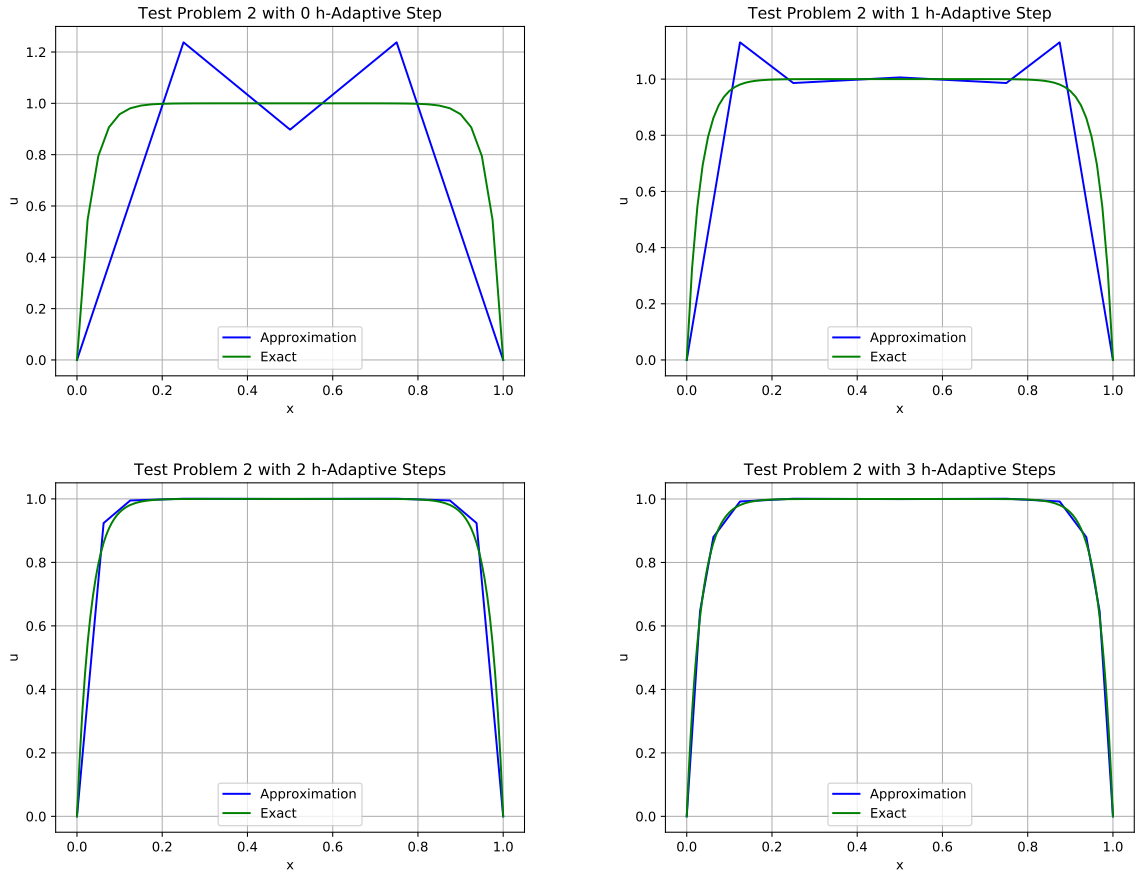


Figure 4.4: *h-adaptivity on Problem 4.2.*

likely to be because this is the stage at which the boundaries are resolved in the approximation, and we therefore get much closer to the solution. When solving these numerical experiments, the generation of the global refinement data used significantly more computational resources on my computer than the local adaptivity data, and this is directly due to the larger number of degrees of freedom. We note that there were roughly four times the number of degrees of freedom involved in reducing the error in the global refinement to roughly the same amount of error in the local adaptivity.

We may plot the mesh from the h-adaptivity for the first 14 refinement steps to give us an idea of how the elements have been marked for refinement, given in Figure 4.6. As one may expect, with having errors initially high near the boundary, the mesh has been mostly refined near the boundaries.

Also note that we could have performed similar analysis to those above without the exact

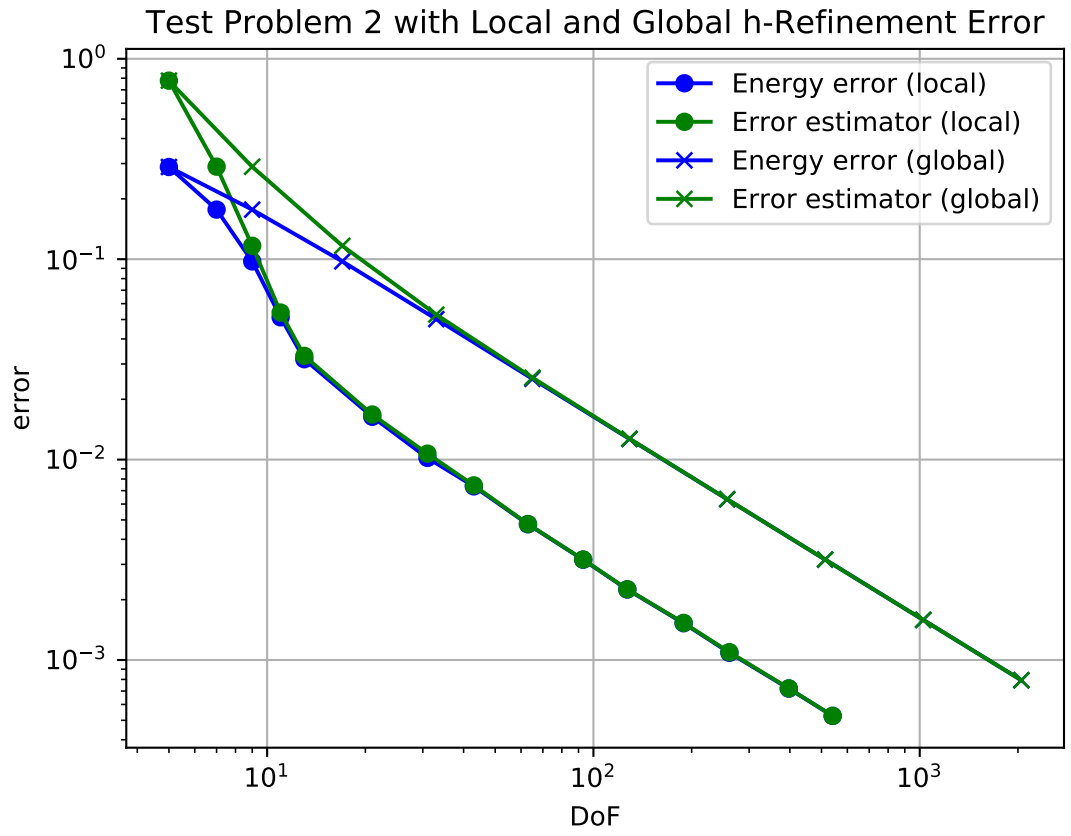


Figure 4.5: *h-adaptivity (local refinement) and global h-refinement convergence rates on Problem 4.2.*

solutions to the equations; however it is still very useful to have this for these test problems, as we can make further comparisons between the actual error and the estimate error.

4.3.3 Test Problem 3

Recall that Problem 4.3 is a problem exhibiting a shock with the exact solution

$$u(x) = \arctan(100(x - 1/3)) + (1 - x)\arctan(100/3) - x\arctan(200/3).$$

As a change to the previous examples, we set the initial mesh for this problem to have 6 linear elements; this is because 4 elements don't yield desirable results for some of the adaptivity algorithms due to a lack of resolution over the shock. The first 3 steps of the h-adaptive

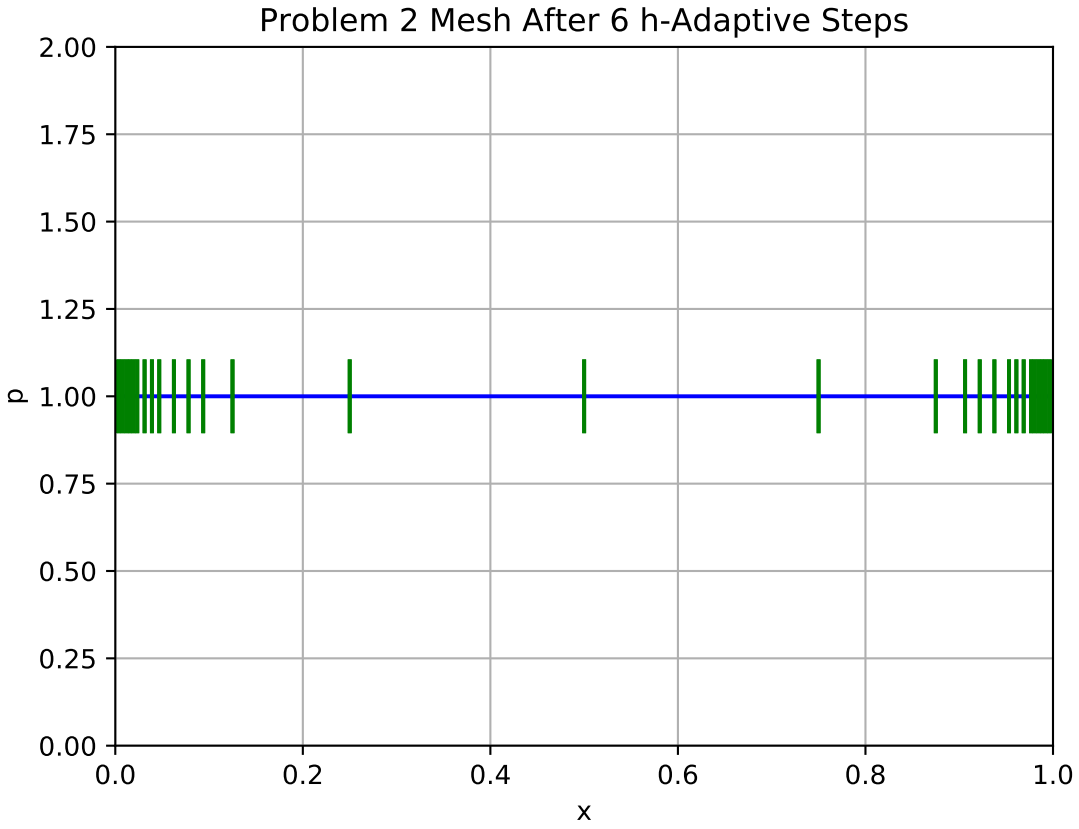


Figure 4.6: *Resulting h-adaptivity mesh for Problem 4.2.*

algorithm are shown in Figure 4.7.

The initial mesh produces a solution that roughly describes the true solution near the boundaries, but doesn't do a very good job at approximating the true solution in the middle of the domain. Adding to this observation, we see that the derivative across the shock points becomes steeper between adaptive steps, until the steepness is roughly met by the third step. By this third step we have most of the features that we would expect to see in a good approximation of the solution.

As shown in Figure 4.8 we very clearly see that the mesh has rightfully been refined around the shock region, allowing the solution to become much more accurate. Because the mesh is so fine, our plot does not really show how fine the mesh goes at its finest, but we see the important feature that the element sizes are certainly smaller in this region.

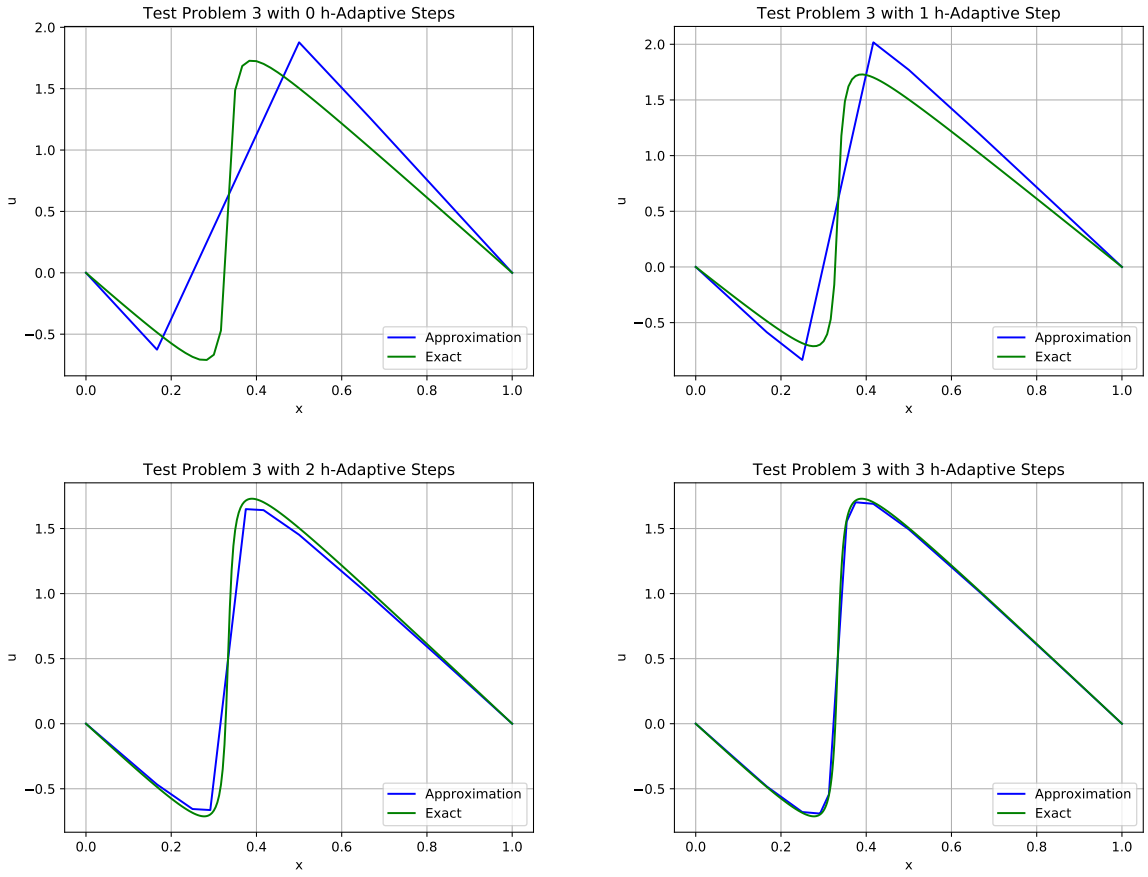


Figure 4.7: *h-adaptivity on Problem 4.3.*

4.4 p-adaptivity

Here, just like with h-adaptivity, we again choose to mark elements for refinement for those elements that have local error indicators greater or equal to one third of the largest local error indicator. However we now refine elements by increasing the polynomial on elements marked for refinement, rather than splitting them into two new elements. This procedure is outlined in Algorithm 4.3.

