The hp-Finite Element Method, a C++ implementation

Andrew Oldham

Dissertation submitted to the University of Nottingham for the degree of Master of Science

August 2016

Declaration of Originality

I, Andrew Oldham, hereby declare that the work contained in this dissertation has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due references are made.

SIGNED...Andrew Oldham

DATE..24/08/2016

Acknowledgements

I would like to thank Professor Paul Houston for his initial willingness to supervise this project and consistent support throughout.

Contents

Abstract

The finite element method is a numerical method used in many disciplines of engineering, science and mathematics to efficiently numerically approximate solutions to specific partial differential equations with boundary conditions. There are many different forms of finite element methods that can be applied to different classes of PDE problems, one such method is called the hp-finite element method that uses an iterative approach to generate increasingly accurate numerical approximations to the solution of a specific PDE problem. For this dissertation project a hp-finite element method code was developed that can be applied to general second order linear one dimensional PDE problems whereby the coefficient of the second derivative in the PDE is a negative constant and the boundary conditions specified at either end of the domain are either homogeneous or nonhomogeneous dirichlet. Particular details of the hp-finite element method vary from implementation to implementation and the more fundamental of these considerations are discussed. As well as the thesis giving a mathematically accessible introduction to the underlying theory of the hp-finite element method it also gives a detailed exposition of the developed code so that an individual wishing to amend the current implementation can do so or can attempt to develop their own implementation.

The first chapter details necessary mathematical machinery required to understand proceeding chapters. This section is fairly brief and just serves as a mathematical refresher for those who are unfamiliar with certain aspects of real and functional analysis. The second chapter is an introduction to the general finite element method and to the particular hp-finite element variant. The general FEM and hp-FEM are explained in a very general framework within this chapter, leaving the details to later chapters. The second chapter also outlines the targets and goals of the dissertation. The third chapter develops mathematically the hp-finite element method, whilst mentioning very few implementation specific details. The purpose of this chapter is to give the reader a sufficient understanding of the hp-finite element method to enable them to understand, appreciate, develop and contribute to an implementation. The fourth chapter gives a comprehensive description of the implementation developed by the author using the C++ programming language. Here implementation specific details are discussed and solutions proposed that were used within the implementation developed. This implementation will be made available to the public so that this fourth chapter will also aid any future users in understanding and using the code. The fifth chapter describes a test problem that was used to verify the hp-FEM code that was developed. Results for this test problem are discussed within this

chapter. The sixth chapter discusses conclusions and further work that can be done to improve the code/implementation developed by the author.

The author is aware of the lack of rigour when discussing the mathematical development of the finite element method (for example we never give the definition of a Hilbert Space), however due to time constraints and the authors preference towards algorithmic development, the author believes that the mathematical exposition is acceptable and sufficient in imparting understanding with the reader.

Chapter 1

Mathematical Preliminaries

We shall now describe certain fundamental mathematical concepts that are used throughout the dissertation.

Derivative Notation:
Define the multi index, α, such that $\alpha = (\alpha_1, ..., \alpha_n) \in \mathbb{N}$. The length of a multi index is defined as $|\alpha| = \alpha_1 + ... + \alpha_n$.

Using this notation, we define

$$D^\alpha = \frac{\partial^{\alpha_1}}{\partial x_1^{\alpha_1}} .... \frac{\partial^{\alpha_n}}{\partial x_n^{\alpha_n}} = \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} ... \partial x_n^{\alpha_n}}.$$

The Weak Derivative:
Suppose that u is a locally integrable function on $\Omega \subset \Re^n$. Suppose also that there exists a function w_α that is locally integrable on Ω such that

$$\int_\Omega w_\alpha(\mathbf{x}) \cdot v(\mathbf{x}) \ d\mathbf{x} = (-1)^\alpha \int_\Omega u(\mathbf{x}) \cdot D^\alpha v(\mathbf{x}) \ d\mathbf{x} \qquad \forall v \in C_0^\infty(\Omega),$$

then we say that w_α is a weak derivative of the function u of order $|\alpha| = \alpha_1 + ... + \alpha_n$, and we write $w_\alpha = D^\alpha u$.

Function Space definitions:
The space denoted by $L_2(\Omega)$ is defined by the set of all real valued functions defined on the open subset $\Omega \in \Re^n$ such that

$$\|u\|_{L_2(\Omega)} = \left(\int_\Omega |u(\mathbf{x})|^2 \ d\mathbf{x} \right)^{1/2} < \infty.$$

The space $L_2(\Omega)$ can be equipped with the inner product

$$\langle u, v \rangle_{L_2(\Omega)} = \int_\Omega u(\mathbf{x}) v(\mathbf{x}) d\mathbf{x}.$$

Let m be a non-negative integer and D^α denote a weak derivative of order $|\alpha|$, then we define

$$H^m(\Omega) = u \in L_2(\Omega) : D^\alpha u \in L_2(\Omega) \qquad \forall |\alpha| \leqslant m.$$

$H^m(\Omega)$ is called a Sobolev space of order m and it is equipped with the norm

$$\|u\|_{H^m(\Omega)} = \left(\sum_{|\alpha| \leqslant m} \|D^\alpha u\|^2_{L_2(\Omega)} \right)^{1/2}.$$

The Sobolev space $H^m(\Omega)$ becomes a Hilbert Space with the inner product that induces the above norm. One may also define the following semi-norm

$$|u|_{H^m(\Omega)} = \left(\sum_{|\alpha| = m} \|D^\alpha u\|^2_{L_2(\Omega)} \right)^{1/2}.$$

For a rigorous development of Sobolev Space theory see [1].

Chapter 2

Introduction

The finite element method is an efficient numerical method with rigorous mathematical foundations that is used to approximate solutions to partial differential equations (PDE's) where boundary conditions are specified. As such the finite element method is used in many disciplines, generally to approximate solutions to the equations that govern/model some aspect of reality, such as the equations of fluid mechanics [9], thermodynamics, general relativity, astrophsyics, electromagnetics, and structural mechanics[10]. We shall now, briefly, describe the finite element method before outlining the specific focus of this project.

Let $\Omega \subset \Re^n$ be a domain upon which the differential equation $\mathbf{L}u = f$ holds, whereby $\mathbf{L}$ is a differential operator consisting of weak derivatives and f is a real valued function defined on Ω. From this point onwards we assume that all derivatives are weak derivatives. Then, we assume that a solution ,u, exists and that both u and f are sufficiently well behaved on the domain Ω, so that we have:

$\forall$ real valued function v defined on Ω

$$\int_\Omega (\mathbf{L}u) \cdot v \ \ dx = \int_\Omega f \cdot v \ \ dx.$$

We are then free to selectively impose conditions on v and further assumptions on u to obtain a real Hilbert Space, $\mathbf{V}$, such that,

u $\in \mathbf{V}$, and $\forall$ v $\in \mathbf{V}$

$$a(u,v) = \int_\Omega (\mathbf{L}u) \cdot v \ \ dx = \int_\Omega f \cdot v \ \ dx \tag{2.1}$$

whereby a$(\cdot,\cdot)$ is a bilinear form defined on $\mathbf{V}\times\mathbf{V}$ such that $\forall v \in \mathbf{V}$, $|a(v,v)| < \infty$. Thus we wish to find $u \in \mathbf{V}$ such that (2.1) holds true. This is known as the weak formulation of the PDE problem. A solution to the weak formulation (often called a weak solution) may not necessarily define a solution to the original PDE in a classical sense (e.g. smoothness up to a certain degree), but nevertheless behave in the expected way with regards to (2.1).

The finite element method is a numerical method which produces an approximation to the solution of the weak formulation of the original PDE problem (2.1). There are ways of refining the initial approximation to iteratively tend towards the weak solution.

To obtain an initial approximation of u, we consider an approximation of u that lies within a finite dimensional subspace of $\mathbf{V}$, $u_h \in \mathbf{V_h}$, that also behaves in correspondence with the weak formulation in the following way,

$\forall \ \mathrm{v_h} \in \mathbf{V_h}$

$$a(u_{\mathrm{h}}, v_{\mathrm{h}}) = \int_{\Omega} f \cdot v_{\mathrm{h}} \ dx. \tag{2.2}$$

We can then compute, using linear algebra, the approximation $\mathrm{u_h}$ as follows: Let $\beta = \{\phi_1, ..., \phi_{\mathrm{N}}\}$ be an ordered basis for $\mathbf{V_h}$, $A \in M_{\mathrm{NxN}}(\Re)$ and $F \in \Re^{\mathrm{N}}$ such that $A_{\mathrm{i,j}} = a(\phi_{\mathrm{j}}, \phi_{\mathrm{i}})$ and $F_{\mathrm{i}} = \int_{\Omega} f \cdot \phi_{\mathrm{i}} \ dx$. Then it follows that (2.2) is true iff $\exists U = \{U_1, ..., U_{\mathrm{N}}\} \in \Re^{\mathrm{N}}$ such that $AU = F$. Furthermore, if the Lax-Milgram theorem holds for the weak formulation (2.2), then the solution vector $U = \{U_1, ..., U_{\mathrm{N}}\} \in \Re^{\mathrm{N}}$ is unique.

u_{h} then takes the form

$$u_{\mathrm{h}} = \sum_{i=1}^{N} U_{\mathrm{i}} \cdot \phi_{\mathrm{i}}. \tag{2.3}$$

In true mechanical engineering fashion, we shall call A the stiffness matrix and F the load vector. Having solved the system $AU = F$ then, the approximate solution of the weak formulation can be found via a simple linear combination of functions in $\mathbf{V_h}$. This approximate solution of the weak formulation can be called the finite element solution. The finite element method for solving a given PDE with boundary conditions can be thus thought of broadly as deriving a weak formulation, deciding upon and constructing a finite dimensional space upon which we consider the discrete form of the weak formulation (2.2), assembling the stiffness matrix and load vector and finally solving this linear matrix system.

The finite element solution u_{h} that we obtain is contained within $\mathbf{V_h} \subset \mathbf{V}$ and thus can only be as accurate (as close to $u \in \mathbf{V}$ as possible) as the subspace $\mathbf{V_h}$ allows. One way of iteratively increasing the accuracy of the finite element solution u_{h} is through selectively altering the finite dimensional subspace $\mathbf{V_h}$ so that, measured in some norm, the space $\mathbf{V_h}$ and thus the potential solution u_{h} becomes closer to the true weak solution u. This method requires reassembling the stiffness matrix and load vector for the altered space $\mathbf{V_h}$ and solving this new system. The process of iteratively altering the finite dimensional space $\mathbf{V_h}$ in order to gain increasing accuracy in the approximate weak solution u_{h} is often called the adaptive finite element method, with different varieties of adaptive finite element methods choosing to refine the space $\mathbf{V_h}$ using different techniques.

Typically the space $\mathbf{V}$ that the true solution to the weak formulation lies in is infinite dimensional, so that often in order to obtain an approximation to meaningful accuracy the dimension of the finite dimensional subspace $\mathbf{V_h}$, N, can range from anywhere between 1e3 and 1e9, meaning that the matrix system $AU = F$ must be constructed and solved on a (set of) machine(s).

Due to the size of the matrix system in question, when constructing, storing, and solving the system on a machine it is extremely important to be able to access low level machine features to help optimize overall efficiency. However, despite the clear benefit to being able to access low level machine features directly (such as memory addresses) without any time/resource consuming middle process, when implementing a finite element method code it is still important for the code to be easily understood to allow efficient maintainability and development. Generally there exists a trade off between how close a programming language resembles exact machine instructions and how easy the language is to develop in, read, and maintain.

The C++ programming language was implemented by Bjarne Stroustrup, with the first standard being released on 1 September 1998. The language offers a compromise between being "close to the hardware", ie providing facilities that make it possible to write code that accesses low level machine properties, whilst at the same time being a compiled language (and thus any code developed in C++ may be distributed to run on different CPU's) that supports OOP (Object Oriented Programming) principles that allow large projects to be easily developed and maintained. Due to the nature of the C++ programming language it is widely used for many tasks including driver development, compiler writing, system level critical code in embedded systems as well as general tasks that require optimal performance and portability such as scientific computing and data analytics. As a result of the language supporting OOP principles (enhanced code organization, human readability and maintainability), low level machine features (increased performance and run time efficiency), CPU independent compilation and execution (efficient distribution), and also the existence of detailed freely available language specific documentation, this language has been chosen to develop the implementation of a specific adaptive finite element method. As an introduction to the C++ programming language or (high level) programming in general, see [7], and for an introduction to machine level programming see [5].