Algorithm 4.3: p-refinement

Input : $\tau_h, u_h, \boldsymbol{\eta_\kappa}$
Output: $\tau_h^{\text{new}}, u_h^{\text{new}}$
forall κ **do**
  if $\eta_\kappa \geq \max \boldsymbol{\eta_\kappa}/3$ **then**
    Increase polynomial degree on element;

This is somewhat of an easier algorithm to implement computationally as the number of

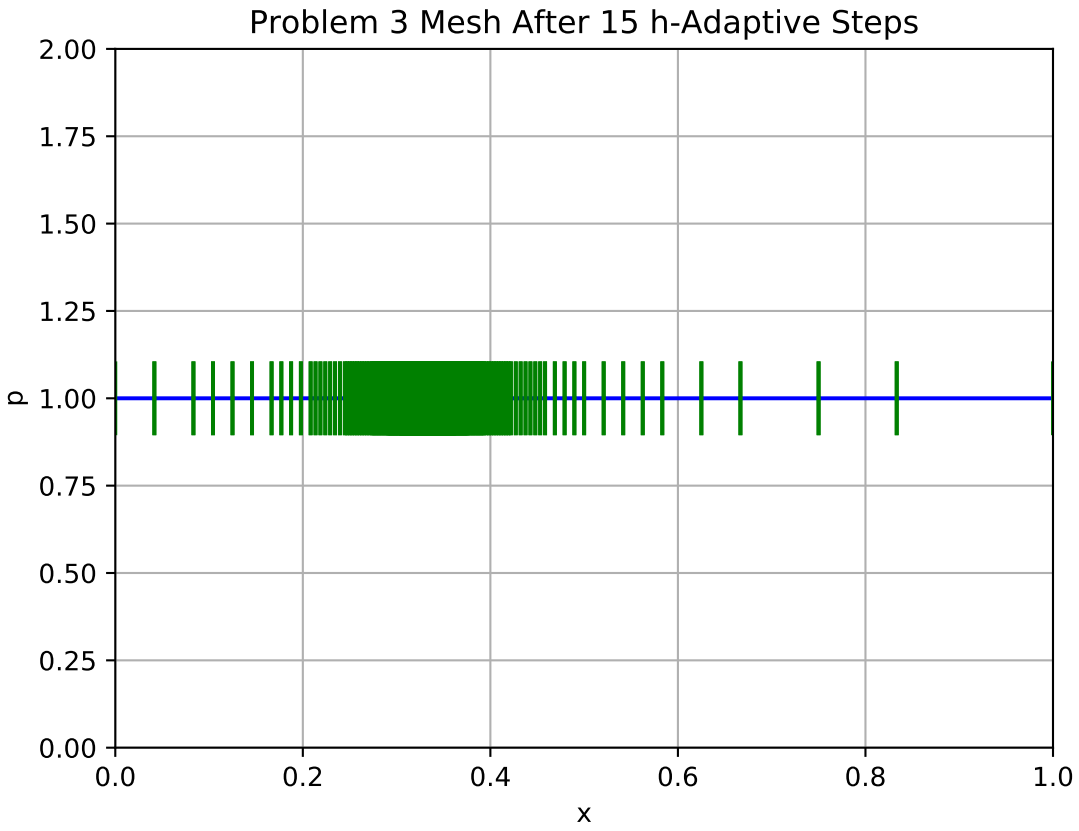


Figure 4.8: *Resulting h-adaptivity mesh for Problem 4.3.*

elements between subsequent iterations remains the same.

We note that, as described in [18], we can get exponentially converging solutions for solutions that are very smooth (solutions lie in C^p for some large p). This is very good news for our convergence rates: this means that, for sufficiently smooth functions, we will converge to the solution extremely quickly. This idea will be further developed in Section 4.5, where we will combine this extraordinarily useful feature with mesh refinement.

4.4.1 Test Problem 1

For Problem 4.1 we again choose an initial mesh with 4 linear elements.

Immediately we can see in Figure 4.9 that the solution, after one p-adaptive step, is very close to the exact solution. As we will see in Section 4.5 this makes sense with the smoothness

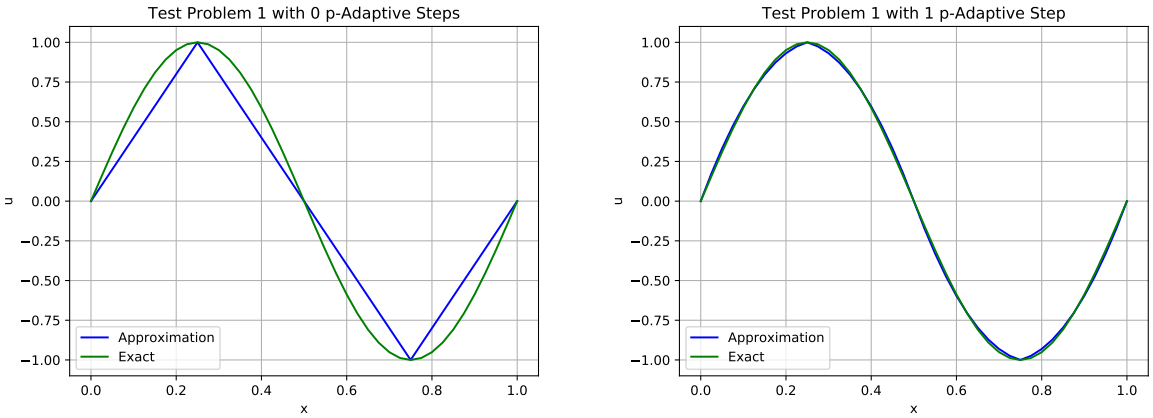


Figure 4.9: *p-adaptivity on Problem 4.1.*

of the solution to this problem.

We can study the convergence rates for this problem under p-adaptivity, as can be seen in Figure 4.10. As discussed for a similar smooth problem in [17, p. 15] we see exponential convergence rates here. In fact, with the slight concavity we're actually seeing some super-exponential convergence rates here.

One thing that we do notice about Figure 4.10 that may be slightly concerning is the point where the error in the energy norm is higher than the error estimate at 25 degrees of freedom. We've calculated an upper bound to the error, so the bound should never be higher than the error. However, we must remember that these calculations have been computed on a computer with finite precision, and so this situation may have happened due to machine precision. In fact, we may plot the error and estimator values for a few more iterations of p-refinement in Figure 4.11 to see that the error actually increases with increasing degrees of freedom; this is further evidence to suggest that we may be having issues with machine precision.

We note that the p-adaptive process has actually given us the same as global p refinement, meaning that the error estimator must have been roughly equally distributed. We can see, though, that this was probably the right thing to do! Figure 4.12 shows the convergence rates for both the h- and p-adaptive algorithms, and shows that p-adaptivity is far superior in reduc-

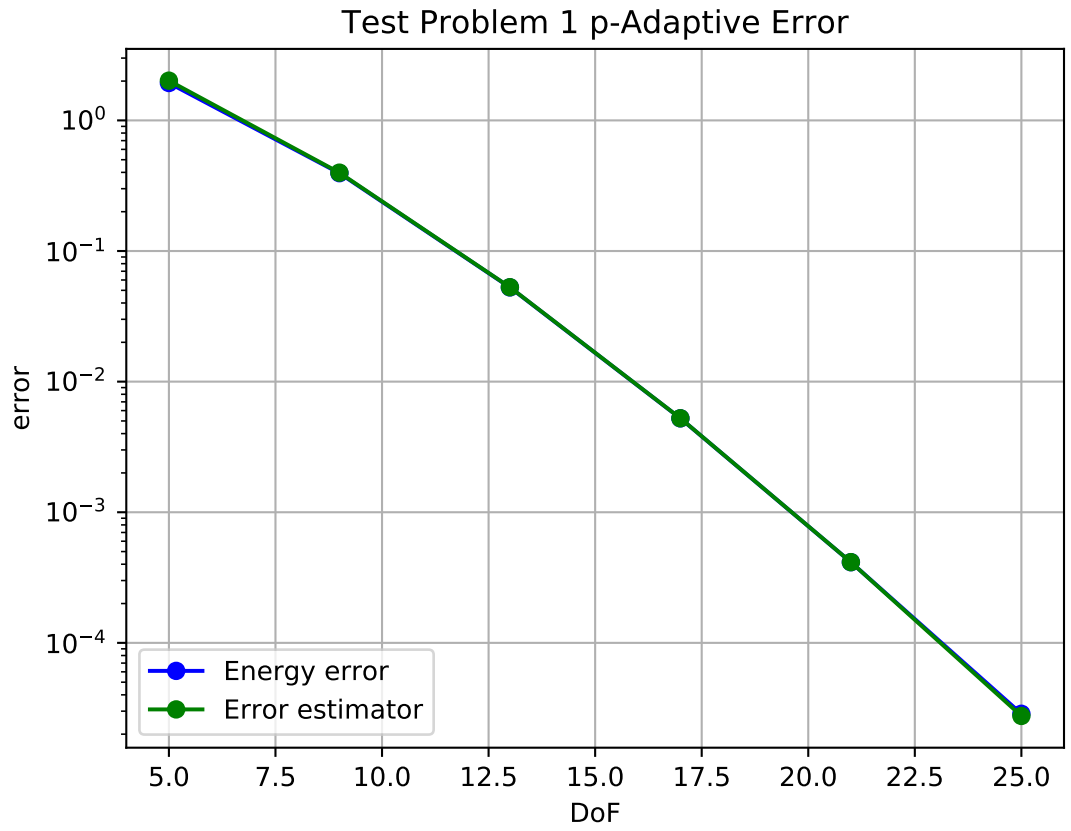


Figure 4.10: *p-adaptivity convergence rates on Problem 4.1.*

ing the error than h-adaptivity for this problem.

Our efficiency indices (defined in Section 4.3) here begin with $\Theta_0 = 1.042$, and then by third iteration reduce to $\Theta_3 = 1.002$; this means, just as what we saw with h-adaptivity, the error bound on this problem is very efficient.

4.4.2 Test Problem 2

For p-adaptivity for Problem 4.2, we choose an initial mesh of 4 linear elements, and produce Figure 4.13 after 3 p-adaptive steps.

We see that the results here look a little strange to begin with: there are various oscillations that appear across the transition between the boundary layers and the plateau in the centre of the domain. However these oscillations eventually disappear once we introduce enough

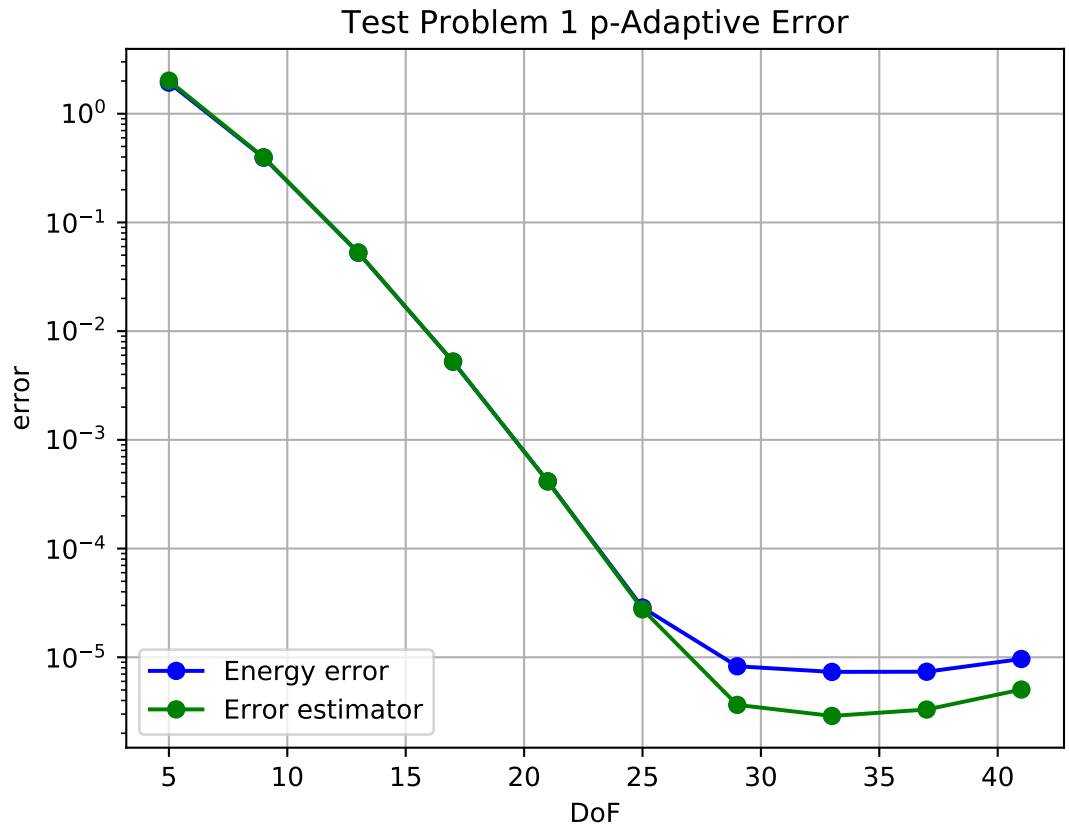


Figure 4.11: *p-adaptivity convergence issues on Problem 4.1.*

degrees of freedom, as shown in the convergence rate in Figure 4.14. Interestingly, the error indicator does not perform as efficiently as it did for the h-adaptive algorithm, with an efficiency index of $\Theta = 1.401$ at the eighth adaptive step. This isn't a huge problem since the indicator remains above the actual error, but it shows that the error indicator may not perform as efficiently in some problems.

Comparing h- and p-adaptivity for this problem, Figure 4.15 shows us that p-adaptivity is better for this test problem, with its far superior convergence rate.