In this project we shall consider the general second order linear ordinary differential equation in one spatial dimension of the form:

$$-\varepsilon \cdot \frac{\partial^2 u}{\partial x^2} + b(x) \cdot \frac{\partial u}{\partial x} + c(x) \cdot u = f, \;\; u(a) = u_a, \;\; u(b) = u_b \qquad (2.4)$$

on the interval $\Omega = (a, b)$. Whereby $\varepsilon > 0$, $u_a, u_b \in \Re$ and $b(x), c(x)$ and f are all real valued functions that satisfy certain conditions which shall be specified and explained in the next chapter. The specific adaptive finite element method that has been implemented to obtain a finite element solution to this general problem is called the h-p Finite Element Method (hp FEM). This project aims to elucidate the developed hp-FEM implementation that can be applied to solve (2.4) as well as the associated underlying mathematical theory.

Briefly, suppose $\Omega = (a, b)$, then we denote the computational domain by $\Omega_C = [a, b]$. There then exists a finite number of closed intervals within Ω_C, denoted by the set $I = \{I_1, I_2, .., I_n\}$ whereby $I_1 = [x_0 = a, x_1]$, $I_2 = [x_1, x_2]$, $..., I_n = [x_{n-1}, x_n = b]$. $\forall 1 \leq k \leq n$, associated with the closed interval $I_k = [x_{k-1}, x_k]$ there is a positive integer d_k. The hp-FEM solution is then defined on Ω_C in a piecewise manor, whereby on the k^{th} closed interval, I_k, the finite element solution u_h is a polynomial of degree d_k. The space V_h is the set of all piecewise polynomial functions that can be defined on Ω_C in this way (using a particular convention). The stiffness matrix and load vector are then assembled as previously described and the first approximation of the solution to the weak formulation (often called weak solution) is obtained by solving the system $AU = F$. The space $\mathbf{V_h}$ is then refined by computing a floating number , η_k for each closed interval I_k called the local error indicator, that is a measure of the error of the finite element solution on the domain I_k. These local error indicators are then used to determine which domains I_k should be refined. To refine the space V_h on a particular domain I_k we either increase the value of d_k (p-refinement), which allows the finite element solution to display a greater degree of freedom locally on the interval I_k, or we split the domain I_k into two separate domains (of identical length) $I_{k1} = [x_{k-1}, \frac{x_k + x_{k-1}}{2}]$, $I_{k2} = [\frac{x_k + x_{k-1}}{2}, x_k]$ whereby $d_{k1} = d_{k2} = d_k$

(h-refinement), which allows the finite element solution to be a piecewise polynomial of degree d_k on the interval $I_k = I_{k1} \cup I_{k2}$. A fundamental part of the hp FEM is the process which decides, upon a specific interval I_k, whether to refine via increasing the local polynomial degree (p refinement) or to refine the spatial resolution of the grid locally (h refinement). The hp FEM code will allow user specification of a tolerance that the approximate weak solution obtained shall adhere to when measured in the $\|\cdot\|_{H^1(\Omega)}$ norm against the true weak solution, this accurate solution shall be obtained via iteratively refining the space $\mathbf{V_h}$ as described briefly above.

Within this dissertation we give a mathematical presentation of the theory that underpins the finite element method and mathematically address certain issues unique to the hp-FEM (chapter 3). We explain how the hp-FEM code was developed and also discuss certain design choices (chapter 4). The developed code is verified using a particular test case and areas for further development are discussed (chapters 5 and 6). The developed hp-FEM code will be made publicly available so that this dissertation can also help individuals who want to understand and adapt the code for their own use.

Chapter 3

The hp-FEM (Linear 1D 2nd Order BVP)

We shall now in detail mathematically develop the finite element method for the general problem (2.4) with either homogeneous or nonhomogeneous dirichlet boundary conditions. Thus, let us consider the following problem: Let $\varepsilon > 0 \in \Re$, $b(x), c(x)$ and $f \in C[a, b]$, whereby $\Omega = (a, b) \subset \Re$ is such that

$$-\varepsilon \cdot \frac{d^2u}{dx^2} + b(x) \cdot \frac{du}{dx} + c(x) \cdot u = f \qquad \forall x \in \Omega = (a, b), \tag{3.1}$$

$$u(a) = u_a, \;\; u(b) = u_b. \tag{3.2}$$

We first multiply (3.1) by an arbitrary test function v, which is defined on Ω, thus we obtain

$$-\varepsilon \frac{d^2u}{dx^2} v + b\frac{du}{dx} v + cuv = fv \qquad \forall x \in \Omega = (a, b). \tag{3.3}$$

Taking the integral of (3.3) over the domain $\Omega = (a, b)$ gives us the equation

$$\int_\Omega -\varepsilon \frac{d^2u}{dx^2} v + b\frac{du}{dx} v + cuvdx = \int_\Omega fvdx.$$

Making use of integration by parts, we have that

$$-\varepsilon \cdot \int_{\partial\Omega} \frac{du}{dx} v \;\, dx + \int_\Omega \varepsilon \frac{du}{dx}\frac{dv}{dx} + b\frac{du}{dx} v + cuv \;\, dx = \int_\Omega fvdx.$$

Now, we impose the condition on v that $v(a) = v(b) = 0$ so that

$$\int_\Omega \varepsilon \frac{du}{dx}\frac{dv}{dx} + b\frac{du}{dx} v + cuv \;\, dx = \int_\Omega fv \;\, dx.$$

In order to prove necessary properties of the bilinear form that our weak formulation shall involve, it is necessary to make certain assumptions at this point about the functions u and v, namely that $v \in H_0^1(\Omega)$ and $u \in H_0^1(\Omega)$[homogeneous dirichlet bc's] or $H^1(\Omega)$[nonhomogeneous dirichlet bc's]. Using this assumption together with the Cauchy-Schwartz and Triangle Inequality for Inner Product spaces we obtain the following:
$\forall v \in H_0^1(\Omega)$,

$$|\int_\Omega \varepsilon \frac{dv}{dx}\frac{dv}{dx}+b\frac{dv}{dx}v+cvv \;\; dx| \leqslant \varepsilon \|\frac{dv}{dx}\|_{L_2(\Omega)}\|\frac{dv}{dx}\|_{L_2(\Omega)}+\max_{x\in\Omega}|b(x)|\cdot\|\frac{dv}{dx}\|_{L_2(\Omega)}\|v\|_{L_2(\Omega)}$$
$$+\max_{x\in\Omega}|c(x)|\cdot\|v\|_{L_2(\Omega)}\|v\|_{L_2(\Omega)}<\infty.$$

Thus we have that $\forall v \in H_0^1(\Omega),\;\; a(v,v) < \infty$ and $a(0_{H_0^1(\Omega)}, 0_{H_0^1(\Omega)}) = 0$. These are necessary conditions that the bilinear form $a(\cdot,\cdot)$ must satisfy in order to be able to construct the weak formulation.

3.1 Weak formulation development

3.1.1 homogeneous dirichlet boundary conditions

We define the weak formulation of problem (3.1-3.2) with homogeneous dirichlet boundary conditions as follows:
We wish to find $u \in H_0^1(\Omega)$ such that $\forall v \in H_0^1(\Omega)$,

$$a(u,v) = l(v) \tag{3.4}$$

whereby

$$a(u,v) = \int_\Omega \varepsilon \frac{du}{dx}\frac{dv}{dx} + b\frac{du}{dx}v + cuv \;\; dx$$

and

$$l(v) = \int_\Omega fv \;\; dx = \langle f, v\rangle_{L_2(\Omega)}.$$

Due to the operations of differentiation and integration being linear, it follows that $a(\cdot,\cdot)$ is a bilinear form defined on
$H_0^1(\Omega) \times H_0^1(\Omega)$and $l(\cdot)$ is a linear functional defined on $H_0^1(\Omega)$.

At this point we formally state the Lax-Milgram theorem as it is a powerful theorem that underpins the finite element method, allowing us to guarantee a unique solution to the weak formulation (3.4).

Lax-Milgram Theorem

Suppose that V is a real Hilbert Space equipped with a norm $\|\cdot\|_V$. Let $a(\cdot,\cdot)$ be a bilinear form defined on $V\times V$ such that:

(a) $a(\cdot,\cdot)$ is coercive, i.e., $\exists c_0{>}0$ such that $\forall v\in V, a(v,v)\geq c_0\|v\|_V^2$.
(b) $a(\cdot,\cdot)$ is continuous, i.e., $\exists c_1{>}0$ such that $\forall v,w\in V,\ \ |a(v,w)|\leq c_1\|v\|_V\|w\|_V$,
and let $\ell(\cdot)$ be a linear functional on V such that

(c) ℓ is continuous, i.e., $\exists c_2{>}0$ such that $\forall v\in V\ \ |\ell(v)|\leq c_2\|v\|_V$.

Then, there exists a unique $u\in V$ such that

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

Thus a unique solution to the weak formulation (3.4) exists (with $V=H_0^1(\Omega)$ in the Lax-Milgram theorem) if we can prove that the bilinear form $a(\cdot,\cdot)$ is both coercive and continuous and that the linear functional $\ell(\cdot)$ is continuous. We shall prove these properties now:

Continuity of a$(\cdot,\cdot)$
$\forall v,w\in H_0^1(\Omega)$,

$$\begin{aligned}|a(v,w)| &\leqslant \varepsilon\|\frac{dv}{dx}\|_{L_2(\Omega)}\|\frac{dw}{dx}\|_{L_2(\Omega)}+\max_{x\in\Omega}|b(x)|\cdot\|\frac{dv}{dx}\|_{L_2(\Omega)}\|w\|_{L_2(\Omega)}\\ &\quad+\max_{x\in\Omega}|c(x)|\cdot\|v\|_{L_2(\Omega)}\|w\|_{L_2(\Omega)}\\ &\leq \big(\varepsilon+\max_{x\in\Omega}|b(x)|+\max_{x\in\Omega}|c(x)|\big)\cdot\Big(\|\frac{dv}{dx}\|_{L_2(\Omega)}\|\frac{dw}{dx}\|_{L_2(\Omega)}+\|\frac{dv}{dx}\|_{L_2(\Omega)}\|w\|_{L_2(\Omega)}+\|v\|_{L_2(\Omega)}\|w\|_{L_2(\Omega)}\Big)\\ &\leq \big(\varepsilon+\max_{x\in\Omega}|b(x)|+\max_{x\in\Omega}|c(x)|\big)\cdot\Big(2\|\frac{dv}{dx}\|^2_{L_2(\Omega)}+\|v\|^2_{L_2(\Omega)}\Big)^{1/2}\Big(\|\frac{dw}{dx}\|^2_{L_2(\Omega)}+2\|w\|^2_{L_2(\Omega)}\Big)^{1/2}\\ &\leq \big(\varepsilon+\max_{x\in\Omega}|b(x)|+\max_{x\in\Omega}|c(x)|\big)\cdot(2\|v\|^2_{H_0^1(\Omega)})^{1/2}(2\|w\|^2_{H_0^1(\Omega)})^{1/2}\\ &= 2\big(\varepsilon+\max_{x\in\Omega}|b(x)|+\max_{x\in\Omega}|c(x)|\big)\cdot\|v\|_{H_0^1(\Omega)}\|w\|_{H_0^1(\Omega)}\\ &= c_1\|v\|_{H_0^1(\Omega)}\|w\|_{H_0^1(\Omega)},\end{aligned}$$

as required. Whereby $c_1=2\big(\varepsilon+\max_{x\in\Omega}|b(x)|+\max_{x\in\Omega}|c(x)|\big)$.

Continuity of $\ell(\cdot)$
$\forall v\in H_0^1(\Omega)$,

$$\begin{aligned}|\ell(v)| &= |\langle f,v\rangle_{L_2(\Omega)}|\leq\|f\|_{L_2(\Omega)}\|v\|_{L_2(\Omega)}\\ &\leq\|f\|_{L_2(\Omega)}\Big(\|v\|^2_{L_2(\Omega)}+\|\frac{dv}{dx}\|^2_{L_2(\Omega)}\Big)^{1/2}=\|f\|_{L_2(\Omega)}\|v\|_{H_0^1(\Omega)}=c_2\|v\|_{H_0^1(\Omega)}\end{aligned}$$

as required, whereby $c_2=\|f\|_{L_2(\Omega)}$.