4.4.3 Test Problem 3

For p-adaptivity of Problem 4.3, we (like in Section 4.3) start with an initial mesh of 6 linear

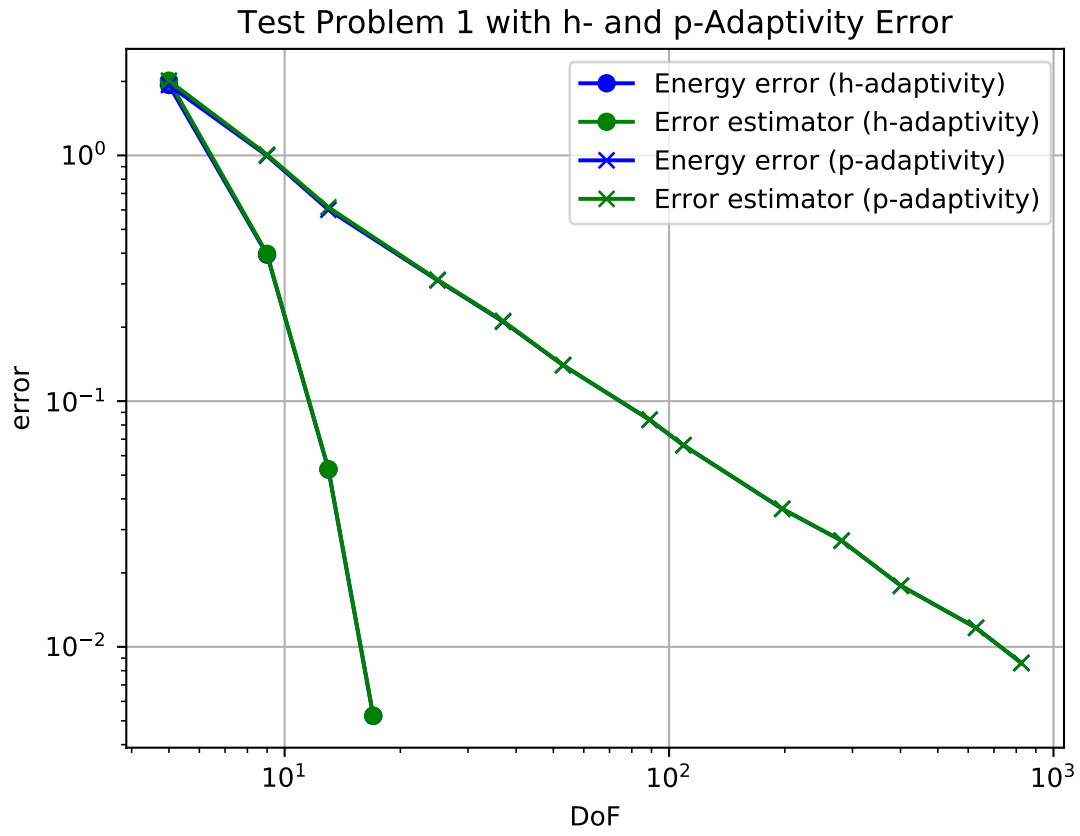


Figure 4.12: *h- and p-adaptivity convergence on Problem 4.1.*

elements. The results for the first 3 steps of p-adaptivity are shown in Figure 4.16.

We notice that after the first p-adaptive step, there is a clear overshoot of the solution's value, likely to be caused by the steep derivative needed over the shock. As we take more steps we see these oscillations both spread out and become smaller in amplitude as the polynomial degrees in that region are increased. Figure 4.17 shows the polynomial degrees across the mesh after the third p-adaptive step, and shows that the polynomial degree on the two elements closest to the shock have been increased to 4; the polynomial degree has been left unchanged as 1 elsewhere on the domain.

4.5 hp-adaptivity

Here we will combine the strategies of the above by making local refinements to both the mesh and the polynomial degrees in a suitable combination. We will still mark elements that

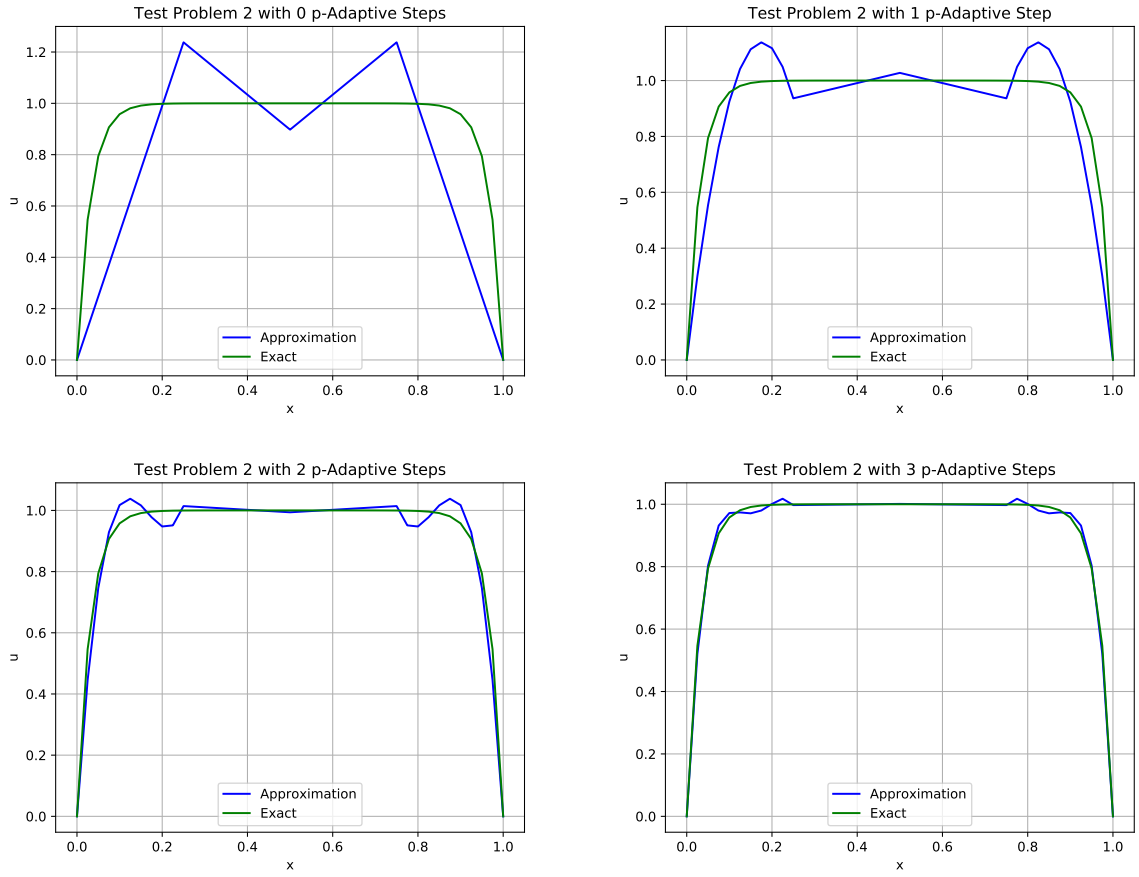


Figure 4.13: *p-adaptivity on Problem 4.2. Note that for plotting purposes we take 10 sample points per element, which leads to plots 3 and 4 looking not as smooth as they should; be assured that the actual solution described there are cubic and quartic, respectively.*

we will refine in the same way as before, but we now have a choice as to whether we h- or p-refine.

As discussed in Section 4.4, we can see exponential convergence rates for solutions that are sufficiently smooth, and we have seen in Section 4.3 that we can achieve polynomial convergence rates. If p-adaptivity has higher convergence rates then why do we need to combine the two in the first place?

If the refinement of the mesh is not strong enough, then the exponential part of error reduction (appearing from p-adaption) cannot appear [18, p. 604]. We therefore may need to refine the mesh before increasing the polynomial degree, and we ought to make this decision depending upon how 'smooth' the solution is; to detect whether the solution is 'smooth', we introduce a so-called smoothness indicator as done in [42, p. 2733] (which is rewritten in more

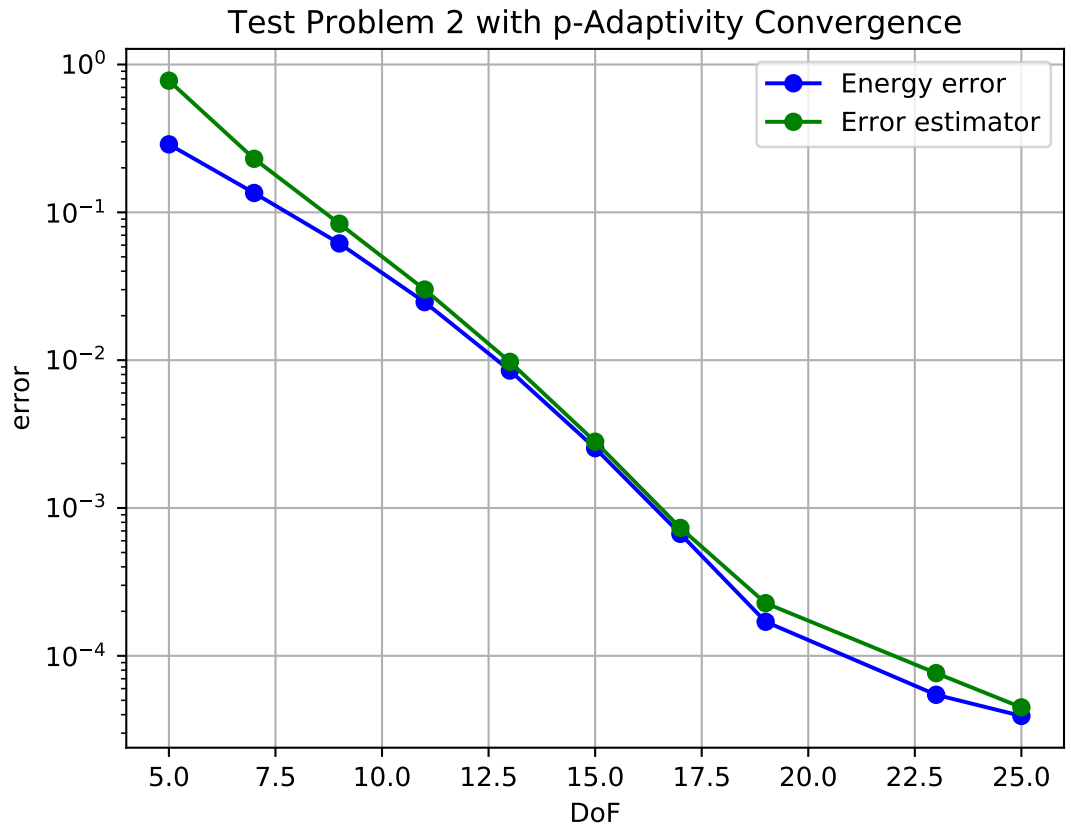


Figure 4.14: *p-adaptivity convergence on Problem 4.2.*

familiar notation Equation (4.10)) for $u \in H^1(K)$, where $K \subseteq \Omega$ is the part of domain denoted by a single element, κ.

$$\mathcal{F}_K[u] := \begin{cases} \|u\|^2_{\infty(K)} \left[\coth(1) \left(h_K^{-1} \|u\|^2_{L^2(K)} + h_K |u|^2_{H^1(K)}\right)\right]^{-1} & \text{if } u \not\equiv 0 \\ 1 & \text{if } u \equiv 0 \end{cases} \tag{4.10}$$

Combining algorithms 4.2 and 4.3 with our smoothness indicator we get Algorithm 4.4, which is effectively the same as [42, algorithm 2] (by taking $\tau = 0.5$).

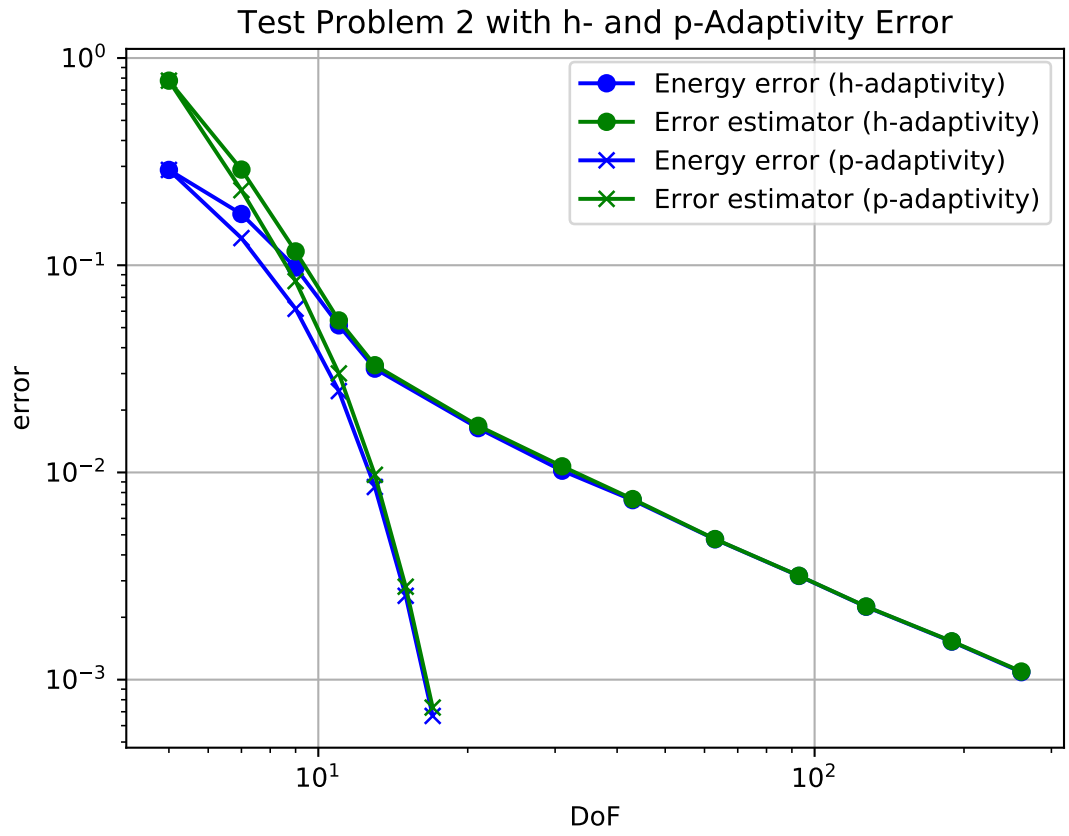


Figure 4.15: *h- and p-adaptivity convergence on Problem 4.2.*

4.5.1 Test Problem 1

For Problem 4.1, we again start with 4 linear elements as an initial mesh. Running the hp-adaptive algorithm for the first 3 steps gives Figure 4.18.

We notice that approximation converges very quickly to the true solution, even after just a few hp-adaptive steps. This is shown especially well in Figure 4.19, which shows the convergence rate of the hp-algorithm after 8 hp-adaptive steps, and very clearly shows exponential convergence rates.

We may also produce a graph showing the sizes and polynomial degrees for each element after 8 hp-adaptive steps, which is shown in Figure 4.20. This shows that the algorithm, from the initial condition, chose to double the number of elements (we now have 8 elements) and increase the polynomial degrees to order 5. This is a very sensible resulting mesh as the true

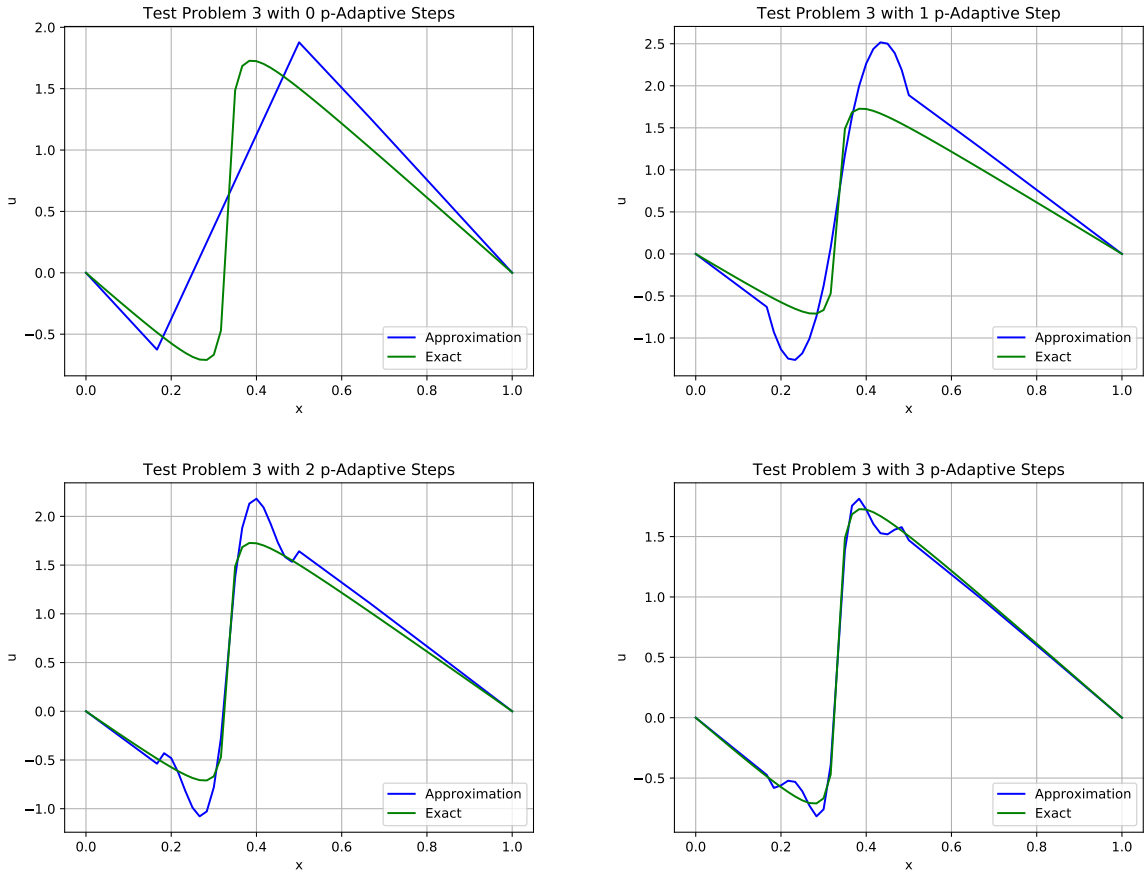


Figure 4.16: *p-adaptivity on Problem 4.3.*

solution is very smooth, upon which we expect higher-order polynomials to approximate the solution better.

4.5.2 Test Problem 2

Problem 4.2 has a smooth solution, but there needs to be sufficient resolution around the boundaries before a viable solution becomes apparent. The hp-adaptive algorithm produces the results in Figure 4.21 for the first 3 hp-adaptive steps.

Wihler [42] approximates a similar problem to Problem 4.2 (where $\epsilon = 10^{-5}$), and produces a plot of the convergence rates and mesh. We produce our version of these results respectively in Figures 4.22 and 4.23, plotting both the results to Problem 4.2 and the modified problem

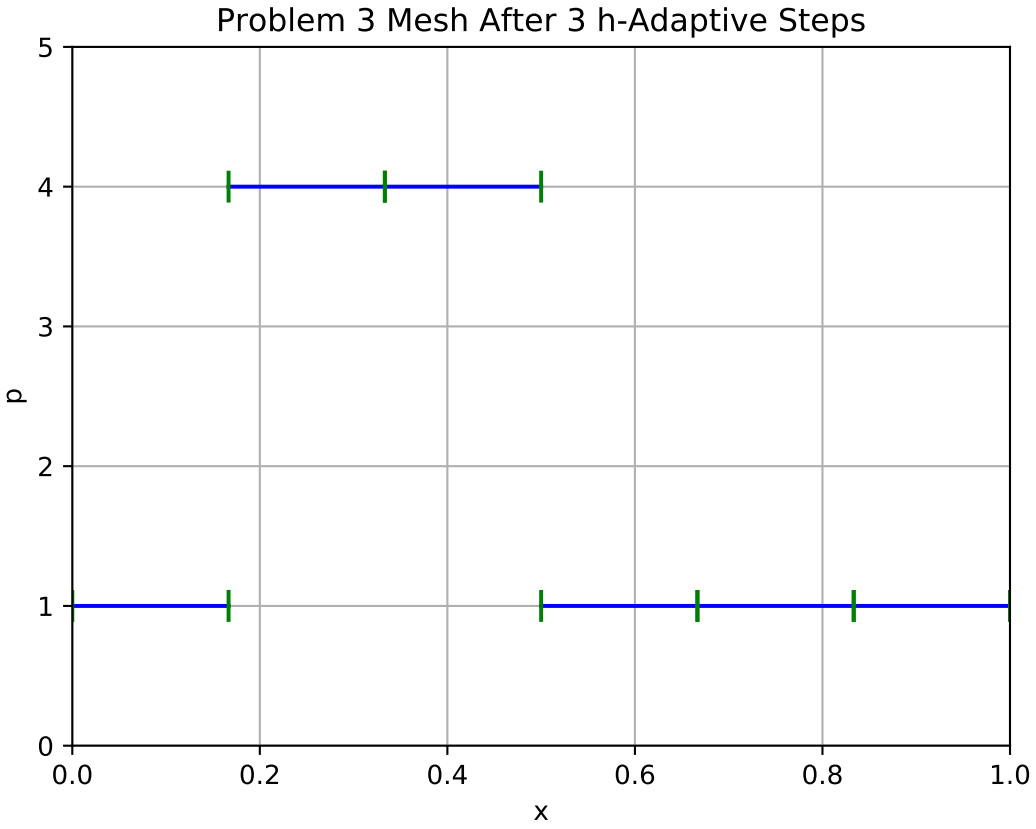


Figure 4.17: *h- and p-adaptivity convergence issues on Problem 4.2.*

given in [42, ex. 2].

First of all, we see that the hp-adaptive strategy for Problem 4.2 yields exponential convergence rates, resulting from the use of p-adaptivity. We notice that there are sometimes lapses in the efficiency of the error estimator, but it otherwise sits efficiently just above the actual error. However there is a problem for when the graph goes beyond 70 degrees of freedom where the energy error is higher than the error indicator; this is likely due to similar reasons as discussed in Section 4.4 when the convergence graph had a similar problem. In fact, by running Blakey FEM with the correct parameters for this problem, we see that the individual error indicators on each element are in the order of 1×10^{-15}, which is in the order of machine precision in this C++ implementation. Therefore the likely cause for this is not a fault in the mathematical analysis, but more likely a rounding error.

The mesh shown for Problem 4.2 also shows reasonable results: the elements are smaller

Algorithm 4.4: p-refinement

Input : $\tau_h, u_h, \boldsymbol{\eta_\kappa}$
Output: $\tau_h^{\text{new}}, u_h^{\text{new}}$
forall κ **do**
  if $\eta_\kappa \geq \max \boldsymbol{\eta_\kappa}/3$ **then**
    if $\mathcal{F}_K \geq 0.5$ **then**
      Increase polynomial degree on element;
    else
      Split element in half and add both these elements to τ_h^{new}

in size near the boundaries. However it is interesting that the polynomial degree is so high over the plateau region. More investigation would need to take place to determine why this happened, as one would expect large linear elements would be sufficient for this problem.

We may also compare our results with Wihler's results for the modified problem to Problem 4.2 where $\epsilon = 10^{-5}$. We notice that there are some very similar features between the two convergence rates: in particular we see that there is a large gap between the error and the estimator while the degrees of freedom are below 10, but this gap closes as the degrees of freedom increase. We also notice that both convergence graphs yield exponential convergence rates towards the true solution. However, Wihler's results appear to indicate that the solution was found within an accuracy of 1×10^{-7} with 78 degrees of freedom — but our algorithm takes around 83 degrees of freedom to achieve this, despite using the same smoothness indicators, error indicators, and problem parameters. There is likely to be some small discrepancy in the specific numerics of each implementation, but we can be can be reassured by the results having the same qualitative behaviour.

Comparing the meshes between our results and Wihler's results show roughly the same behaviour — in that larger elements with lower-order polynomials appear in the centre of the domain, and smaller elements with higher-order polynomials appear at the boundaries. However the polynomial degrees in outer boundary elements do not match between our results and Wihler's. This could be an issue in how the boundary conditions have been applied, and could also explain the slightly higher degrees of freedom needed to reduce the error in our results.

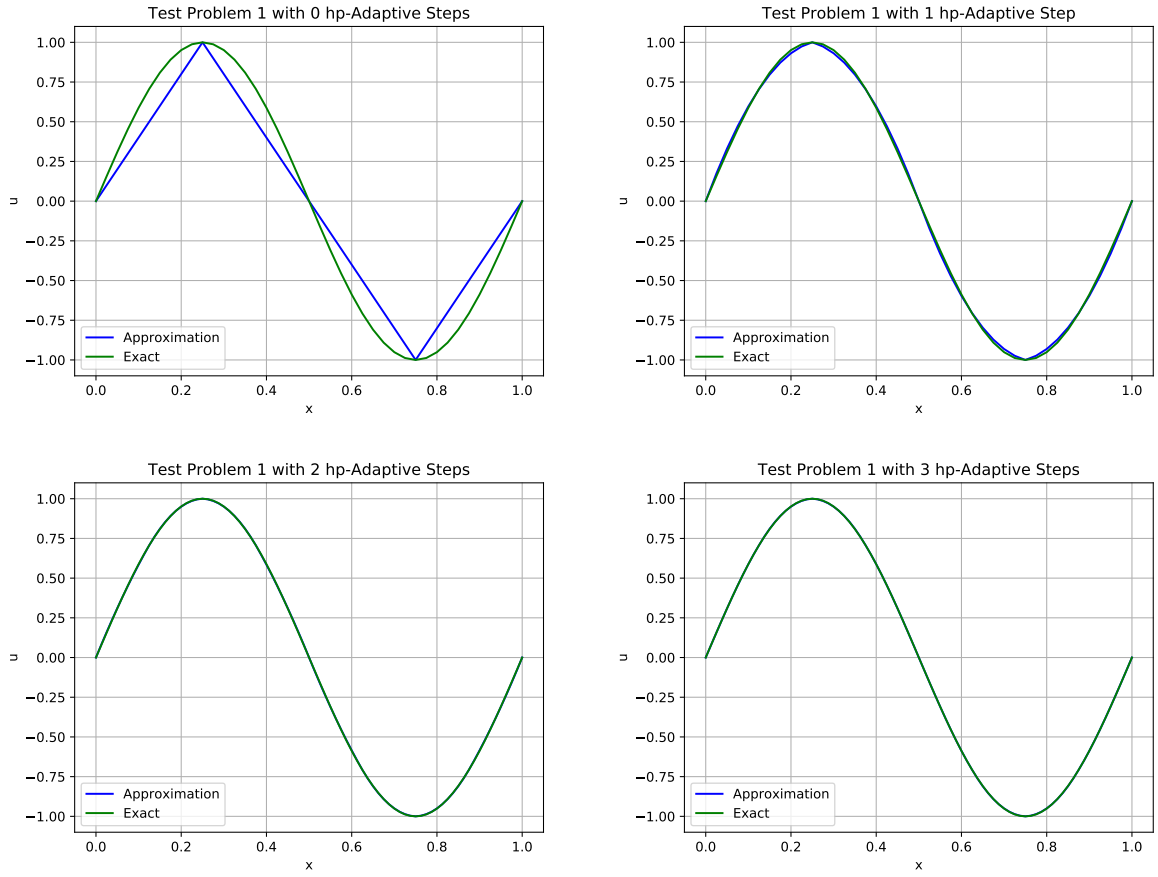


Figure 4.18: *hp-adaptivity on Problem 4.1.*

We can, however, be satisfied that our results do yield exponential convergence rates towards the exact solution for both of these variations on Problem 4.2.

4.5.3 Test Problem 3

Problem 4.3 is designed to solve the same problem as [42, ex. 4], so we can make some direct comparisons between our results and the results of Wihler. Figure 4.24 shows the hp-adaptive algorithm for the first 3 steps, and Figures 4.25 and 4.26 respectively show the convergence rate and resulting mesh after 20 steps. For these plots we have chosen an initial mesh of 6 linear elements.