Coercivity of a$(\cdot,\cdot)$

In order to prove the coercivity of a$(\cdot,\cdot)$ we first need the following result.
Poincare-Friedrichs Inequality: Let $\Omega = (a,b) \subset \Re$ and let $u \in H_0^1(\Omega)$, then there exists a constant $c_{PF} = \frac{1}{2}(b-a)^2$ such that $\|u\|^2_{L_2(\Omega)} \leq c_{PF}\|\frac{du}{dx}\|^2_{L_2(\Omega)}$.
Now then, we have that
$\forall v \in H_0^1(\Omega)$,

$$a(v,v) = \int_\Omega \varepsilon \frac{dv}{dx}\frac{dv}{dx} + b\frac{dv}{dx}v + cvv \ \ dx.$$

Now, via recalling the integration by parts formula we obtain

$$\int_\Omega b\frac{dv}{dx}v \ \ dx = \int_\Omega \frac{1}{2}b\frac{dv^2}{dx} \ \ dx = \int_{\partial\Omega}\frac{1}{2}bv^2 \ \ dx - \frac{1}{2}\int_\Omega \frac{db}{dx}v^2 \ \ dx$$
$$= -\frac{1}{2}\int_\Omega \frac{db}{dx}v^2 \ \ dx,$$

so that we have

$$a(v,v) = \int_\Omega \varepsilon\frac{dv}{dx}\frac{dv}{dx} + b\frac{dv}{dx}v + cvv \ \ dx = \int_\Omega \varepsilon\frac{dv}{dx}\frac{dv}{dx} \ \ dx + \int_\Omega \Big(c - \frac{1}{2}\frac{db}{dx}\Big)v^2 \ \ dx$$

and we make the assumption that $c(x) - \frac{1}{2}\frac{db(x)}{dx} \geq 0, \ \ \forall x \in [a,b]$. Thus it follows that,

$$a(v,v) \geq \varepsilon\|\frac{dv}{dx}\|^2_{L_2(\Omega)},$$

hence due to the Poincare-Friedrichs Inequality, we have that

$$\frac{1}{\varepsilon}a(v,v) \geq \|\frac{dv}{dx}\|^2_{L_2(\Omega)} \geq \frac{1}{c_{PF}}\|v\|^2_{L_2(\Omega)}$$

so that

$$\frac{1+c_{PF}}{\varepsilon}a(v,v) \geq \|\frac{dv}{dx}\|^2_{L_2(\Omega)} + \|v\|^2_{L_2(\Omega)} = \|v\|^2_{H_0^1(\Omega)}.$$

Hence, if we let $c_0 = \frac{\varepsilon}{1+c_{PF}}$ then we finally obtain

$$a(v,v) \geq c_0\|v\|^2_{H_0^1(\Omega)}.$$

We have now proven that the bilinear form, $a(\cdot,\cdot)$, and linear functional, $\ell(\cdot)$, in the weak formulation (3.4) are defined so that we may apply the Lax-Milgram theorem (assuming the assumption $c(x) - \frac{1}{2}\frac{db(x)}{dx} \geq 0, \ \ \forall x \in [a,b]$ holds true) to guarantee a unique solution of the weak formulation.

Since the space $H_0^1(\Omega)$ is infinite dimensional, to make the space more manageable and easier to represent on a machine we consider a finite dimensional subspace of this space, which we denote by V_h. It can be shown that for any real Hilbert Space V with an associated bilinear form and linear functional that satisfies the Lax-Migram theorem, the Lax-Milgram theorem also holds true on any finite dimensional subspace $V_h \subset V$, with the bilinear form and linear functional now being restricted to the subspaces $V_h \times V_h \subset V \times V$ and $V_h \subset V$ respectively.

This means then that we may consider an arbitrary finite dimensional subspace of $H_0^1(\Omega)$, denoted by V_h, and consider the discretized version of the weak formulation (3.4) to obtain an approximate solution to the weak formulation that is contained within the space V_h. The Lax-Milgram theorem will then guarantee a unique solution , u_h to our discretized weak formulation in the space V_h. The discrete weak formulation can be defined then as:
We wish to find the unique $u_h \in V_h$ such that, $\forall v_h \in V_h$,

$$a(u_h, v_h) = l(v_h) \tag{3.5}$$

whereby $a(\cdot,\cdot)$ and $\ell(\cdot)$ are as defined in (3.4) but restricted to the spaces $V_h \times V_h$ and V_h respectively. The solution u_h to the discretized weak formulation is an approximation to the true weak solution u that satisfies the weak formulation on the larger infinite dimensional real Hilbert Space $H_0^1(\Omega)$.

3.1.2 nonhomogeneous dirichlet boundary conditions

Now, suppose that the initial problem (3.1-3.2) was given with nonhomogeneous dirichlet boundary conditions. So that we have the weak formulation
We wish to find $u \in H^1(\Omega)$ such that $u(a) = u_a,\ \ u(b) = u_b$, and $\forall v \in H_0^1(\Omega)$

$$a(u, v) = \ell(v) \tag{3.6}$$

whereby $a(\cdot,\cdot)$ and $\ell(\cdot)$ are defined as in (3.4). Similar to the homogeneous dirichlet boundary case we wish to find an approximation of this u, however we consider two finite dimensional spaces instead of one, the spaces $V_h \subset H_0^1(\Omega)$ and $V_{h^+} \subset H^1(\Omega)$. V_{h^+} and V_h are such that there exists an ordered basis $\beta_+ = \{\phi_1, \phi_2, .., \phi_{N-1}, \phi_N\}$ for V_{h^+} whereby the linearly independent ordered set $\beta_0 = \{\phi_2, \phi_3.., \phi_{N-1}\} \subset \beta_+$ is an ordered basis for V_h so that $dim(V_h) = dim(V_{h^+}) - 2$. We then have the discrete weak formulation
We wish to find $u_h \in V_{h^+}$ such that, $\forall v \in V_h$

$$a(u_h, v_h) = \ell(v_h). \tag{3.7}$$

Recall that $u_h(a) = u_a$ and that $u_h(b) = u_b$. The construction process of V_{h^+} is such that $\forall 2 \leq j \leq n-1$, $\phi_j(a) = \phi_j(b) = \phi_1(b) = \phi_N(a) = 0$, where $\phi_1(a) = \phi_N(b) = 1$. Thus $\exists w_h \in V_h \subset H_0^1(\Omega),\ \ b_h \in V_{h^+} \subset H^1(\Omega)$ such that $b_h = u_a\phi_1 + u_b\phi_N$ and $u_h = w_h + b_h$. Hence we have that $\forall v_h \in V_h$

$$a(w_h, v_h) = \ell(v_h) - a(b_h, v_h).$$

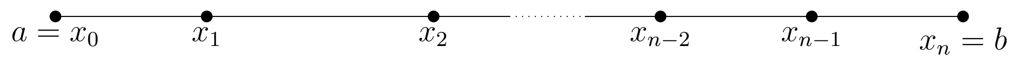
Thus if we define the linear functional $\ell_b(\cdot)$ on V_h via $\forall v_h \in V_h\ \ell_b(v_h) = \ell(v_h) - a(b_h, v_h)$ then we may equivalently define the weak formulation (3.7) as:
We wish to find $w_h \in V_h$ such that $\forall v_h \in V_h$

$$a(w_h, v_h) = \ell_b(v_h). \tag{3.8}$$

It is fairly trivial to prove that $\ell_b(\cdot)$ defined on $V_h \subset H_0^1(\Omega)$ is a continuous linear functional and is thus left as an exercise for the interested reader. It follows then that the Lax-Milgram theorem can be applied to the discrete weak formulation (3.8) to guarantee a unique weak solution $u_h = (w_h + b_h) \in V_{h^+}$ when the initial problem prescribes nonhomogeneous dirichlet boundary conditions.

3.2 A mesh of lagrangian finite elements

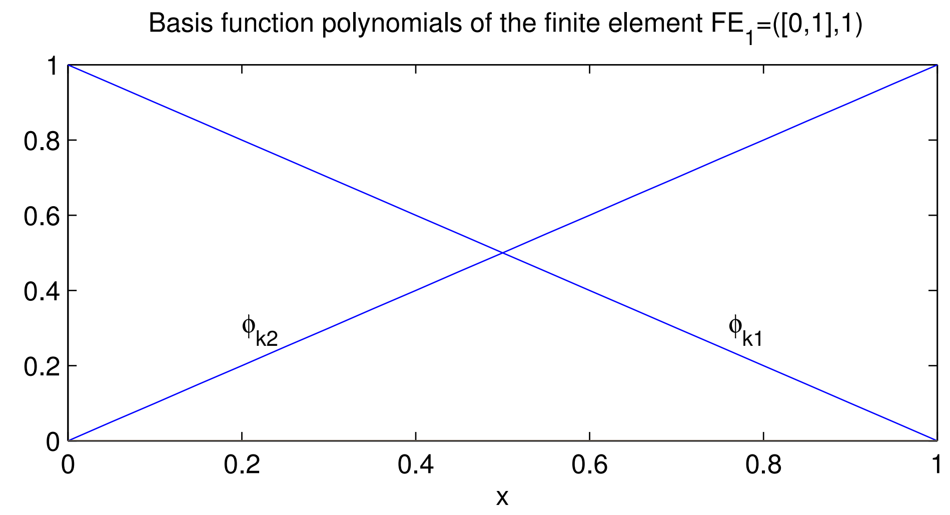
Exactly how we construct and store the finite dimensional subspace V_h on a machine is an algorithmic implementation detail that shall be thoroughly addressed in the next chapter, however we briefly mathematically describe the construction process now. Let $\Omega_C = [a, b] \subset \Re$ define the computational domain of our problem, and let $I_k = [x_{k-1}, x_k] \subset \Omega_C,\ \forall k = 1, 2, ..., n$ such that $\forall 1 \leq i<j \leq n,\ x_i \leq x_{j-1}$ and $I_i \cap I_j \in \{\emptyset,\ \{x_i\}\}$. Crucially we also have that $\Omega_C = \cup_{k=1}^{n} I_k$. We diagrammatically illustrate the set of closed intervals $\{I_k\ |1 \leq k \leq n\}$ below.



whereby $I_k = [x_{k-1}, x_k]$.

Now, for each $I_k \subset \Omega_C$ we have a linearly independent set of polynomials (all of finite degree $d_k \geq 1 \in \mathbb{N}$) defined on I_k, that we shall denote by P_k. $P_k = \{\phi_{k1}, \phi_{k2}, ..., \phi_{k(d_k+1)}\}$ is such that if you consider the set of $d_k + 1$ uniformly distanced points $\{x_{k-1} = x_{k1}<x_{k2}<...<x_{k(d_k+1)} = x_k\}$, then

$\forall 1 \leq i, j \leq d_k + 1,\ \ \phi_{ki}(x_{kj}) = \delta_{ij}$. The set $I_k \subset \Omega_C$ together with the linearly independent set of polynomials P_k defined in the above manor on I_k is an example of a lagrangian finite element, where P_k is the set of finite element basis functions. We can therefore define a lagrangian finite element with any pairing (I_k, d_k), where $d_k \geq 1 \in \mathbb{N}$ and we implicitly assume that P_k is the set of $d_k + 1$ linearly independent polynomials $\{\phi_{k1}, \phi_{k2}, .., \phi_{k(d_k+1)}\}$, each of degree d_k defined on I_k (the domain of the finite element), such that if we consider the set of uniformly distanced points $\{x_{k1} = x_{k-1}, x_{k2}, .., x_{k(d_k+1)} = x_k\}$, we have that $\forall 1 \leq i, j, \leq d_k + 1,\ \ \phi_{ki}(x_{kj}) = \delta_{ij}$. For a more rigorous treatment of lagrangian finite elements see [3] or [4]. Using this notation, consider the set of lagrangian finite elements $\{(I_k = [0, 1], d_k) \mid 1 \leq d_k \leq 2\ \}$. We show below plots of the finite element basis polynomial functions of each of these lagrangian finite elements.



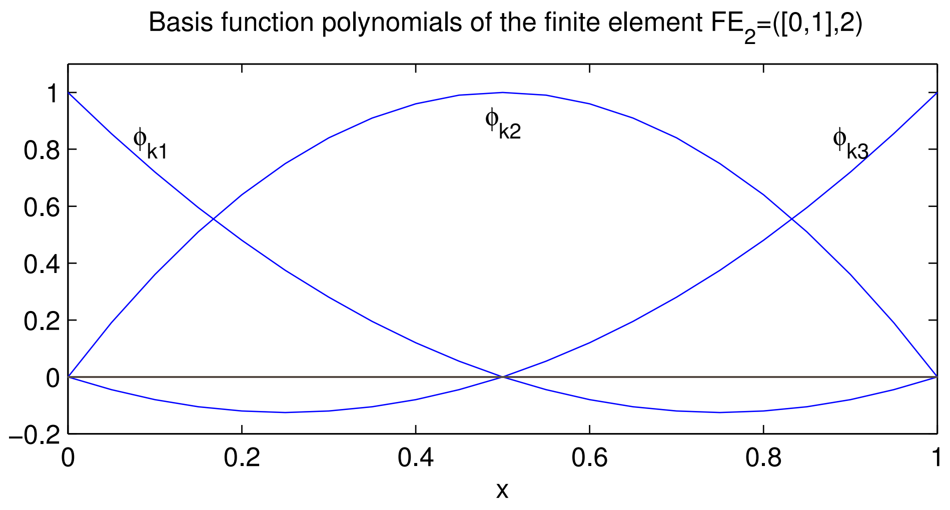


Figure 3.1: Basis function polynomials of the finite elements $(I_k = [0,1],1)$[top] and $(I_k = [0,1],2)$[bottom].