Figure 4.24 clearly shows that hp-adaptivity is helping the approximation approach the solution step-by-step, and Figure 4.25 shows that the solution is converging exponentially. Comparing the convergence plot with Wihler's plot in [42, fig. 5], we first note that our initial meshes

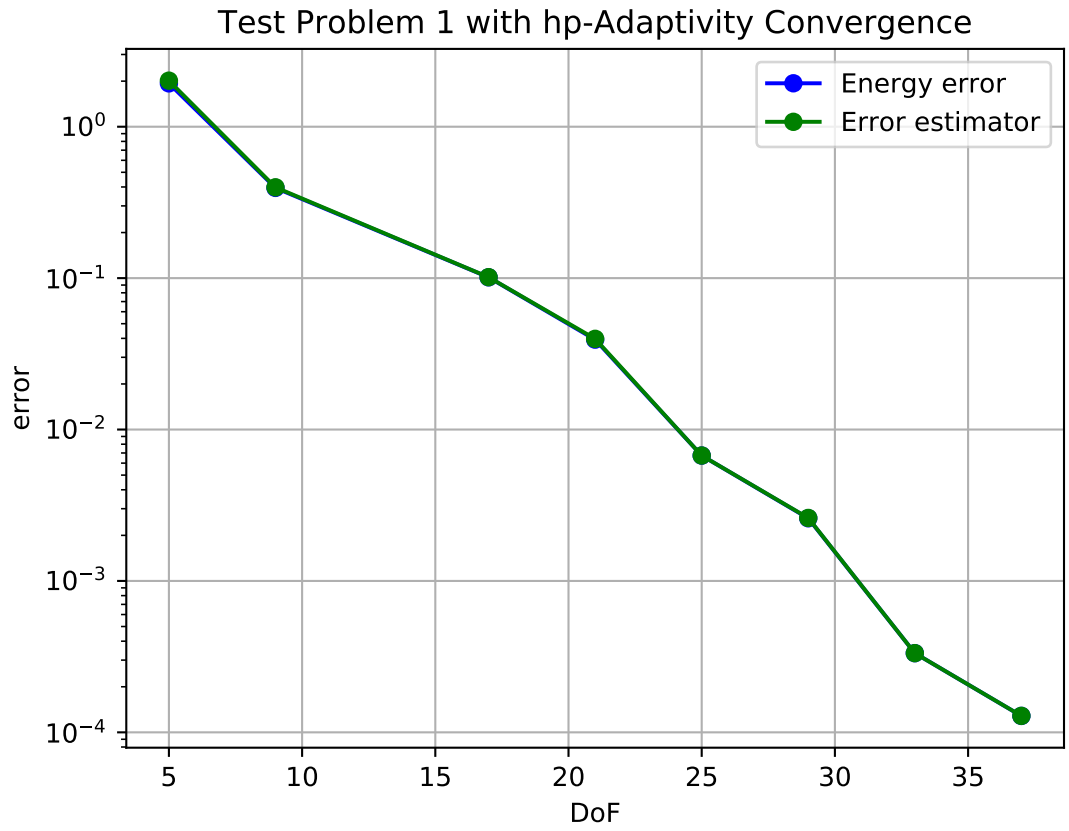


Figure 4.19: *hp-adaptivity convergence on Problem 4.1.*

are different: ours consists of 6 linear elements, and Wihler's consists of 4 linear elements. Despite this, the convergence rates have roughly the same features. We actually notice that at around 100 degrees of freedom, our error is calculated at just under 1×10^{-4} and Wihler's error is calculated at somewhere between 1×10^{-3} and 1×10^{-4}. This actually shows that our solution has a smaller error for fewer degrees of freedom than Wihler's but, just like in Section 4.5, this could be due to numerical errors or specific implementation features. It could also be due to the difference in initial condition.

The mesh shown in Figure 4.26 is very similar to the mesh shown in [42, fig. 5]: larger elements of degrees 4–7 appear near the boundaries, and smaller elements appear around the shock region with high polynomial degrees around the point and low polynomial degrees in the centre of the shock region.

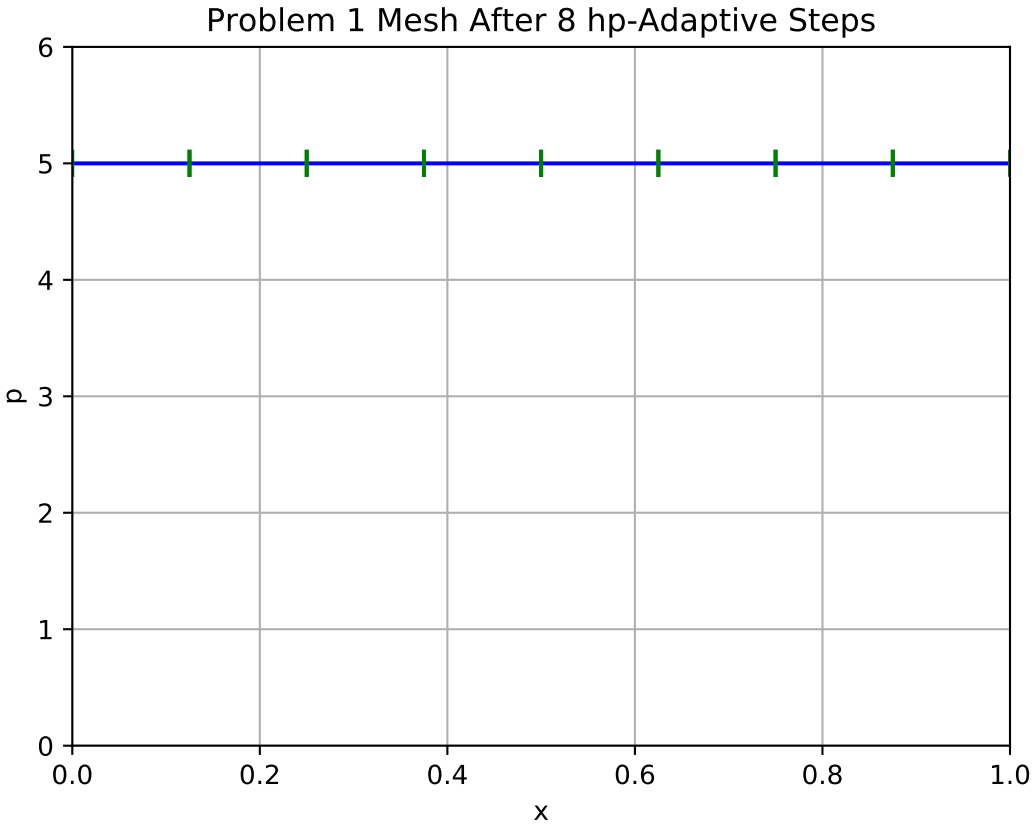


Figure 4.20: *hp-adaptivity element sizes and polynomial degrees on Problem 4.1.*

4.6 Results Summary

Table 4.2 illustrates the performance and effectiveness of each algorithm on each of the different problems. We note here that the h (global) and h entries for Problem 3 are accurate to only 1×10^{-2} but were terminated early because of a high number of degrees of freedom, and are indicated by the braces (); the entries for the same problem for p (global) and p are accurate to only 1×10^{-1} for similar reasons, and are indicated with curly braces {}.

Problem	h (global)	h	p (global)	p	hp
4.1	11 [8193]	14 [1677]	4 [21]	4 [21]	6 [33]
4.2	9 [2049]	13 [397]	6 [29]	4 [17]	7 [27]
4.3	(13 [32769])	(19 [2131])	{14 [91]}	{14 [35]}	10 [64]

Table 4.2: *For each problem, shows the number of iterations [and degrees of freedom in brackets] for each algorithm to take the approximation within 1×10^{-3} of the exact solution in the energy norm, with the exception of those in braces and curly braces.*

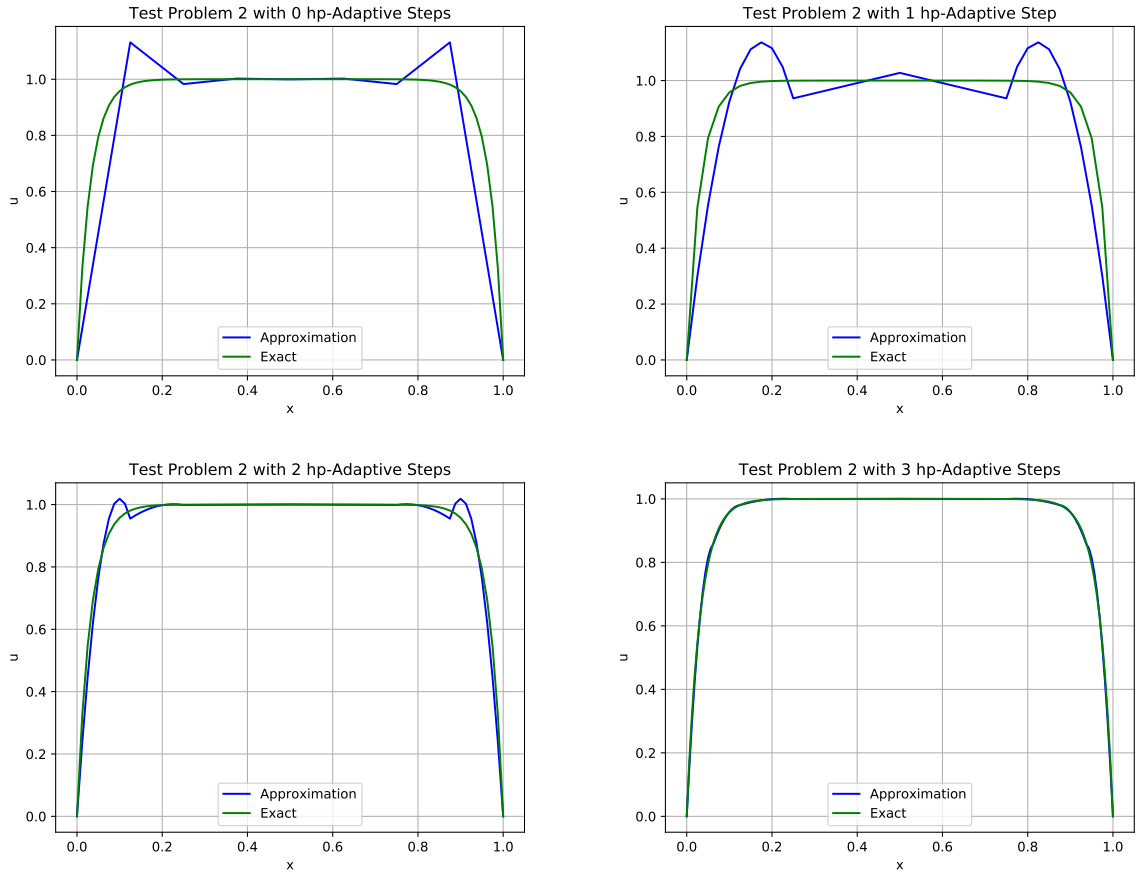


Figure 4.21: *hp-adaptivity on Problem 4.2.*

For each of the test problems, we can see that h-adaptivity certainly has at least some advantage over global h-refinement (illustrated by Figures 4.3, 4.5, and 4.8). We can see, therefore, that our error indicators are functioning as we'd like and that they are suitable for adaptive algorithms. In particular, Problem 4.2 had achieved a much lower error in the energy norm with fewer degrees of freedom due to the algorithm only refining the mesh near the boundaries, and not in the centre of the domain where it is not needed.

We notice similar results for p-adaptivity, where the adaptivity has had at least some sort of advantage, as shown in Figures 4.10, 4.14, and 4.17. In particular, Problem 4.1 benefited hugely from using p-adaptivity.

For hp-adaptivity, we see (with the exception of Problem 4.1) an advantage of using hp-adaptivity over exclusive h or p global refinement. In fact, for Problem 4.3, the degrees of freedom for a lower error were reduced by a factor of roughly 500, which shows the power of

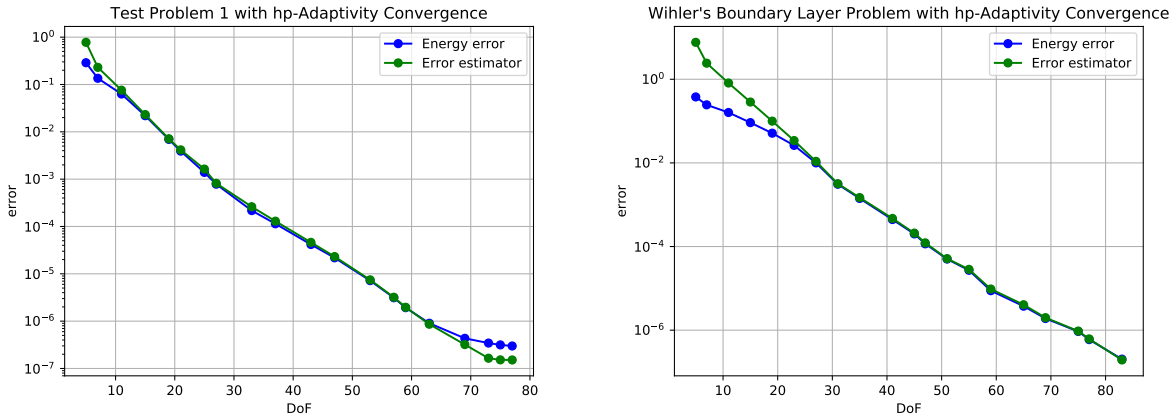


Figure 4.22: *hp-adaptivity convergence on Problem 4.2 (left) and Wihler's problem (right).*

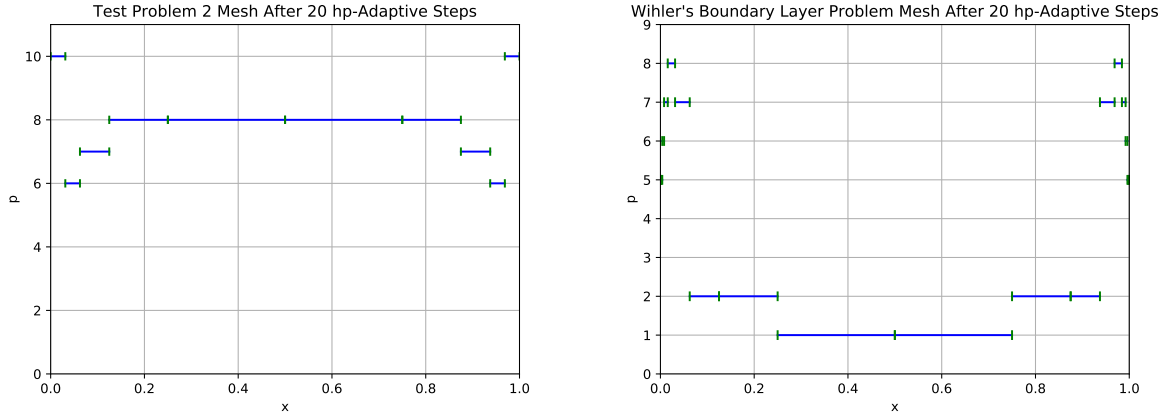


Figure 4.23: *hp-adaptivity element sizes and polynomial degrees on Problem 4.2 (left) and Wihler's problem (right).*

hp-adaptive algorithms.

The avid reader may notice that these three examples were chosen specifically to highlight the types of solutions that may benefit from the different adaptivity techniques. In particular, we notice that Problem 4.1 performed particularly well with p-adaptivity (likely due to the high smoothness), Problem 4.2 performed particularly well with h-adaptivity (likely due to the boundaries), and Problem 4.3 performed particularly well with hp-adaptivity (likely due to a mixture of the small shock region and the solution's overall smoothness).

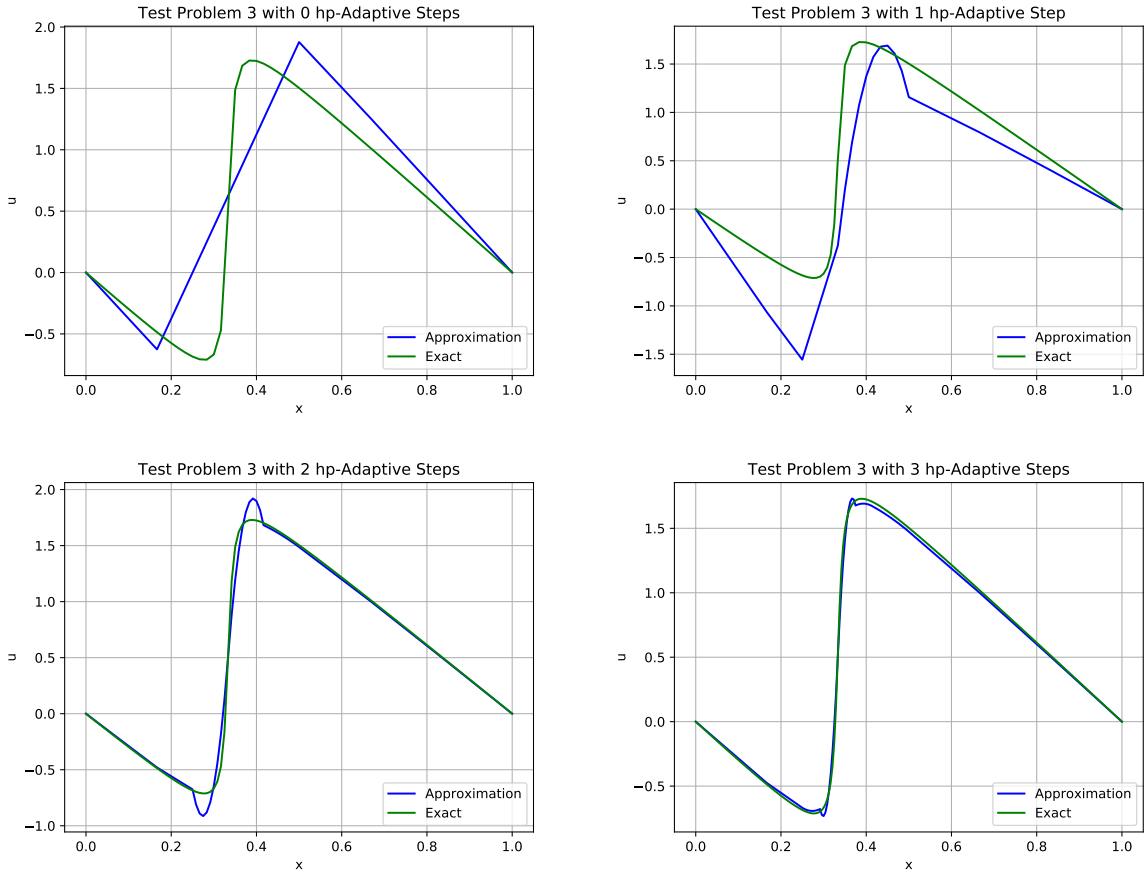


Figure 4.24: *hp-adaptivity on Problem 4.3.*

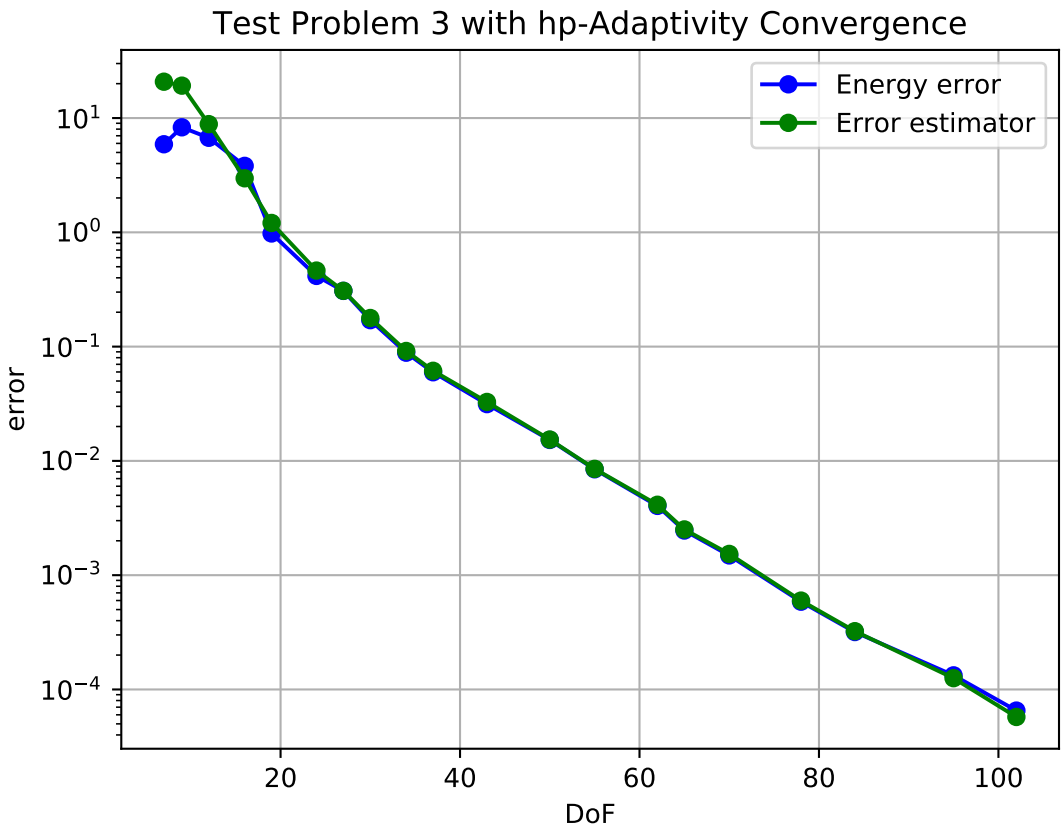


Figure 4.25: *hp-adaptivity element sizes and polynomial degrees on Problem 4.3.*

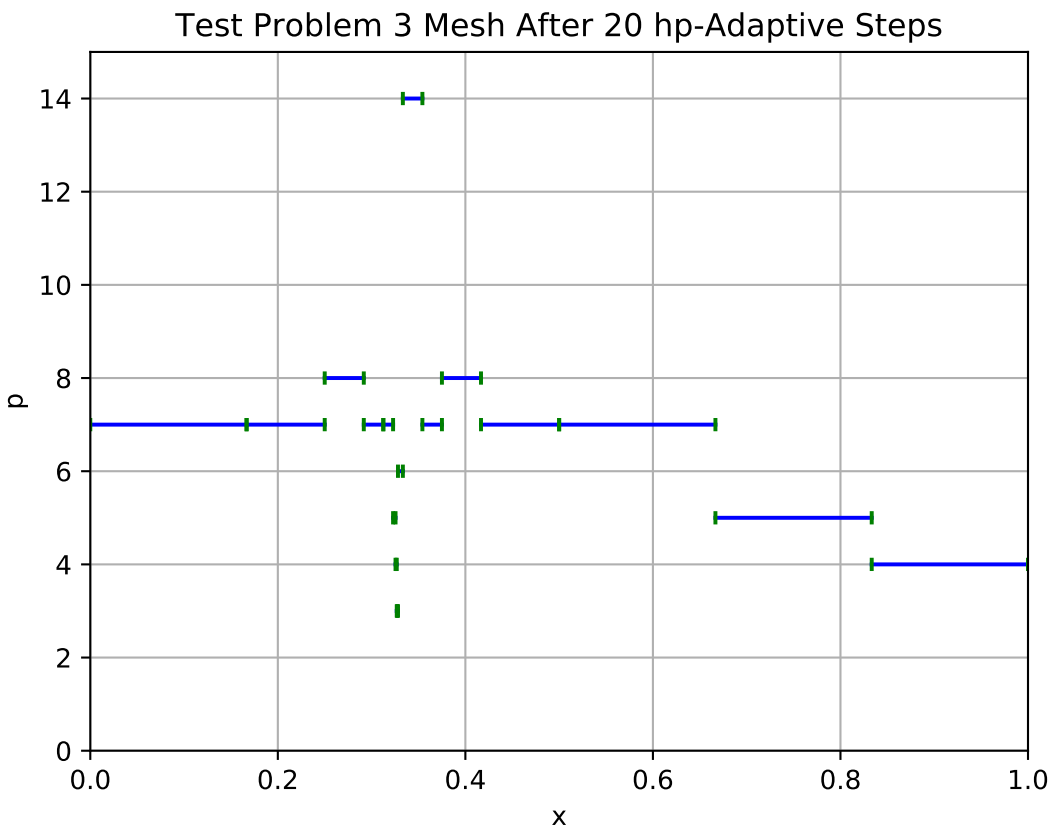


Figure 4.26: *hp-adaptivity element sizes and polynomial degrees on Problem 4.3.*

Section 5

Nonlinear Problems

The problem given in Section 2.2.1 is very clearly a linear differential equation; we can also consider a different model problem that is nonlinear, as given in Equation (5.1), and follow a similar procedure as before to derive the weak formulation of this problem. Similarly to the previous problem, for a given bounded Lipschitz domain $\Omega \subseteq \mathbb{R}^d, d \geq 1$, we seek u such that

$$-\epsilon \Delta u = f(x, u(x)), x \in \Omega \tag{5.1a}$$

$$u = 0, \text{ on } \partial\Omega \tag{5.1b}$$

Here, $\epsilon > 0$ and $f : \Omega \times \mathbb{R} \to \mathbb{R}$ is continuously differentiable.

For this problem we will seek u in the function space $H_0^1(\Omega) =: V$.

Multiplying Equation (5.1a) by a test function, $v \in H_0^1(\Omega)$ to give

$$-\epsilon \Delta u v = fv, \ \forall v \in H_0^1(\Omega).$$

Integrating over the domain and performing integration by parts as done previously we need to find $u \in H_0^1(\Omega)$ such that

$$\epsilon(\nabla u, \nabla v) = (f(u), v), \ \forall v \in H_0^1(\Omega), \tag{5.2}$$

where we have suppressed the dependence of x in f for simplicity. Notice that this problem is almost the same as the problem given in Section 2.2.1 except we have implicitly moved the "(cu, v)" into f and now allowed f to depend upon u.

Our finite element method is: $u_{h,p} \in V_h \subset H_0^1(\Omega)$ such that

$$\epsilon(\nabla u_{h,p}, \nabla v_h) = (f(u_{h,p}), v_h),\ \forall v_h \in V_h. \tag{5.3}$$

Note that we cannot use Lax-Milgram for proving existence and uniqueness of the nonlinear problem; in fact this report will not concern itself with proving the existence or uniqueness of this nonlinear problem and instead supposes that at least one solution exists, just as Amrein et. al [3]. The report instead relies on results such as those in [28] for checking final solutions.

Writing $X = H_0^1(\Omega)$, we denote $X^{-1} = H^{-1}(\Omega)$ as the dual space of X (cf. [7, p. 219]). Hence, we may define the map $F_\epsilon : X \to X^{-1}$ by

$$\langle F_\epsilon(u), v\rangle := (\epsilon\nabla u, \nabla v) - (f(u), v), \forall v \in X,$$

where $\langle\cdot,\cdot\rangle$ is the dual product in $X^{-1} \times X$.

Hence, the weak formulation may be written in the equivalent nonlinear operator form: Find $u \in X$ such that

$$F_\epsilon(u) = 0. \tag{5.4}$$

We define the ϵ-norm as:

$$\|u\|_\epsilon := \left(\|\nabla u\|^2_{L^2(\Omega)} + \|u\|^2_{L^2(\Omega)}\right)^{\frac{1}{2}}.$$

Newton's method seeks to compute zeros such that Equation (5.4) is satisfied. Assuming that the Fréchet derivative of F_ϵ, F'_ϵ exists, then the Newton's method is given by: for an initial

guess, u_0, we generate

$$u^{n+1} = u^n + \Delta u^n, n \geq 0,$$

where each update, Δu^n, satisfies

$$F'_\epsilon(u^n)\Delta u^n = -F_\epsilon(u^n), n \geq 0.$$

Newton's method is not very reliable when the initial guess is far away, so we introduce a dampening parameter, $\theta_n \in [0, 1]$:

$$u^{n+1} = u^n - \theta_n \Delta u^n, n \geq 0,$$

where θ_n may be chosen according to [3, sec. 2.2]. We have chosen the value in a similar way, except we choose the ϵ-norm in the calculation of the parameter:

$$\theta_n = \sqrt{2\tau \|F_\epsilon(u_n)\|_\epsilon^{-1}},$$

where $\tau > 0$ is the tolerance in which we hope to solve subsequent steps to within Newton's method.

For our particular model problem in Equation (5.1), the Fréchet derivative of F_ϵ is given by

$$\langle F'_\epsilon(u)w, v\rangle = \int_\Omega \epsilon w'v'\,dx - \int_\Omega f'(u)wv\,dx.$$

Hence, given u_0, Newton's method is: Find $u^{n+1} \in X$ such that

$$F'_\epsilon(u^n)(u^{n+1} - u_n) = -\theta_n F_\epsilon(u^n). \tag{5.5}$$

Akin to the linear model problem, we may introduce two functionals to help us write this

more concisely:

$$a_\epsilon(u^n; u^{n+1}, v) = a_\epsilon(u^n; u^n, v) - \theta_n l_\epsilon(u^n; v), \forall v \in X, \tag{5.6}$$

where

$$a_\epsilon(u; w, v) := \int_\Omega (\epsilon w'v' - f'(u)wv)\ dx$$

and

$$l_\epsilon(u; v) := \int_\Omega (\epsilon u'v' - f(u)v)\ dx.$$

We can use a_ϵ and l_ϵ in an analogous way to the linear problem to compute the Newton update at each Newton step.