For each $1 \leq k \leq n$, we have a positive integer d_k that generates the lagrangian finite element (I_k, d_k) using the convention detailed previously. We define the mesh to be the set of lagrangian finite elements $\{(I_k, d_k)\ |1 \leq k \leq n\}$. On any finite element domain I_k, the approximate weak solution u_h is just a linear combination of the polynomials in P_k, for this reason the space generated by P_k is often called the space of shape functions of the lagrangian finite element (I_k, d_k).

Let us assume that there are $n > 1 \in \mathbb{N}$ lagrangian finite elements in the mesh. Then $\forall \phi_{ki} \in P_k,$ where $1 \leq i \leq d_k+1$, we denote by ϕ_{ki}^{ext} the piecewise polynomial function defined on Ω_C such that $\phi_{ki}^{ext}(x) = \phi_{ki}(x)\ \forall x \in I_k$, $\phi_{ki}^{ext}(x) = 0\ \forall x \in \Omega_C \backslash I_k$. $\forall 1 \leq k \leq n-1$, $\phi_{k(d_k+1)}^{ext}$ and $\phi_{(k+1)1}^{ext}$ are such that $\phi_{k(d_k+1)}^{ext}(x_k) = 1 = \phi_{(k+1)1}^{ext}(x_k)$. We define the piecewise function ϕ_k^{bext} on Ω_C via $\phi_k^{bext} = \phi_{k(d_k+1)}^{ext} + \phi_{(k+1)1}^{ext}$, then the set

$$\beta = \{\phi_{11}^{ext}, .., \phi_{1d_k}^{ext}, \phi_1^{bext}, \phi_{22}^{ext}, .., \phi_{2d_k}^{ext}, \phi_2^{bext}, .., \phi_{(n-1)d_{n-1}}^{ext}, \phi_{(n-1)}^{bext}, \phi_{n2}^{ext}, .., \phi_{n(d_n+1)}^{ext}\}$$

is an ordered basis for the real Hilbert Space V_{h^+}. Although with this definition, V_{h^+} is not a subspace of $H_0^1(\Omega)$ (but instead a subspace of $H^1(\Omega)$), we construct the stiffness matrix A and load vector F using V_{h^+} and then impose the boundary conditions on the system $AU = F$ to obtain the correct discretized weak solution u_h. $V_h = span(\beta \backslash \{\phi_{11}^{ext}, \phi_{n(d_n+1)}^{ext}\})$.

Let $\beta = \{\phi_{11}^{ext}, .., \phi_{1d_k}^{ext}, \phi_1^{bext}, \phi_{22}^{ext}, .., \phi_{2d_k}^{ext}, \phi_2^{bext}, .., \phi_{(n-1)d_{n-1}}^{ext}, \phi_{(n-1)}^{bext}, \phi_{n2}^{ext}, .., \phi_{n(d_n+1)}^{ext}\} = \{\phi_1, \phi_2, ..., \phi_N\}$, and let us assume that we have constructed the stiffness matrix A and load vector F such that $A \in M_{N\times N}(\Re)$, $F \in \Re^N$ whereby $A_{i,j} = a(\phi_j, \phi_i)$, $F_i = \ell(\phi_i)$. Suppose $U = \{U_1, U_2, ..., U_N\} \in \Re^N$ is the solution vector to the finite dimensional linear system $AU = F$, so that $\forall v_h \in V_{h^+},$ there exists unique ordered scalars $\alpha_1, \alpha_2, ..., \alpha_N \in \Re$ such that $v_h = \sum_{i=1}^N \alpha_i \phi_i$, whereby $\beta = \{\phi_1, \phi_2, ..., \phi_N\}$ is an ordered basis for V_{h^+} as described above. Furthermore, since $a(\cdot,\cdot)$ is a bilinear form and $\ell(\cdot)$ is a linear functional defined on the spaces $V_{h^+} \times V_{h^+}$ and V_{h^+} respectively, we have that (whereby $u_{h^+} = \sum_{i=1}^N U_i \phi_i$)

$$a(u_{h^+}, v_h) = a(u_{h^+}, \sum_{i=1}^N \alpha_i \phi_i) = \sum_{i=1}^N \alpha_i a(u_{h^+}, \phi_i)$$
$$= \sum_{i=1}^N \alpha_i \ell(\phi_i) = \ell(\sum_{i=1}^N \alpha_i \phi_i) = \ell(v_h).$$

Thus it follows that $u_{h^+} \in V_{h^+}$ is such that $\forall v_h \in V_{h^+}\ a(u_{h^+}, v_h) = \ell(v_h)$. Since $V_h \subset V_{h^+}$, it follows that u_{h^+} behaves exactly as our weak solution should do with respect to the discrete weak formulation (3.5) (i.e. that $\forall v_h \in V_h\ a(u_{h^+}, v_h) = \ell(v_h)$) however the only problem is that u_{h^+} may not be of the desired form depending on the whether the values U_1 and U_N match the specified boundary conditions or not. To rectify this problem we slightly alter the system $AU = F$ (before computing the solution U) so that when we solve the altered system the solution vector obtained defines (through a linear combination of the basis functions of β) a true solution to the disrcetized weak formulation (3.5)[corresponding to homogeneous boundary conditions] or (3.7)[corresponding to nonhomogeneous boundary conditions].

The process of altering the system $AU = F$ depends on the type of boundary conditions given, we shall first explain the procedure of altering the system for when homogeneous dirichlet boundary conditions are initially specified in the problem. The dimension of the stiffness matrix and load vector remain identical so that the solution obtained is still represented as a linear combination of the elements in the ordered basis B for V_{h^+}. Due to this it is a requirement that the first and last elements in the solution vector $U \in \Re^N$ must be zero for the homogeneous boundary condition case, otherwise the obtained solution will have nonzero values on at least one of the boundary points. Hence we set the first and last elements in the load vector F to be zero, and let the first and last rows of the matrix A be $e_1 = \{1, 0, ..., 0\}$, and $e_N = \{0, 0, ..., 1\} \in \Re^N$ respectively. By doing this we ensure that the solution obtained to the system $AU = F$ will be such that $U_1 = 0 = U_N$. Although we could solve this system and obtain a correct solution to the weak formulation, we make further simple adjustments in order for the matrix A to become more sparse. Hence any nonzero element occurring in either the first or last column of A is zeroed, assuming it does not lie within either the first or last rows of A. This yields the adjusted stiffness matrix A and load vector F such that if we then solve the system $AU = F$ we obtain the unique finite element solution $u_h = \sum_{i=1}^{N} U_i \phi_i$ to the discrete weak formulation (3.5).

Suppose that the problem as defined in (3.1-3.2) specifies nonhomogeneous boundary conditons, which, in the context of this problem, will amount to specifying values in $\Re$ that the final weak solution should take on the boundary points. Thus we assume that the solution to the weak formulation must have the values u_a and u_b on the boundary points a and b (whereby the domain upon which the PDE holds is defined by $\Omega = (a, b)$.Furthermore suppose that we have constructed the stiffness matrix A and the load vector F using the finite dimensional real Hilbert Space V_{h^+}, so that we just need to slightly alter the stiffness matrix A and load vector F before a solving the altered system, thus obtaining a solution to the weak formulation. The way in which we do this is similar to the homogeneous dirichlet boundary conditions case. Thus we first set the first and last rows of A be of the form $e_1 = \{1, 0, ..., 0\}$, and $e_N = \{0, 0, ..., 1\} \in \Re^N$. Next, $\forall 2 \leq i \leq N - 1$, we consider the entries $A_{i1},\ \ A_{iN}$. If $A_{i1} \neq 0,\ \$ then we let $F_i = F_i - A_{i1}u_a$. Similarly if $A_{iN} \neq 0,\ \$ then we let $F_i = F_i - A_{iN}u_b$. So that if both $A_{i1} \neq 0$ and $A_{iN} \neq 0$ we let $F_i = F_i - A_{i1}u_a - A_{iN}u_b$. Once that matrix A and load vector F have been adjusted in this way we zero all entries on the first and last column of A that do not occur in either the first or final row. Altering the system in the above way guarantees that when we solve the matrix equation $AU = F$ we obtain a solution to the weak formulation defined on the space $H^1(\Omega)$ that satisfies the nonhomogeneous dirichlet boundary conditions as required.

We have now described in relative detail how the space $V_{h^+} \subset H^1(\Omega)$ is constructed and how (assuming stiffness matrix and load vector have been already constructed using this space V_{h^+}) we can alter the system $AU = F$ slightly so that we obtain the correct solution to the discretized weak formulation that abides to the boundary conditions specified in the problem , whether homogeneous dirichlet or nonhomogenoeus dirichlet.

3.3 Stiffness matrix and load vector construction

We shall now describe exactly how we construct the stiffness matrix A and load vector F, assuming that we have already defined the space (whereby for notational ease we drop the $^+$ symbol in V_{h^+}) $V_h \subset H^1(\Omega)$ through an ordered basis $\beta = \{\phi_1, \phi_2, ..., \phi_N\}$ as previously detailed.

Recall that β is an ordered basis, whereby

$$\beta = \{\phi_{11}^{ext}, \phi_{12}^{ext}, .., \phi_{1d_1}^{ext}, \phi_1^{bext}, phi_{22}^{ext}, .., phi_{2(d_2}^{ext}, phi_2^{bext}, .., \phi_{(n-1)d_{n-1}}^{ext}, \phi_{n-1}^{bext}, \phi_n^{ext}, .., \phi_{n(d_n+1)}^{ext}\} = \{\phi_1, \phi_2, ..., \phi_N\} \quad (3.9)$$

and the ordering of these basis functions within β defines a natural numbering of these functions from 1 to N. We also have that $\forall 1 \leq k \leq n$ there exists the piecewise functions $\phi_{k1}^{ext}, \phi_{k2}^{ext}, .., \phi_{k(d_k+1)}^{ext}$ that are defined on $\Omega_C = [a, b]$ that all are nonzero only in the closed interval $I_k \subset \Omega_C$. Also recall that each closed interval I_k defines the domain of the k^{th} lagrangian finite element in the computational domain Ω_C. Define a mapping $M : S_1 \to S_2$, whereby
$S_1 = \{\phi_{11}^{ext}, \phi_{12}^{ext}, .., \phi_{1d_1}^{ext}, \phi_{1(d_1+1)}^{ext}, \phi_{21}^{ext}, phi_{22}^{ext}, .., phi_{2d_2}^{ext}, phi_{2(d_2+1)}^{ext}, .., \phi_{n1}^{ext}, \phi_{n2}^{ext}, .., \phi_{n(d_n+1)}^{ext}\}$
and
$S_2 = \{1, 2, .., N\}$, whereby $M(v)$ takes the integer value that defines the position of the function v within the ordered basis β. $\forall 1 \leq k \leq (n-1)$ $M(\phi_{k(d_k+1)}^{ext}) = M(\phi_{(k+1)1}^{ext}) = M(\phi_k^{bext})$. With this function defined we may easily define the connectivity array, $C \in M_{n \times \max_{1 \leq k \leq n}(d_k+1)}(\Re)$
whereby $C_{ij} = M(\phi_{ij}^{ext})$, and the convention holds that if $j > (d_i + 1)$, then

$C_{ij} = 0$.

Recall that the matrix A and load vector F (before alterations have taken place to impose the boundary conditions on the system $AU = F$) are such that $A_{ij} = a(\phi_j, \phi_i), \;\; F_i = \ell(\phi_i), \;\; \forall 1 \leq i, j \leq N$ whereby the functions ϕ_i form an ordered basis for V_h, β, as previously described. Since each ϕ_i takes nonzero values on at most the domains of two adjacent finite elements in the domain Ω_C, we can make use of this by only considering areas of the domain (specifically finite element domains) Ω_C for which the integrands of $a(\phi_j, \phi_i)$ and $\ell(\phi_i)$ are nonzero. As such, the process of constructing the stiffness matrix A and load vector F can be algorithmically explained as follows.