5.1 Simple Numerics

5.1.1 Test Problem 1 (Nonlinear)

Problem 5.1.

The Bratu problem in one dimension has, depending upon the bifurcation parameter ϵ, somewhere between zero and two solutions.

The data is set on Equation (5.1) *as* $f = \exp(u)$, $c \equiv 0$, *and we will allow* ϵ *to vary slightly. We note from [28, p. 27] that the critical value of the bifurcation parameter is* $\epsilon_c \approx 1/3.514$.

Provided that our bifurcation parameter $\epsilon > \epsilon_c$, the problem has two solutions — and this is the case that we will consider.

We will solve this nonlinear problem with 20 elements, from which we will vary the bifurcation parameter and initial condition. As suggested by Mohsen [28, p. 28], we will set the initial

condition for each simulation to be

$$u_0(x) = a\,\sin(\pi x).$$

We show the results for various initial conditions and bifurcation parameters, producing Figure 5.1.

We note that, after comparing with the maximum values of the solution given by Mohsen [28, p. 29], that the values appear to be correct.

5.1.2 Test Problem 2 (Nonlinear)

For this nonlinear test problem, we will actually choose a linear problem to make sure that the solver works in this degenerative case. We will take our new problem as the same as the boundary layer problem, given as Problem 4.2. For the parameters for our new model equation, we give these in Problem 5.2.

Problem 5.2.

A boundary layer problem, exhibiting boundaries near $x = 0$ and $x = 1$, as given in [42, ex. 2] with the exact solution

$$u(x) = -\frac{\exp(x/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} - \frac{\exp(-x/\sqrt{\epsilon})\exp(1/\sqrt{\epsilon})}{\exp(1/\sqrt{\epsilon}) + 1} + 1.$$

The data is set on Equation (5.1) *as $\epsilon = 10^{-3}$, $f \equiv -1 - u$.*

Running with the correct parameters produces Figure 5.2. We see that the features of the solution are present, very similarly to the solutions shown in Section 4 which suggests that our nonlinear solver works for the degenerative linear case.

5.1.3 Test Problem 3 (Nonlinear)

Problem 5.3.

Here we will solve the steady one-dimensional version of Fisher's equation with zero boundary conditions. The problem infinitely many solutions [3, p. 1652].

The data is set on Equation (5.1) *as* $\epsilon = 0.00025$, $f = u(u-1)$.

We choose a similar initial condition to Amrein et. al [3, fig. 3] that consists of four plateaued peaks at $u_0 = 1$, and three plateaued troughs at $u_0 = -0.4$. From this initial data we produce Figure 5.3.

The figure displays the same attributes as those in [3] so that we can be fairly sure that this works. Although we aren't studying here the convergence or refinement properties of the approximation, we note that the Newton solver converged within an l^2-norm tolerance of 1×10^{-3} between subsequent terms within 30 steps.

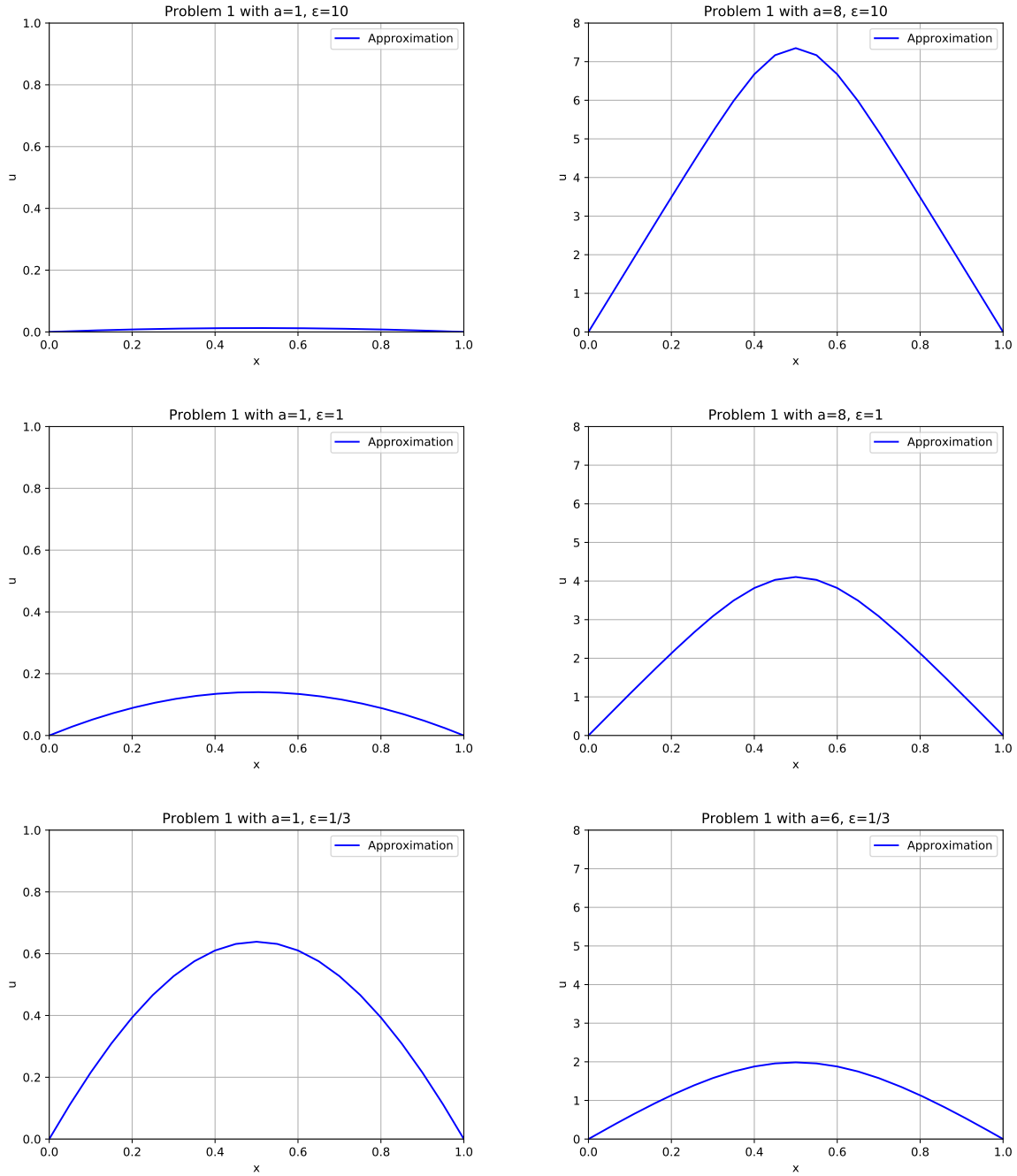


Figure 5.1: *Solutions of Problem 5.1. Lower solutions are displayed on the left and upper solutions are displayed on the right.*

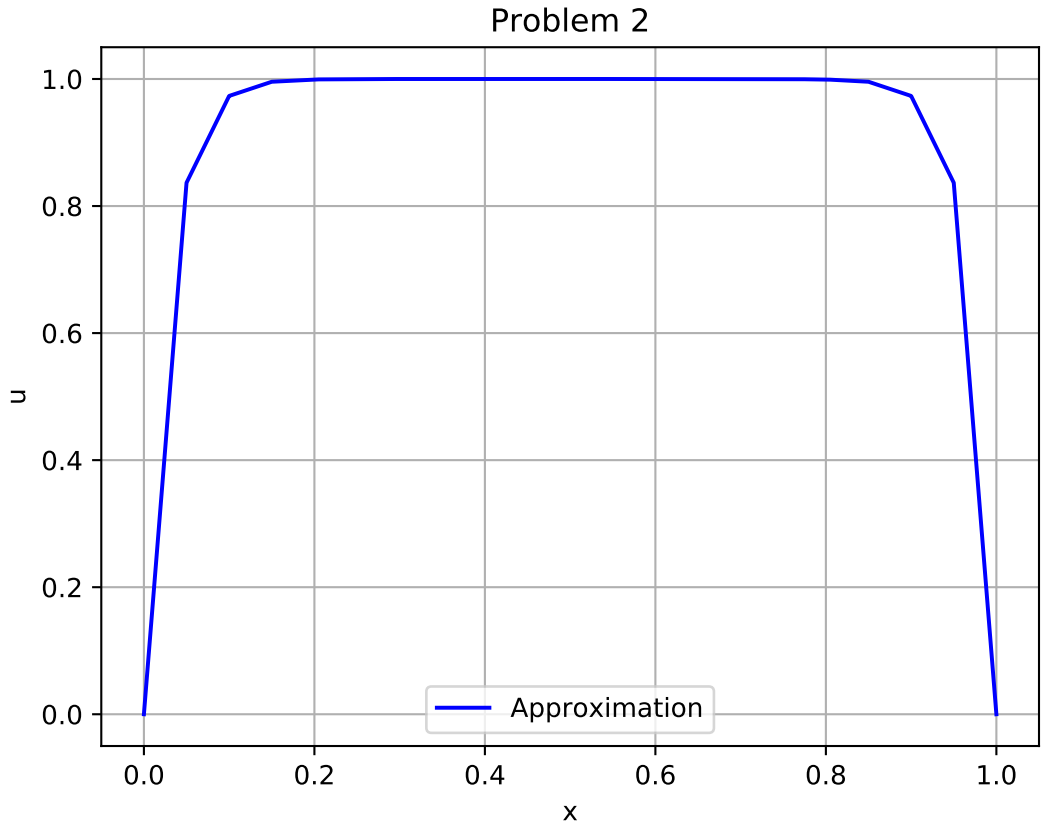


Figure 5.2: *Solution of Problem 5.2.*

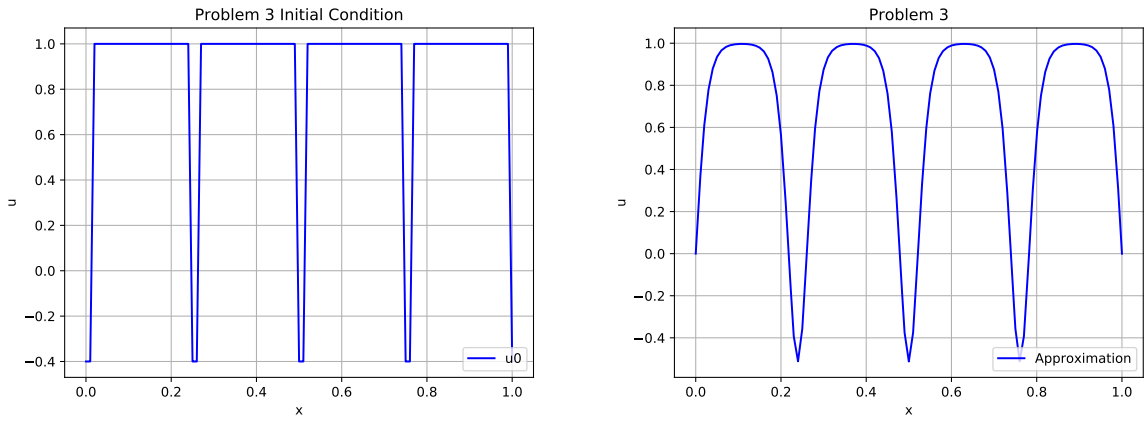


Figure 5.3: *Initial condition (left) and solution (right) of Problem 5.3.*

Section 6

Conclusions

In conclusion, we found that an hp-adaptive algorithm for finite element methods can greatly reduce the degrees of freedom for high-accuracy results, especially when smooth functions with steep derivatives are involved. This considerably reduced computational resources over the global refinement strategy that would be necessary for the same accuracy. We did find some situations where a p-adaptive algorithm yielded higher convergence rates between the degrees of freedom and error, but the hp-adaptive was the most consistent in doing so.

We started in Section 2 by laying the background in hp-FEMs, creating a linear model problem, and proving that the chosen problem was well-posed.

We outlined the specific implementation choices in Section 3, which were heavily influenced by the work of Šolín et. al [33]; the implementation focused on creating an efficient piece of software: the efficiency was particularly increased by the use of an intelligent cache with the quadrature calculations and an efficient class structure. Further discussions were made at how the code could be further optimised with, for example, parallelising the linear system solving process and implementing a sparse matrix data structure. In particular we note that we used a non-parallelisable iterative conjugate gradient method for solving all linear systems that arose, which significantly restricts the available computing resources available on most modern-day computers; we note that a parallelisable version of the conjugate gradient method could be implemented, as suggested by Hestenes et. al [20]. We also note that the code was written in a generic way, allowing for further adaptability and extension of features if future developers were to continue this project.

A posteriori error bounds were derived and proved for a linear problem in Section 4, influenced by the work of Schwab [32]. We further derived local element indicators leading to

robust and efficient adaptivity algorithms, and made use of a smoothness parameter that was used in the work of Wihler [42]. We solved some numerical calculations for three chosen test problems; we found that the h- and p-adaptive algorithms worked with varying success depending upon the specific test problem, but the hp-adaptive algorithm consistently performed well and yielded high (exponential) rates of convergence. In nearly all cases, global refinement of h and p separately were very costly and unnecessary.

Section 5 introduced a new model problem that was nonlinear (specifically semilinear). This new model problem was implemented into the code, using inheritance to reduce code redundancy from the linear solving process. In particular we used a Newton solver to solve the resulting nonlinear system. For this model problem we chose a further three test problems to be solved, for which the results appeared to coincide with the results of other authors performing the same calculations [28, 3].

Further work on this project could involve generalising the implementation to work with dimensions higher than one, which would benefit most real-world problems; our implementation is concerned only with one-dimensional PDEs, but is designed in a way that would easily support higher-dimensional PDEs in the future. We could also work toward implementing the a posteriori error bounds and using them to give hp-adaptive algorithms to nonlinear model problems, like the work demonstrated by Amrein et. al [3].