First we zero the matrix A and load vector F. Then, $\forall 1 \leq t \leq n$, we let $A_{ij} = A_{ij} + a(\phi_j, \phi_i)_t$, whereby $a(v_1, v_2)_t$ defines the bilinear form evaluated over the restricted domain (x_{t-1}, x_t) (recall that $I_k = [x_{k-1}, x_k]$ is the domain of finite element k) and $C_{t1} \leq i, j \leq \max\{C_{t1}, C_{t2}, ..., C_{t(\max_{1\leq k\leq n}(d_k+1))}\}$. Similarly $\forall 1 \leq t \leq n$, we let $F_i = F_i + \ell(\phi_i)_t$, (whereby $\ell(\cdot)_t$ represents the evaluation of the restriction of the linear functional $\ell(\cdot)$ to the domain of the t^{th} finite element within the computation domain Ω_C) $\forall C_{t1} \leq i \leq \max\{C_{t1}, C_{t2}, ..., C_{t(\max_{1\leq k\leq n}(d_k+1))}\}$. We thoroughly define below the meaning of the restrictions of the bilinear form and linear functional ($a(\cdot,\cdot)$ and $\ell(\cdot)$) to a specific finite element, say the t^{th} finite element within the domain Ω_C. Thus, let t=3, and suppose we wish to compute $a(\phi_4, \phi_7)_3$, then we would evaluate

$$a(\phi_4, \phi_7)_3 = \int_{x_2}^{x_3} \varepsilon \frac{d\phi_4}{dx}\frac{d\phi_7}{dx} + b\frac{d\phi_4}{dx}\phi_7 + c\phi_4\phi_7 \;\, dx$$

whereby the domain of the 3^{rd} finite element in the domain Ω_C is given by the closed interval $[x_2, x_3]$ (i.e. $I_3 = [x_2, x_3]$). The restriction of the linear functional $\ell(\cdot)$ is defined in the corresponding way. Practically when evaluating integrals over an interval of the domain we perform integration over the interval $(0, 1)$ and scale the value obtained accordingly, to do this an affine transformation is required from the reference element $(0, 1)$ to the interval in the computational domain that we wish to integrate over (more details on the nature of this shall be given in the implementation chapter).

Thus having now in detail described the process of constructing the stiffness matrix and the load vector A and F we then solve the corresponding linear system $AU = F$ (after altering the system slightly to adhere to specific boundary conditions) to obtain the discretized weak solution, u_h which adheres to the dirichlet boundary conditions given in the initial problem definition. This solution lives in the finite dimensional space V_h and is thus an approximation of the true weak solution u that is contained within an infinite dimensional real Hilbert Space. It is possible to utilize the structure of the underlying PDE, along with Sobolev Space results, to obtain a sharp upper a posteriori error bound, which is given by:

$$\|u - u_h\|_{H^1(a,b)} \leq \frac{1}{2c_0} \|\frac{h}{p \cdot (p+1)} R(u_h)\|_{L^2(a,b)} \tag{3.10}$$

whereby $c_0 > 0$ is the coercivity constant detailed in the Lax-Milgram theorem, $R(u_h) = f - (-\varepsilon u_h'' + b(x)u_h' + c(x)u_h)$ (where the dashes denote weak derivatives), and $p = d_k$ $\forall 1 \leq k \leq n$ is defined (finite) element wise. For a proof of (3.10) see [6].

3.4 hp-mesh refinement

We shall now explain in detail the process of how we adjust the finite dimensional subspace V_h to iteratively increase the accuracy of our approximation. Recall that regardless of the type of dirichelt boundary conditions specified in the problem, either homogoeneous or nonhomogeneous, we still obtain the approximate weak solution as a linear combination of the basis functions within the ordered basis β (whereby if the boundary conditions specified were homogeneous the first and last elements of the solution vector are 0). The space V_h can be expressed as the the mesh (I, D). Whereby $I = \{I_1, I_2, ..., I_n\}$ is the set of ordered closed intervals that define the domains of the lagrangian finite elements and the set $D = \{d_1, d_2, .., d_n\}$ defines the degree of the polynomials on each finite element (whereby $d_k + 1$ polynomials are defined on I_k in the manor previously described). V_h is then the space of all piecewise polynomial functions, v_h, defined on Ω_C such that the restriction to any finite element domain I_k is a linear combination of the polynomials in P_k, whereby $\forall 1 \leq k \leq n-1$, the coefficient of $\phi_{k(d_k+1)}$ in the linear combination defining the restriction of v_h to I_k is identical to the coefficient of $\phi_{(k+1)1}$ in the linear combination defining the restriction of v_h to I_{k+1}. The question then becomes how do we alter the mesh (I, D), to ensure that when we reconstruct the stiffness matrix and load vectors using the space V_h that corresponds to this altered mesh it becomes possible to obtain an improved approximation of the weak solution (via solving $AU = F$).

Let the sets I and D be defined as per the above, and let $tol \geq 0$ be such that the norm $\| \cdot \|_{H^1(a,b)}$ of the error in the finite element solution should be less than this tolerance. We can obtain such a finite element solution if the following holds true.

$$\|u-u_h\|_{H^1(a,b)} \leq \frac{1}{2c_0}\|\frac{h}{p\cdot(p+1)}R(u_h)\|_{L^2(a,b)} = \frac{1}{2c_0}\Big(\int_a^b \big(\frac{h}{p\cdot(p+1)}R(u_h)\big)^2\ dx\Big)^{1/2}$$
$$= \frac{1}{2c_0}\Big(\sum_{1\leq k\leq n}\int_{x_{k-1}}^{x_k} \big(\frac{h}{d_k\cdot(d_k+1)}R(u_h)\big)^2\ dx\Big)^{1/2} \leq tol.$$

Requiring this then forces

$$\sum_{1\leq k\leq n}\int_{x_{k-1}}^{x_k} \big(\frac{h}{d_k\cdot(d_k+1)}R(u_h)\big)^2\ dx \leq (2c_0\cdot tol)^2.$$

Let us then define the local error indicator (on finite element k), denoted by err_ind_k, via

$$err\_ind_k = \int_{x_{k-1}}^{x_k} \big(\frac{h}{d_k\cdot(d_k+1)}R(u_h)\big)^2\ dx. \tag{3.11}$$

The basic strategy we adopt in order to refine the mesh (I,D) effectively is to iterate over the finite element domains I_k, evaluate the local error indicator, err_ind_k, and if the local error indicator is too large then we decide whether to spatially refine the k^{th} finite element (via replacing $I_k = [x_{k-1},x_k]$, d_k in I ,D with $I_{kl} = [x_{k-1},\frac{x_{k-1}+x_k}{2}]$, $d_{kl} = d_k$ and$I_{kr} = [\frac{x_{k-1}+x_k}{2},x_k]$, $d_{kr} = d_k$) or refining the polynomial degree of the k^{th} finite element (via replacing ($I_k = [x_{k-1},x_k]$, d_k) with ($I_k = [x_{k-1},x_k]$, d_k+1) in (I,D)). If the local error indicator is sufficiently small on some I_k then we leave I_k and d_k unchanged in I and D. Indeed, the terms sufficiently small and too large are sufficiently vague to warrant further explanation. Let us assume that initially we construct V_h by imposing that $I_k = [x_{k-1},x_k]$ is such that $\forall 1 \leq k \leq n,\ x_k - x_{k-1} = h$, for some fixed value $h > 0 \in \Re$. Futhermore recall that the mesh (I,D) comprises of n number of lagrangian finite elements (where n will increase if we spatially refine one or more finite elements when we refine the mesh). Then, regardless of the (finite) number of times we refine the mesh, we have that $\forall 1 \leq k \leq n,\ x_k - x_{k-1} = \frac{h}{2^{href_k}}$, whereby $href_k$ is the number of times we have h-refined a finite element that originally had a domain of length h to obtain the k^{th} finite element in the mesh. Now if we assume that

$$err\_ind_k \leq \frac{(2c_0\cdot tol)^2}{2^{href_k}n}$$

then we have that

$$\sum_{1\leq k\leq n} err\_ind_k \leq \sum_{1\leq k\leq n} \frac{(2c_0 \cdot tol)^2}{2^{href_k}n} \leq (2c_0 \cdot tol)^2$$

so that we have $\|u-u_h\|_{H^1(a,b)} \leq tol$ as required. Hence if the local error indicator on the domain of the k^{th} finite element is larger than $\frac{(2c_0 \cdot tol)^2}{2^{href_k}n}$, then we refine the corresponding finite element, otherwise we leave the element unchanged. We are still yet to define a method for deciding between h-refinement and p-refinement of a specific finite element.

The method that we use to decide between h and p refinement of a specific finite element method was initially presented and developed by T.P.Wihler (see [8]). We consider the lagrangian finite element (I_k, d_k). Generally, where $a < b \in \Re$ and $h = b - a$,
$\forall u \in H^1(a,b)$ it can be shown that

$$\|u\|^2_{\infty,(a,b)} \leq coth(1)\Big(h^{-1}\|u\|^2_{L^2(a,b)} + h\|u'\|^2_{L^2(a,b)}\Big)$$

where $\coth(\cdot)$ is the hyperbolic cotangent function and $\|u\|_{\infty,(a,b)} = max_{a\leq x\leq b}|u(x)|$. Thus if we let $K = (x_{k-1}, x_k)$ and $h_k = x_k - x_{k-1}$ we may define the element wise value $F_k[u_h]$ via $F_k[u_h] = 1$ if $u_h = 0$ on K, otherwise

$$F_k[u_h] = \|u_h\|^2_{\infty,K}\Big[coth(1)\Big(h^{-1}\|u\|^2_{L^2(a,b)} + h\|u'\|^2_{L^2(a,b)}\Big)\Big]^{-1}. \qquad (3.12)$$

Intuitively we can see that $F_k[u_h]$ will be close to 1 if u_h is smooth on K, and that if u_h varies strongly on K or has a steep derivative then $F_k[u_h]$ will be closer to 0. As such we may think of $F_k[u_h]$ as a smoothness measure of u_h on finite element (I_k, d_k). Thus if the k^{th} finite element has been marked for refinement, we find the value of $F_k[u_h]$ and if the value is larger than some predefined smoothness parameter (say 0.5) we perform p-refinement, otherwise h-refinement is performed on the element. Using this method we can, once a finite element solution u_h has been obtained, refine the mesh (I, D) by generating the altered sets I and D as described above, together which discretize the computational domain into a number of lagrangian finite elements which together define the refined space V_h as previously described. Once we have refined the space V_h we can then reconstruct the stiffness matrix and load vector, impose boundary conditions on the resulting system and solve this system again to obtain a more accurate finite element solution. This process can be iteratively applied until the FEM solution error (measured in the $\|\cdot\|_{H^1(\Omega)}$ norm) is bounded above by the desired tolerance .

Chapter 4

C++ Implementation

The C++ implementation detailed within this chapter applies the h-p finite element method to the problem described in (3.1-3.2). Despite this however, large sections of the code was designed to accommodate a more general type of PDE problem. As such if in the future the code is to be adapted to solve problems not of the form (3.1-3.2), little additional development will be required. We now show how certain mathematical entities are stored/implemented on the machine in this implementation.

Polynomial Representation: Let $p(\mathbf{x})$ be a multivariate polynomial of degree n having k independent variables, whereby $\mathbf{x} = \{x_1, x_2, ..., x_k\}$. Each term in the polynomial (negating the coefficients of each term) can be represented as a non-negative integer in base (n+1) consisting of k digits, for example the term $x_1^{\alpha_1} x_2^{\alpha_2} ... x_k^{\alpha_k}$ can be represented by the ordered digits $(\alpha_1, \alpha_2, ..., \alpha_k)$ whereby $\sum_{1 \leq i \leq k} \alpha_i \leq n$. Thus, the coefficients in the polynomial $p(\mathbf{x})$ may be represented through a finite lengthed vector, whereby the entry in the vector having index 0 is the coefficient of the term $(0, 0, ..., 0)$. We then have that the next entry in the vector having index 1 is the coefficient of the term $(0, 0, ..., 1) = x_k$. Generally, the entry in the coefficient vector having index i is the coefficient of the term that is represented by the ordered digits $(\alpha_1, \alpha_2, ..., \alpha_k)$, whereby $(\alpha_1, \alpha_2, ..., \alpha_k)$ is the $(i + 1)^{th}$ number (starting at and inclusive of zero) represented in base (n+1) such that $\sum_{1 \leq i \leq k} \alpha_i \leq n$. Using this ordering, we represent the general polynomial $p(\mathbf{x})$ via a vector of floating point numbers. The first entry of the vector defines the order/degree of the polynomial, the second entry in the vector is set to zero for this implementation but can be set to 1 to consider the tensor product of polynomials, the third entry defines the number of independent variables of the polynomial. The remaining values in the vector starting from position four correspond to the coefficients of the polynomial using the ordering as defined above, with the fourth entry of the vector corresponding to the coefficient of the constant term (which can be represented as $(0, 0, ..., 0)$). As an example, suppose we have the polynomial of order 2 of two variables defined by $p(x, y) = 1 + 2.3y + 3.4y^2 + 6.5xy + x^2$. This polynomial would then be represented via the vector $(2.0, 0.0, 2.0, 1.0, 2.3, 3.4, 0, 6.5, 1.0)$.

Matrix Representation, CSRMatrix Struct: The CSRMatrix struct was created as an efficient way to store (large sparse) matrices. It represents matrices using the compressed sparse row (CSR) format. All data members within the struct are publicly accessible. The default cosntructor, copy constructor, assignment operator and a useful standard constructor were all implemented as well as a destructor for freeing resources during object destruction. The matrix itself is defined within the struct via a pointer to a dynamically allocated array of floating point numbers (double data type), called matrix_entries, two pointers to dynamically allocated arrays of integers, called row_start and col_no respectively, and an integer called NoOfRows. The block of memory pointed to by matrix_entries holds the nonzero entries of the matrix in contiguous memory with left to right top to bottom ordering. col_no holds the column numbers of the corresponding elements in the matrix_entries array and row_start holds the 1 starting index position of the first non-zero element in each row within the matrix_entries array, with top to bottom row ordering. The final entry in row_start is equal to the integer value that is 1 greater than the total number of entries in the memory block pointed to by matrix_entries.

Integration over (a subset of) $\Omega = (a, b)$: We define the reference element, ref_elem = [0,1], and suppose we wish to integrate some function, $f(x)$, over the interval $D \in \Omega$, whereby $D = (D_1, D_2)$. We may then define the affine map $AM : [0, 1] \to D$ by $AM(\xi) = (D_2 - D_1)\xi + D_1$, a linear polynomial of 1 variable. Using integration by substitution we have that $\int_{D_1}^{D_2} f(x)\ dx = \int_0^1 f(AM(\xi))\ d\xi \cdot |J_{AF}|$, whereby $|J_{AF}|$ is the determinant of the jacobian of the affine map AM $AM : (0, 1) \to D$. In order to construct the stiffness matrix and load vector and also bound the error of our solution in the $\| \cdot \|_{H^1(\Omega)}$ norm, we are required (repeatedly) to evaluate the integral over a specific finite elements of an expression that involves up to the 2nd weak derivative of a linear combination of the basis functions of that particular finite element. Let ϕ_i be the i^{th} basis function polynomial within the set P_k that generates the space of shape functions for the lagrangian finite element having domain $I_k = [x_{k-1}, x_k]$. ϕ_i is of degree d_k so that $i \le (d_k + 1)$ and ϕ_i has d_k zeros contained within the set of equidistant points $\{[x_{k-1} = x_{k1}, x_{k2}, ..., x_{k(d_{k+1})} = x_k]\}$. Let $AM : [0, 1] \to I_k$ be the affine map from the reference element $[0,1]$ to I_k. Consider the set of points in the reference element $\{0, 1/d_k, 2/d_k, ..., 1\}$, and define the polynomial γ_i, of degree d_k, on the reference element such that $\gamma_i(t/d_k) = \phi_i(x_{k(t+1)})\ \forall 0 \leq t \leq d_k \in \mathbb{N} \cup \{0\}$, then it follows that $\gamma_i(\xi) = \phi_i(AM(\xi))\ \forall \xi \in [0,1],\ 1 \leq i \leq d_k + 1$. It follows by the chain rule (and the fact that Affine Maps are linear) then that $\frac{d^n\gamma_i}{d\xi^n} = \left(\frac{d\xi}{dx}\right)^n \cdot \frac{d^n\phi_i}{dx^n},\ \forall n \geq 1 \in \mathbb{N}$. Thus $\frac{1}{\left(\frac{d\xi}{dx}\right)^n} \cdot \frac{d^n\gamma_i}{d\xi^n}(\xi) = \frac{d^n\phi_i}{dx^n}(AM(\xi))$, and so using integration by substitution we have that $\int_0^1 \frac{1}{\left(\frac{d\xi}{dx}\right)^n} \cdot \frac{d^n\gamma_i}{d\xi^n}(\xi)\ d\xi \cdot |J_{AM}| = \int_{I_k} \frac{d^n\phi_i}{dx^n}\ dx$. It is far more computationally efficient to evaluate any integral quantity over the reference element and scale the obtained value appropriately as opposed to computationally evaluating integrals over varied domains.

We shall now describe the fundamental classes (and struct) within the implementation before explaining how these classes are used together to form the hp FEM code.

4.1 The BaseRep Class

Due to the way that polynomials are represented in this implementation, in order to perform differentiation on polynomials represented in this way it is necessary to have some method of performing addition operations and representing non-negative integers in an arbitrarily based number system. This is the primary purpose of the BaseRep class. The class has two arguments in its standard constructor, two references to constant integers, one integer specifying the base of the number system and one integer representing the number of digits to be used when representing non-negative integers in this number system. Thus since any instantiation of the BaseRep class uses a fixed finite number, say n, of digits to represent non-negative integers in using a particular base, say k, the instantiated object can only represent the non-negative integers ranging from 0 to $n^k - 1$. The class stores the base of its number system using a private int data type and has included within its implementation a standard and default constructor, copy constructor, assignment operator, overloaded increment operator, and comparison operators == and !=. The representation of the nonnegative integer is stored within a publicly accessible std::vector<int> object called Representation whose first entry (having index 0) corresponds to the least significant digit in the integer representation. Use of the overloaded increment operator increments the internal stored integer representation by 1. Two public member functions zero() and get_Base() are implemented that allow resetting the representation to zero and return an integer representing the base of the number system.

4.2 The Basis_Functions Struct

The basis functions of any finite element are stored in terms of their corresponding definition on the reference element. Every finite element has a corresponding degree (integer), d_k, such that the $d_k + 1$ polynomials defined on the finite element are all of degree d_k. As we iteratively refine the mesh, we may require polynomials of new degrees to be defined on specific finite elements. The Basis_Function struct can compute and store the polynomials defined on the reference element corresponding to a specific degree, d_k. The struct consists of two private static member objects, a static std::vector< std::vector< std::vector<double> > > object called BasisFunctions and a static std::vector<int> object called OrderedBasisFunction_Degrees. The i^{th} element in BasisFunctions is a std::vector< std::vector<double> > object which defines the set of polynomials (defined on the reference element) that correspond to the set of lagrangian finite element basis function polyonmials all having degree equal to the i^{th} element in the vector OrderedBasisFunction_Degrees. There are also two static public member functions, one called static void Generate_Basis_Functions(const int& Degree) which takes a reference to a constant integer as an argument which specifies the degree of the set of polynomials to be generated. The function then generates and stores the set of polynomial basis functions of the specified degree if this set has not already been computed and stored (this is easy to check since the vector containing all basis function degrees that are currently supported is static), otherwise if the set of basis function polynomials has already been generated and stored this function does nothing to the private static data members. The other static public member function is called static int maxDegree() which returns the maximum degree for which the corresponding set of basis function polynomials have already been computed and stored inside the object BasisFunctions. The final member function is public and not static. This function takes an integer as an argument which specifies a specific basis function polynomail degree and returns the corresponding set of basis function polynomials if already supported, otherwise it returns a vector of length 1 consisting of one vector also of length 1 with the first entry being zero. This struct is used to generate and store all of the sets of polynomials that are defined on the finite elements (whereby we store their representations on the reference element for consistency and computational efficiency) as we construct and iteratively enhance the space V_h. For each set of basis function polynomials corresponding to degree d_k that is required we only ever store the set of polynomials unique to that degree once within this struct.

4.3 The GaussianQuadrature Class

As has been previously explained we choose to evaluate all necessary integrals over the reference element $(0, 1)$ for consistency and computational efficiency. The method of numerical integration that we use is the Gaussian Quadrature formula [2]. Using the Gaussian Quadrature method we can only evaluate integrals exactly whereby the integrand is a polynomial having degree up to and including $2n - 1$, whereby n is the number of quadrature points and weights used in the gaussian quadrature evaluation of the integral. Suppose $p(x)$ is a polynomial having degree less than or equal to $2n - 1$. Then there exists a set of nodes $N = \{x_1, x_2, .., x_n\}$ and a corresponding set of weights $W = \{w_1, w_2, ..., w_n\}$ such that

$$\int_a^b p(x)\ dx = \frac{b-a}{2}\sum_{i=1}^{n} w_i f\Big(\frac{b-a}{2}x_i + \frac{a+b}{2}\Big). \tag{4.1}$$

The number of quadrature points and thus weights that we use is predetermined by the user as part of the input file. Thus the polynomial degree that the quadrature rule yields exact results for is determined by the user therefore if unsure it is better to specify a larger input parameter here than one too low. The GaussianQuadrature class has two static public data members, named Nodes and Weights, both of which are std::vector<double> objects. These static public members hold the ordered nodes and weights that are used during application of the gaussian quadrature rule to computationally evaluate a particular integral.

4.4 The Finite_Element Class

The finite element solution to the PDE problem is constructed as a piecewise function, whereby each section of the function definition is defined upon a separate finite element. As such, a Finite_Element class was created to store all the information required to define a lagrangian finite element contained within a one dimensional domain. The specific data types required by the main Finite_Element constructor are three seperate references to constant integer (const int& data), and a reference to a constant (standard C++ library) vector of vectors containing 8 byte floating point numbers (const std::vector< std::vector<double>>& VectVal). The three references to constant integers specify the type of lagrangian finite element being constructed (1D, 2D triangle, 3D tetrahedron), the degree of the polynomials to be defined on the element, and the spatial dimension of the element (which shall always be 1 in our case). The reference to a constant std::vector< std::vector<double> > input argument specifies the vertex points of the finite element, which in the 1D case is just a vector containing two vectors, both having size 1 whereby the first vector contains the real value specifying the left boundary point of the interval $I_k = [x_{k-1}, x_k]$ and the second vector contains the real value specifying the right boundary point. The convention used is that each vertex point $\in \Re^g$ is represented by a std::vector<double> of size g, whereby the i^{th} element of the vertex point is stored as the i^{th} element of the vector.

Each of the constructor arguments are stored as private objects (either as an int or std::vector< std::vector<double> >) internally within the class. There are three other private data members of the Finite_Element class and each of these are std::vector< std::vector<double> > objects. Generally, if the domain of the PDE problem (3.1-3.2) is given by $\Omega \in \Re^n$, then for any finite element having domain $I_k \in \Omega \in \Re^n$ the Affine Map from the unit reference element to I_k may be represented by n polynomials with each polynomial having n independent variables. As such we have named two of the three std::vector< std::vector<double> > private objects AffineMap and InverseAffineMap which store the vectors of polynomials required to represent the Affine Map from reference element to finite element and the inverse Affine Map in the other direction. The third private object std::vector< std::vector<double> > is called BasisFunctions and stores the ordered basis of polynomials belonging to the finite element. The relevant basis function polynomials are generated if necessary and supplied by the Basis_Functions struct. These polynomials are stored in terms of their definitions on the reference element, so that $\text{BasisFunction}_i(\xi) = \phi_i(AM(\xi))$, whereby ϕ_i is the ith basis function in the ordered basis $P = \{\phi_1, \phi_2, ..\}$ that generates the finite dimensional space of shape functions defined the finite element domain and $\xi \in [0, 1]$.

In order to obtain copies of these private data members there exists public member functions that return copies of each private member object. These public member functions have names that are self explanatory such as get_BFncs(), get_BFDegree(), get_Dim(), get_AffineMap(). There are also public member functions called Jacobian() and detJ() that return the jacobian matrix of the Affine transformation (from reference element to finite element) and the determinant of this transformation. The jacobian matrix is returned using the CSRMatrix struct object. Implementations of a copy constructor and an assignment operator are also present within this class. A public member function, called Differentiate_BF, is also implemented, that takes as input arguments an integer and two vectors of floating point numbers. The integer specifies the basis function under consideration and the two vectors together define a finite ordered derivative. One vector defines the order of spatial variables and the other vector defines the corresponding derivative orders of those spatial variables. For example an input argument list of std::vector<double> SpatialVars = $[1.0, 2.0, 1.0, 3.0]$, std::vector<double> DifferntialOrders = $[2.0, 1.0, 5.0, 4.0]$, int BFSpecifier = 3, would define the differential operator $\frac{d^{12}}{dx_1^2 dx_2 dx_1^5 dx_3^4}$. The member function Differentiate_BF returns a polynomial defined on the reference element, $p(\xi)$, such that $p(\xi) = D^{\alpha}(\phi_i)(AM(\xi))$ whereby ϕ is the i^{th} basis function belonging to the finite element object that was specified through the integer input argument and D^{α} is the differential operator defined through the two vector input arguments.