Appendix A

Code

Note that all of the code for this project can be found on the GitHub repository, as shown on Page 2. We will specifically list and discuss the code here for the `Solve` method of the `Solution_linear` class, to give an idea of the process involved in the solvers. We also include the calculation of the value of the bilinear functional, $a(u, v)$, and linear functional $l(v)$.

```
1  void Solution_linear::Solve(const double &a_cgTolerance)
2  {
3      // Left and right boundary conditions.
4      double A = 0;
5      double B = 0;
6
7      // Degrees of freedom and elements pointer.
8      int n = this->mesh->elements->get_DoF();
9      Elements* elements = this->mesh->elements;
10
11     // Stiffness and matrix and load vector for the FEM sovler.
12     Matrix_full<double> stiffnessMatrix(n, n, 0);
13     std::vector<double> loadVector(n, 0);
14
15     // Loops over all elements.
16     for (int elementCounter=0; elementCounter<this->noElements;
       ↪  ++elementCounter)
17     {
```

```
18          // Pointer to our current element.
19          Element* currentElement =
            ↪   (*(this->mesh->elements))[elementCounter];
20
21          // All degrees of freedom associated with this element.
22          std::vector<int> elementDoFs =
            ↪   elements->get_elementDoFs(elementCounter);
23
24          // Loops over first combination of basis functions.
25          for (int a=0; a<elementDoFs.size(); ++a)
26          {
27              // Current first degree of freedom.
28              int j = elementDoFs[a];
29
30              // Basis functions needed.
31              f_double basis = currentElement->basisFunction(a, 0);
32
33              // Adds to load vector.
34              loadVector[j] += this->l(currentElement, basis);
35
36              // Loops over second combination of basis functions.
37              for (int b=0; b<elementDoFs.size(); ++b)
38              {
39                  // Current second degree of freedom.
40                  int i = elementDoFs[b];
41
42                  // Basis functions needed.
43                  f_double basis1  = currentElement->basisFunction(b, 0);
44                  f_double basis2  = currentElement->basisFunction(a, 0);
```

```
45              f_double basis1_ = currentElement->basisFunction(b, 1);
46              f_double basis2_ = currentElement->basisFunction(a, 1);
47
48              // Adds to stiffness matrix.
49              double value = stiffnessMatrix(i, j);
50              stiffnessMatrix.set(i, j, value +
                ↪  this->a(currentElement, basis1, basis2, basis1_,
                ↪  basis2_));
51          }
52      }
53  }
54
55  // Temporary load vector and boundary element contribution.
56  std::vector<double> F_(n);
57  std::vector<double> u0(n, 0);
58
59  // The degree of freedom to apply the second boundary condition at.
60  int m = this->mesh->elements->get_noElements();
61
62  // Zeroes rows and columns associated with first boundary condition.
63  for (int i=0; i<stiffnessMatrix.get_noRows(); ++i)
64      stiffnessMatrix.set(0, i, 0);
65  for (int j=0; j<stiffnessMatrix.get_noColumns(); ++j)
66      stiffnessMatrix.set(j, 0, 0);
67  loadVector[0] = 0;
68
69  // Zeroes rows and columns associated with second boundary
    ↪  condition.
70  for (int i=0; i<stiffnessMatrix.get_noRows(); ++i)
```

```
71          stiffnessMatrix.set(m, i, 0);
72      for (int j=0; j<stiffnessMatrix.get_noColumns(); ++j)
73          stiffnessMatrix.set(j, m, 0);
74      loadVector[m] = 0;
75
76      // Enforces boundary condition on contribution vector.
77      u0[0] = A;
78      u0[m] = B;
79
80      // Removes contribution from load vector.
81      F_ = stiffnessMatrix*u0;
82      for (int i=0; i<n; ++i)
83          loadVector[i] -= F_[i];
84
85      // Enforces boundary condition in the stiffness matrix.
86      stiffnessMatrix.set(0, 0, 1);
87      stiffnessMatrix.set(m, m, 1);
88
89      // Calculates and stores the solution to a specified conjugate
        ↪ gradient tolerance.
90      this->solution = linearSystems::conjugateGradient(stiffnessMatrix,
        ↪ loadVector, a_cgTolerance);
91
92      // Re-enforces the boundary conditions.
93      this->solution[0] = A;
94      this->solution[m] = B;
95  }
96
97  double Solution_linear::l(Element* currentElement, f_double &basis)
```

```
98  {
99      // Jacobian for this element.
100     double J = currentElement->get_Jacobian();
101
102     // Initialises return value.
103     double integral = 0;
104
105     // Gets element quadrature.
106     std::vector<double> coordinates;
107     std::vector<double> weights;
108     currentElement->get_elementQuadrature(coordinates, weights);
109
110     // Loops over all coordiantes and weights.
111     for (int k=0; k<coordinates.size(); ++k)
112     {
113         // Basis function and f values at this coordinate.
114         double b_value = basis(coordinates[k]);
115         double f_value =
            ↪  this->f(currentElement->mapLocalToGlobal(coordinates[k]));
116
117         // Adds these combinations to the return value.
118         integral += b_value*f_value*weights[k]*J;
119     }
120
121     // Returns value.
122     return integral;
123 }
124 double Solution_linear::a(Element* currentElement, f_double &basis1,
    ↪  f_double &basis2, f_double &basis1_, f_double &basis2_)
```

```
125 {
126     // Jacobian for this element.
127     double J = currentElement->get_Jacobian();
128 
129     // Initialises return value.
130     double integral = 0;
131 
132     // Gets element quadrature.
133     std::vector<double> coordinates;
134     std::vector<double> weights;
135     currentElement->get_elementQuadrature(coordinates, weights);
136 
137     // Loops over all coordinates and weights for first term in
        ↪ equation.
138     for (int k=0; k<coordinates.size(); ++k)
139     {
140         // Combination of basis functions at this coordinate.
141         double b_value = basis1_(coordinates[k]) *
            ↪ basis2_(coordinates[k]);
142 
143         // Adds to the return value.
144         integral += this->epsilon*b_value*weights[k]/J;
145     }
146 
147     // Loops over all coordinates and weights for second term in
        ↪ equation.
148     for (int k=0; k<coordinates.size(); ++k)
149     {
150         // Basis and c values at this coordinate.
```

```
151         double b_value = basis1(coordinates[k]) *
            ↪ basis2(coordinates[k]);
152         double c_value =
            ↪ this->c(currentElement->mapLocalToGlobal(coordinates[k]));
153
154         // Adds to the return value.
155         integral += c_value*b_value*weights[k]*J;
156     }
157
158     // Returns value.
159     return integral;
160 }
```

The comments on this piece of code are hopefully clear enough to describe the process, but the general idea is:

1. Loop over all elements
    (a) For each element, find the associated degrees of freedom
    (b) Loop over first combination of degrees of freedom
        i. Add value of $l(v)$ (or nonlinear equivalent) to appropriate index in load vector
        ii. Loop over second combination of degrees of freedom
            A. Add value of $a(u, v)$ (or nonlinear equivalent) to appropriate index in stiffness matrix
2. Find contribution of boundary conditions to the solution
3. Remove boundary condition contribution from the load vector
4. Enforce both boundary conditions
5. Solve the resulting linear (or nonlinear) system

Appendix B

References

[1] Object Management Group (OMG). *OMG Unified Modeling Language*. `https://www.omg.org/spec/UML/2.5.1/PDF`. [Online; accessed 07-May-2020]. 2017.

[2] Robert A. Adams. *Sobolev Spaces*. Academic Press, 1975. ISBN: 9780120441501.

[3] Mario Amrein and Thomas P. Wihler. "Fully adaptive Newton-Galerk in methods for semilinear elliptic partial differential equations". In: *SIAM Journal on Scientific Computing* 37.4 (2015), A1637–A1657.

[4] Harbir Antil, Ricardo H. Nochetoo, and Patrick Sodré. "Optimal control of a free boundary problem with surface tension effects: a priori error analysis". In: 53.5 (2015), pp. 2279–2306.

[5] I. Babuska and W.C. Rheinboldt. "Error Estimates for Adaptive Finite Element Computations". In: *SIAM Journal on Numerical Analysis* 15.4 (1978), pp. 736–754.

[6] Amit Bhaya et al. "A cooperative conjugate gradient method for linear systems permitting efficient multi-thread implementation". In: *Computational and Applied Mathematics* 37.2 (2018), pp. 1601–1628. ISSN: 18070302.

[7] Haim Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Springer, 2010. ISBN: 978-0-387-70913-0.

[8] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. 9th ed. Cengage Learning, 2010. ISBN: 9780538733519.

[9] Ward Cheney. *Analysis for Applied Mathematics*. 2001. ISBN: 978-1-4757-3559-8.

[10] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Philadelphia: Society for Industrial and Applied Mathematics, 2002. ISBN: 9780898715149.

[11] Lokenath Debnath and Piotr Mikusiński. *Introduction to Hilbert Spaces with Applications*. Academic Press, 2005. ISBN: 978-0122084386.

[12] James F Epperson. *An introduction to numerical methods and analysis / James F. Epperson.* 2nd ed. Wiley, 2013. ISBN: 1118367596.

[13] Stanley J. Farlow. *An Introduction to Differential Equations and their Applications*. Dover Publications, Inc., 2015. ISBN: 9780486445953.

[14] Stefano Giani and Paul Houston. "Anisotropic hp-adaptive discontinuous Galerkin finite element methods for compressible fluid flows". In: *International Journal of Numerical Analysis and Modeling* 9.4 (2012), pp. 928–949.

[15] Wei Gong, Michael Hinze, and Zhaojie Zhou. "Finite Element Method and A Priori Error Estimates for Dirichlet Boundary Control Problems Governed by Parabolic PDEs". In: *Journal of Scientific Computing* 66.3 (2016), pp. 941–967. DOI: 10.1007/s10915-015-0051-2.

[16] Hauke Gravenkamp, Sundararajan Natarajan, and Wolfgang Dornisch. "On the use of NURBS-based discretizations in the scaled boundary finite element method for wave propagation problems". In: *Computer Methods in Applied Mechanics and Engineering* 315 (2017), pp. 867–880. ISSN: 00457825. DOI: 10.1016/j.cma.2016.11.030.

[17] Hauke Gravenkamp, Albert A. Saputra, and Sascha Duczek. "High-Order Shape Functions in the Scaled Boundary Finite Element Method Revisited". In: *Archives of Computational Methods in Engineering* (2019). ISSN: 18861784. DOI: 10.1007/s11831-019-09385-1.

[18] W. Gui and I. Babuška. "The h, p and h-p Versions of the Finite Element Method in 1 Dimension: Part I. The Error Analysis of the p-Version". In: *Numerische Mathematik* 49 (1986), pp. 577–612. ISSN: 0029599X.

[19] Yiqian He, Haitian Yang, and Andrew J. Deeks. "Use of Fourier shape functions in the scaled boundary method". In: *Engineering Analysis with Boundary Elements* 41 (2014), pp. 152–159. DOI: 10.1016/j.enganabound.2014.01.012.

[20] Magnus R. Hestenes and Eduard Stiefel. “Methods of Conjugate Gradients for Solving Linear Systems”. In: *Journal of Research of the National Bureau of Standards* 49 (6 1952), pp. 409–436.

[21] Paul Houston, Christoph Schwab, and Endre Süli. “Stabilized hp-finite element methods for first-order hyperbolic problems”. In: *SIAM Journal on Numerical Analysis* 37.5 (2000), pp. 1618–1643. DOI: 10.1137/S0036142998348777.

[22] Paul Houston and Endre Süli. “A note on the design of hp-adaptive finite element methods for elliptic partial differential equations”. In: *Computer Methods in Applied Mechanics and Engineering* 194.2-5 SPEC. ISS. (2005), pp. 229–243. ISSN: 00457825.

[23] Paul Houston and Thomas P Wihler. “An hp-adaptive Newton-Discontinuous-Galerkin Finite Element Approach for Semilinear Elliptic Boundary Value Problems”. In: *Mathematics of Computation* 87 (2018), pp. 2641–2674.

[24] https://www.learncpp.com/. *0.3 — Introduction to C/C++*. https://www.learncpp.com/cpp-tutorial/introduction-to-cplusplus/. [Online; accessed 5-March-2020]. 2007.

[25] Claes Johnson. *Numerical Solutions of Partial Differential Equations by the Finite Element Method*. Press Syndicate of the University of Cambridge, 1987. ISBN: 0521347580.

[26] Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2004. ISBN: 9780521009249.

[27] William F. Mitchell and Marjorie A. McClain. “A survey of hp-adaptive strategies for elliptic partial differential equations”. In: *Recent Advances in Computational and Applied Mathematics* 41.1 (2011), pp. 227–258.

[28] A. Mohsen. “A simple solution of the Bratu problem”. In: *Computers and Mathematics with Applications* 67.1 (2014), pp. 26–33.

[29] Alexander Petkov. *How to explain object-oriented programming concepts to a 6-year-old*. https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/. [Online; accessed 5-March-2020]. 2018.

[30] Rostamian Rouben. “Gaussian Quadrature”. In: *Programming Projects in C for Students of Engineering, Science, and Mathematics*. Society for Industrial and Applied Mathematics, 2014, pp. 291–300. ISBN: 9781611973495.

[31] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003. ISBN: 9780898715347.

[32] Christoph Schwab. *p- and hp- Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Oxford Science Publications, 1998. ISBN: 0198503903.

[33] Pavel Šolín, Karel Segeth, and Ivo Doležel. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC, 2003. ISBN: 9781584884385.

[34] John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, 2004. ISBN: 9780898715675.

[35] Bjarne Stroustrup. *The C++ Programming Language*. 4th ed. Pearson Education, 2013. ISBN: 978-0-321-56384-2.

[36] Endre Süli and David F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

[37] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. `http://www.gotw.ca/publications/concurrency-ddj.htm`. [Online; accessed 30-April-2020]. 2009.

[38] Gabor Szegö. *Orthogonal Polynomials*. American Mathematical Society, 1967.

[39] R. Verfürth. “A posteriori error estimation and adaptive mesh-refinement techniques”. In: 50.1-3 (1994), pp. 67–83. ISSN: 0377-0427.

[40] Thu Hang Vu and Andrew J. Deeks. “Blossom-Quad: A non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm”. In: *International Journal for Numerical Methods in Engineering* 73 (2008), pp. 47–70.

[41] Werner C . Rheinboldt. “On a Theory of Mesh-Refinement Processes”. In: *SIAM Journal on Numerical Analysis* 17.6 (1980), pp. 766–778.

[42] Thomas P. Wihler. “An hp-adaptive strategy based on continuous Sobolev embeddings”. In: *Journal of Computational and Applied Mathematics* 235.8 (2011), pp. 2731–2739. ISSN: 03770427. DOI: `10.1016/j.cam.2010.11.023`.

[43] Henry Wilbraham. “On a certain periodic function”. In: *The Cambridge Mathematical Journal* 7 (1848), pp. 198–201.