4.5 The Mesh Class

Recall that we defined the mesh can be as the ordered collection of lagrangian finite elements $(I, D) = \{(I_k, d_k) \mid 1 \leq k \leq n\}$. As described previosuly, each finite element has its own set of polynomial basis functions which are defined on the domain of the finite element and form a basis for the space of shape functions. The finite element solution restricted to a specific finite element is defined by a particular linear combination of the finite elements polynomial basis functions. For each finite element, the polynomial basis functions defined on the finite element are numbered from 1 to $d_k + 1$, where d_k is the degree of each of the polynomials. This numbering of the basis functions is unique to the finite element to which they belong and thus we shall call this ordering the local ordering of any given finite elements polynomial basis functions. When considering the collection of all finite elements, we may consider the extension of the basis functions to be defined over the whole of the computational domain Ω to obtain an ordered basis $\beta = \{\phi_1, ..\phi_N\}$ for the subspace $V_h \in H^1(\Omega)$, whereby each of the finite element basis functions (defined on a particular finite element) corresponds to a particular extended function in β. Thus we have a natural numbering of any basis function defined on any finite element such that it is numbered as the 1 starting index of its corresponding extended function occurring in the ordered basis β. We shall call this numbering/numerical ordering the mesh ordering. Thus for any finite element we may number its basis functions using the mesh ordering or local ordering. The way in which we store this information is through a connectivity array. The finite elements themselves can be numbered quit easily in our case, since each of their domains are closed intervals are non overlapping (ie the intersection of any two distinct finite element domains will be either the empty set or a set containing one element, a real number corresponding to a boundary point of both domains), with the finite element being numbered 1 whose domain is $I_1 = [a = x_0, x_1]$, the second finite element having domain $I - 2 = [x_1, x_2]$ and so on until we reach the final finite element numbered n whose domain is $I_n = [x_{n-1}, x_n = b]$. We can call this numbering of the finite elements the mesh numbering of the elements.

The Mesh class contains a std::unordered_map<int,Finite_Element> object called ElementContainer which stores each lagrangian finite element (I_k, d_k) as a Finite_Element object along with its mesh number, k, as a signed int data type. The object std::unordered_map<int,Finite_Element> is a hash table, which must posses certain general properties as specified by the C++ standard library but also has a compiler dependent implementation. When we insert an int - Finite_Element pair into ElementContainer, ElementContainer hashes the integer using the identity function, giving us the integer that we initially specified in the input pair. This hashed value (k, which is represented using the signed int data type) is then used to obtain the address of a specific std::unordered_map::bucket object in an array of std::unordered_map::bucket objects that we input our Finite_Element object into. Each std::unordered_map::bucket object in the array of (std::unordered_map::)buckets corresponds to a unique integer value and contains a linked list of Finite_Element objects whose keys hash (using in our case the identity function as a hash function) to the integer value associated with that std::unordered_map::bucket. The C++ standard requires that any inserted key-map pair have a unique key, but does not require the key to have a unique hash value so that any implementation must be able to resolve such cases (called collisions). Since the identity function is used as a hash function and since we use the mesh number as the key for each Finite_Element object (whereby each finite element in the mesh, (I_k, d_k), has a unique mesh number, k), no collisions occur in the ElementContainer object so that the linked list of Finite_Element objects in each (std::unordered_map::)bucket consists of only one Finite_Element object. Thus the ElementContainer object constructed in this manor has search, insertion and deletion time complexity $O(1)$.

The constructor of this class takes two const int& objects, one of these specifies the total number of finite elements in the mesh. Recall that the finite-dimensional space $V_h \subset H^1(\Omega)$ has an ordered basis of
$\beta = \{\phi_{11}^{ext}, .., \phi_{1d_k}^{ext}, \phi_1^{bext}, \phi_{22}^{ext}, .., \phi_{2d_k}^{ext}, \phi_2^{bext}, .., \phi_{(n-1)d_{n-1}}^{ext}, \phi_{(n-1)}^{bext}, \phi_{n2}^{ext}, .., \phi_{n(d_n+1)}^{ext}\} = \{\phi_1, \phi_2, ..., \phi_N\}$, whereby there exists a set of nodes/points in the domain $\Omega_C = [a, b]$, given by $\{a = t_1, t_2, .., t_N = b\}$, such that $\forall 1 \leq i, j \leq N, \ \ \phi_i(t_j) = \delta_{ij}$. The second reference to a constant integer in the constructor argument list specifies the total number of nodes, N (which is often called the degree of freedom of the FEM solution since it is also the dimension of the space V_h that contains the FEM solution). The other constructor arguments are a reference to a constant CSRMatrix struct which defines the connectivity array of the mesh, a reference to a constant std::unordered_map<int,Finite_Element> object which defines the hash table private data member named ElementContainer, and also a reference to a constant floating number (double data type) which gives the maximum domain length of any finite element in the mesh. The connectivity array is stored as a private CSRMatrix data member named ConnectivityArray, the integer values specifying the total number of nodes and finite elements are stored as private int types called NoOfElements and NoOfNodes respectively. The maximum domain length over all of the finite elements in the mesh is an important mesh property and is stored privately using a floating number (double data type) with the name h.

A copy constructor, default constructor, assignment operator, and destructor are all implemented. In order to access the private data members listed previously, the class contains several public member functions. These public member functions can be used to obtain copies of the private data members defining the connectivity array, the total number of nodes, the number of finite elements contained within the mesh, and the value of h. There is also a public member function called redefineMESH() that takes the same arguments as the constructor and assigns the private data members with the input arguments, this function can be thus used to assign all of the private data members with one function call.

4.6 Class synthesis

4.6.1 Initial uniform mesh generation

We now explain how the aforementioned classes and structs are used together to create the hp-FEM C++ implementation. Two input files exist, Computational Domain.dat and PDE_Definition.cpp, that must be adjusted before the code is compiled, linked, and run. One of these files (Computational Domain.dat) specifies 10 critical pieces of information that is related to the PDE problem definition. These 10 pieces of information are all represented by either an integer, floating point number, two floating point numbers, or a letter, so that the user does not have to do any monumental typing in order to use the code. The other file (PDE_Definition.cpp) defines the constants and functions $\varepsilon, b(x), c(x), f(x)$ found in (3.1) (along with the nonhomogeneous boundary conditions if required). Within Computational Domain.dat a density parameter and basis function degree is specified as a decimal number and integer respectively. The density parameter represents the maximum length of the domain of any finite element to be used within the mesh. The basis function degree specified in the input file is the degree that all basis function polynomials have initially on all finite elements. The first step in the code is to read these values (basis function degree and density parameter) from the input file Computational Domain.dat along with the two decimal values that together define the 1 dimensional domain $\Omega = (a, b)$. Once we have these values we can compute an initial constant finite element domain length that is less than or equal to the density parameter given in the Computational Domain.dat file that also divides the domain length $b-a$ with (practically) zero remainder. Once this constant finite element domain length is computed, we may construct and store an initial mesh (I, D) of numbered finite elements by constructing a Mesh object with appropriate constructor arguments.

The integer n such that $n\cdot$(constant finite element domain length)$=(b-a)$ is the number of finite elements that the mesh shall consist of. To construct the connectivity array and the std::unordered_map<int,Finite_Element> object that holds all of the Finite_Element objects of the mesh is then simple, we first instantiate a std::unordered_map<int,Finite_Element> with its default constructor and iterate over the integers $1 \leq i \leq n$, constructing the i^{th} Finite_Element and adding the integer - Finite_Element pair to the std::unordered_map<int,Finite_Element> object as we progress through the for loop. During each iteration we can input the appropriate entries into the i^{th} row of the connectivity matrix. Let the constant finite element domain length be denoted by DL, then the i^{th} finite element on the mesh is 1 dimensional (corresponding to a type specifier argument of 0 in the Finite_Element constructor) and has the domain $I_i = [a+((i-1)\cdot DL), a+(i\cdot DL)]$ with the degree of the basis function polynomials defined on I_i being defined by the value specified in the input file. We show below the code snippet that fills the connectivity array and element container object that are both passed to the Mesh object constructor during initial mesh generation.

```
for (int i=0; i<NoOfElements; ++i)
ConnectivityMatrix.row_start[i] = i*(BFDeg+1) + 1;
ConnectivityMatrix.row_start[(int)NoOfElements] =
                                  (((int)NoOfElements)*(BFDeg+1))+1;
std::unordered_map<int,Finite_Element> FEContainer;
int iter=0, NodeNumber=1;
for (int i=0; i<NoOfElements;++i) {
  std::vector< std::vector<double> > Vertex_Points;
  double leftVertex = LeftBoundaryPoint+(i*uniformVertexInterval)
  ,rightVertex=LeftBoundaryPoint+((i+1.0)*uniformVertexInterval);
  if (i==NoOfElements-1)
  rightVertex = RightBoundaryPoint;
  std::vector<double> leftPoint = {leftVertex};
  std::vector<double> rightPoint = {rightVertex};
  Vertex_Points = {leftPoint, rightPoint};
  int LINESPECIFIER=0; // defines the 1D line finite element type.
  Finite_Element currFE(LINESPECIFIER,BFDeg,Vertex_Points,Dim);
  for (int j=0; j<BFDeg+1; ++j) {
    ConnectivityMatrix.matrix_entries[iter] = NodeNumber;
        ConnectivityMatrix.col_no[iter]=j+1;
    ++iter;
        ++NodeNumber;
  }
  --NodeNumber;
  FEContainer.emplace((i+1),currFE);
}
```

Since we now know the total number of finite elements in the mesh, n, and the degree of the polynomials on each element, d, the degree of freedom of any finite element solution (or total number of nodes in the mesh) is equal to $(n \cdot d) + 1$. We can then generate the initial Mesh object using its standard constructor.

4.6.2 Assembly of the stiffness matrix and load vector

Once the initial mesh is generated through constructing a Mesh object as described above we can then proceed to construct the stiffness matrix A and load vector F. Recall that
$\beta = \{\phi^{ext}_{11}, .., \phi^{ext}_{1d_k}, \phi^{bext}_{1}, \phi^{ext}_{22}, .., \phi^{ext}_{2d_k}, \phi^{bext}_{2}, .., \phi^{ext}_{(n-1)d_{n-1}}, \phi^{bext}_{(n-1)}, \phi^{ext}_{n2}, .., \phi^{ext}_{n(d_n+1)}\} =$
$\{\phi_1, \phi_2, ..., \phi_N\}$ is an ordered basis for V_h, thus A is an $N \times N$ sparse matrix.

The stiffness matrix is stores as a CSRMatrix struct, called STIFFNESSMATRIX, and as such the maximum number of nonzero entries in the matrix must be determined to be able to dynamically allocate a sufficient but reasonable amount of memory for the matrix_entries, col_no, and row_start public data members. The positions in the stiffness matrix A also need to be obtained for each entry in STIFFNESSMATRIX.matrix_entries so that row_start and col_no data members can be filled accordingly. To do this we make use of the information provided in the connectivity array, which is stored inside the Mesh object. As previously detailed, the i^{th} row of the connectivity array contains all of the basis function 1 starting indices whose corresponding basis functions in β take nonzero values on the i_{th} finite element. Since the stiffness matrix A is an $N \times N$ matrix, we may associate to each position (i, j) corresponding to element $A_{i,j}$ in the matrix a unique integer value, we do this via the function $Hash\_Fnc(i, j) = (i-1) * N + j, \ \forall 1 \leq i, j \leq N$. We then declare a std::list<int> object, called positionHashVals lets say, and loop over each row of the connectivity array. For each row in the connectivity array, we take all of the nonzero entries in the current row (which will be integer values), and for each ordered pair of integers (k,l) whereby k and l are non-zero integers occurring in the current row of the connectivity matrix, we hash this ordered pair of integers using the function $Hash\_Fnc(i, j)$ to obtain an integer and insert this integer into the positionHashVals objects ensuring that insertion of an integer is ordered (whereby integer value increases as the element index is increased) and only takes place if that integer is not already currently within the positionHashVals object. This process gives us a std::list<int> object whose size determines the maximum number of non-zero entries in the stiffness matrix and whose entries implicitly define (through the hash function) the position of each non-zero element in the stiffness matrix. Furthermore due to the way the CSRMatrix struct stores matrices, the ordering of elements in the positionHashVals object corresponds to the ordering of the matrix elements in the STIFFNESSMATRIX.matrix_entries dynamically allocated memory block.

We show below a code snippet which produces the positionHashVals std::list<int> object (called HashedNodePairs below).

```
int n = COMPUTATIONALMESH.get_NoOfNodes();
int NoOfElements = COMPUTATIONALMESH.get_NoOfElements();
int* ConnArray_row_start = COMPUTATIONALMESH.ConnArray_rowptr();
double* ConnArray_mentries = COMPUTATIONALMESH.ConnArray_entryptr()
CSRMatrix STIFFNESSMATRIX;
```

```
std::list<int> HashedNodePairs;
for (int i=0; i<NoOfElements; ++i) {
  for (int j=(ConnArray_row_start[i]-1);
           j<(ConnArray_row_start[i+1]-1); ++j) {
  for (int k=(ConnArray_row_start[i]-1);
           k<(ConnArray_row_start[i+1]-1); ++k) {
      orderedInsert(HashedNodePairs,
                    hashFunc(ConnArray_mentries[j],
                    ConnArray_mentries[k],
                    n) );
    }
  }
}
```

Now that we have the maximum number of nonzero entries of the stiffness matrix (and their associated positions) and can determine the number of nonzero elements in each row of the stiffness matrix (since any element in row i of the stiffness matrix will have its position hashed to an integer value ranging between ((i-1)*N + 1) and i*N) that is stored within the positionHashVals object), it is now possible to construct the stiffness matrix CSRMatrix struct and allocate an appropriate amount of memory pointed to by matrix_entries and initialize all of the entries to zero. The col_no and row_start data members of the connectivity array can also be allocated and correctly initialized at this point, so that we only need to assign the appropriate floating point numeric values to the memory block pointed to by matrix_entries to completely represent the stiffness matrix.

In order to fill the stiffness matrix so that $A_{i,j} = a(\phi_j, \phi_i)$, we once again loop over each row of the connectivity array, whereby for the i^{th} row there exists $(d_i+1)^2$ distinct ordered pairs (j,k), whereby j and k are non-zero integer values of some entry in the current row (not necessarily distinct) that each represent the position in the vector β of some basis functions ϕ_j and ϕ_k. For each ordered ordered pair (j,k), we first compute $a(\phi_j, \phi_k)_i$ and find the index in the memory block pointed to by STIFFNESSMATRIX.matrix_entries, which we shall denote by int index, that corresponds to position (k,j) in the stiffness matrix A before incrementing STIFFNESSMATRIX.matrix_entries[index] by $a(\phi_j, \phi_k)_i$. Doing this for every row in the connectivity matrix constructs the stiffness matrix A as required.

In general for the i^{th} row of the connectivity matrix containing the nonzero positive integers k and j, we find the index corresponding to position (k,j) in STIFFNESSMATRIX.matrix_entries by hashing the ordered pair (k,j) (via

$Hash\_Fnc(k,j)$) and finding the index of this hashed value within the positionHashVals std::list<int> object. We still need to evaluate $a(\phi_j,\phi_k)_i$. Recall that

$$a(\phi_j,\phi_k)_i = \int_{x_{i-1}}^{x_i} \varepsilon\frac{d\phi_j}{dx}\frac{d\phi_k}{dx} + b\frac{d\phi_j}{dx}\phi_k + c\phi_j\phi_k \ dx$$

and that we wish to evaluate this integral on the reference element $(0,1)$. To do this we first obtain the functions γ_j', γ_k', γ_j, γk such that $\forall\xi \in [0,1]$, $\gamma_j'(\xi) = \frac{d\phi_j}{dx}(AM(\xi)), \gamma_j(\xi) = \phi_j(AM(\xi)), \gamma_k'(\xi) = \frac{d\phi_k}{dx}(AM(\xi)), \gamma_k(\xi) = \phi_k(AM(\xi))$, whereby $AM : [0,1] \to I_i = [x_i - 1, x_i]$ is the affine map from the reference element to the i^{th} finite elements domain, I_i. We obtain these functions using the Differentiate_BF() and get_BFncs() public member functions of the Finite_Element class, so that we then have that
$\forall\xi \in [0,1]$

$$\begin{aligned}\varepsilon\gamma_j'(\xi)\gamma_k'(\xi) + b(AM(\xi))\gamma_j'(\xi)\gamma_k(\xi) + c(AM(\xi))\gamma_j(\xi)\gamma_k(\xi)& \\ = (\varepsilon\frac{d\phi_j}{dx}\frac{d\phi_k}{dx} + b\frac{d\phi_j}{dx}\phi_k + c\phi_j\phi_k)(AM(\xi)).&\end{aligned}$$

It then follows that

$$\begin{aligned}\int_0^1 \varepsilon\gamma_j'(\xi)\gamma_k'(\xi) + b(AM(\xi))\gamma_j'(\xi)\gamma_k(\xi) + c(AM(\xi))\gamma_j(\xi)\gamma_k(\xi) \ d\xi \cdot |J_{AM}| =& \\ \int_{I_i} \varepsilon\frac{d\phi_j}{dx}\frac{d\phi_k}{dx} + b\frac{d\phi_j}{dx}\phi_k + c\phi_j\phi_k \ dx = a(\phi_j,\phi_k)_i.&\end{aligned}$$

Assuming a sufficiently large number, n, of gaussian quadrature points was defined by the user, we may evaluate this quantity exactly using the gaussian quadrature rule. Specifically we have that (whereby $W = \{w_1,..,w_n\}$, $X = \{x_1,..,x_n\}$ defines the n quassian quadrature weights and points for the interval [-1,1] and $q_i = 0.5x_i + 0.5, \ \forall 1 \le i \le n$)

$$\begin{aligned}&\int_0^1 \varepsilon\gamma_j'(\xi)\gamma_k'(\xi) + b(AM(\xi))\gamma_j'(\xi)\gamma_k(\xi) + c(AM(\xi))\gamma_j(\xi)\gamma_k(\xi) \ d\xi = \\ &0.5 \cdot \sum_{i=1}^n w_i\Big(\varepsilon\gamma_j'(q_i)\gamma_k'(q_i) + b(AM(q_i))\gamma_j'(q_i)\gamma_k(q_i) + c(AM(q_i))\gamma_j(q_i)\gamma_k(q_i)\Big).\end{aligned}$$

Once we have obtained the value of $a(\phi_j,\phi_k)_i$ using the above method and have the corresponding zero starting index, int index, of the value $Hash\_Fnc(k,j)$ in the positionHashVals std::list<int> object, we then add the contribution of

$a(\phi_j, \phi_k)_i$ to position (k,j) in the stiffness matrix by incrementing $A_{k,j}$ by the value $a(\phi_j, \phi_k)_i$. This is done via STIFFNESSMATRIX.matrix_entries[index]+= $a(\phi_j, \phi_k)_i$.

Recall that the load vector $F \in \Re^N$ (whereby $\beta = \{\phi_1, \phi_2, ..., \phi_N\}$ is an ordered basis for $V_h \in H^1(\Omega)$) is such that $F_i = \ell(\phi_i)$, whereby $\ell(\phi_i) = \langle f, \phi_i \rangle_{L_2(\Omega)}$. N is equal to the number of nodes in the mesh (or degrees of freedom of the mesh) and is obtainable through the get_NoOfNodes() public member function of the Mesh class. In order to construct the load vector F we first dynamically allocate N floating point numbers (double data type) and initialize the allocated memory block to all zeros. We then iterate over each of the rows of the connectivity array, whereby for the i^{th} row ($1 \leq i \leq$ total number of rows of the connectivity array) there exists $d_i + 1$ distinct nonzero integer values $j,\ t \leq j \leq t + d_i$ as entries within the i^{th} row of the connectivity array that each define the 1 starting position of the basis function $\phi_j \in \beta$. For each nonzero integer j occurring in the i^{th} row we evaluate the value $\ell(\phi_j)_i$ and increment the j^{th} entry of F by this value. Since the j values occur in the row ordered in the same order that the local basis functions are ordered, we may use the Mesh class public member fucntion find_FE() to return the i^{th} Finite_Element object in the Mesh and then we may call the Finite_Element public function get_BFncs() to obtain the basis functions γ_j such that $\forall \xi \in [0,1],\ \gamma_j(\xi) = \phi_j(AM(\xi))$ (where AM is the affine map from $[0,1]$ to I_i). Thus it follows that

$$\int_0^1 \gamma_j(\xi) f(AM(\xi))\ d\xi \cdot |J_{AM}| = \ell(\phi_j)_i$$

and we can thus use the gaussian quadrature rule to evaluate this (assuming that f is not too irregular), so that

$$\int_0^1 \gamma_j(\xi) f(AM(\xi))\ d\xi = 0.5 \cdot \sum_{i=1}^{n} w_i \Big(\gamma_j(q_i) f(AM(q_i)) \Big)$$

whereby $W = \{w_1, .., w_n\},\ X = \{x_1, .., x_n\}$ defines the n quassian quadrature weights and associated points for the interval [-1,1] and $q_i = 0.5x_i + 0.5,\ \forall 1 \leq i \leq n$. We then increment the j^{th} value of the load vector F by $\ell(\phi_j)_i$ via $F[j-1]+ = \ell(\phi_j)_i$. This completes the construction of the load vector.

4.6.3 hp-FEM solution and upper a posteriori error bound evaluation

Once we have constructed the stiffness matrix A and load vector F using the above methods, we then adjust the stiffness matrix and load vector to reflect the problem specific boundary conditions and then solve corresponding system $AU = F$ for the solution vector U. The algorithms that are currently supported for solving the system $AU = F$ are the conjugate gradient method, Thomas algorithm, Gauss-Seidel method, and Gaussian Elimination. The user details whether or not the stiffness matrix is symmetric (since A is symmetric iff the bilinear form $a(\cdot, \cdot)$ is symmetric) and from this information one can use Sylesters criterion to determine whether the stiffness matrix is symmetric positive definite or not. The conjugate gradient method is very efficient and converges if the stiffness matrix A is symmetric positive definite. The Thomas algorithm is used if the stiffness matrix is tridiagonal and diagonally dominant and if the stiffness matrix is just diagonally dominant then the Gauss-Seidel method is used. If the stiffness matrix A fails to fall in one of the aforementioned specification categories then one can use gaussian elimination to solve the system. Once the solution vector U is obtained $U = \{U_1, ..., U_N\}$ whereby N is the degree of freedom of the mesh (equivalently the dimension of the space V_h or the total number of nodes in the mesh given by the return value of the get_NoOfNodes() Mesh class public member function). The finite element solution (discrete weak solution) u_h is then given by

$$u_h = \sum_{i=1}^{N} U_i \phi_i \tag{4.2}$$

whereby $\beta = \{\phi_1, ..., \phi_N\}$ is an ordered basis for $V_h \subset H^1(\Omega)$.

Recall that we may obtain an upper bound on the difference between our finite element solution u_h and the solution u to the weak formulation that is contained within the infinite dimensional Hilbert Space $H^1(\Omega)$ via

$$\|u - u_h\|_{H^1(a,b)} \leq \frac{1}{2c_0} \| \frac{h}{p \cdot (p+1)} R(u_h) \|_{L^2(a,b)}$$

whereby $R(u_h) = f - (-\varepsilon u_h'' + b(x)u_h' + c(x)u_h)$. We have that

$$\| \frac{h}{p \cdot (p+1)} R(u_h) \|^2_{L^2(a,b)} = \sum_{i=1}^{n} \int_{x_i-1}^{x_i} \Big(\frac{h}{p_i \cdot (p_i+1)} \cdot (f - (-\varepsilon u_h'' + b(x)u_h' + c(x)u_h)) \Big)^2 dx \tag{4.3}$$

whereby n is the total number of finite elements in the mesh. In order to calculate (4.3), we loop from i=1 to i=n, and for each i we obtain a copy of the i^{th} Finite_Element object in the mesh via calling the find_FE() public member function of the Mesh class. Once we have the current Finite_Element object, the basis functions (contained within the set P_i) defined on the element can be obtained through calling the get_BFncs() pubic member function, whereby if $P_i = \{\phi_{i1}, \phi_{i2}, .., \phi_{i(d_i+1)}\}$, the get_BFncs() function returns the ordered set $\{\gamma_1, \gamma_2, ..., \gamma_{d_i+1}\}$ such that if AM is the affine map from the reference element to I_i then $\forall 1 \leq j \leq d_i + 1,\ \ \gamma_j(\xi) = \phi_{ij}(AM(\xi)),\ \ \forall \xi \in [0,1]$. We then call both γ_j and ϕ_{ij} the j^{th} basis function of the finite element i, since they both can be considered as representations of the same function. Where the finite element number i is clear, we denote ϕ_{ij} by ϕ_j. Let d_i denote the degree of the basis functions of the i^{th} finite element, so that the i^{th} row of the connectivity array contains $d_i + 1$ non zero integers, $t \leq j \leq t + d_i$. Each j specifies the basis function $\phi_j \in \beta$ which corresponds to the $(l+1)^{th}$ basis function defined on the finite element, whereby $t+l = j$. We may then define u_h restricted to the i^{th} finite element via
$\forall x \in I_i$

$$u_h(x) = \Big(\sum_{l=0}^{d_i} U[t+l-1] \cdot BF_{l+1}\Big)(Inverse\_AM(x))$$

whereby the entry in position $(i,1)$ of the connectivity array is equal to t, and BF_{l+1} refers to the $(l+1)^{th}$ basis function of the i^{th} finite element, represented on the reference element. Inverse_AM is the affine map from the finite element domain I_i to the reference element $[0,1]$. Similarly if $\gamma_j^{(s)}$ is the function returned by the Differentiate_BF() public member function that defines on the reference element the derivative of order s w.r.t x of the j^{th} basis function defined on the i^{th} finite element ($\phi_{ij} = BF_j$) so that

$$\gamma_j^{(s)}(\xi) = \frac{d^{(s)}BF_j}{dx^s}(AM(\xi)),\ \ \forall 1 \leq j \leq d_i + 1,\ \ \xi \in [0,1],$$

then it follows that the s^{th} derivative of u_h restricted to the finite element i can be represented on the reference element via
$\forall x \in I_i$

$$u_h^{(s)}(x) = \sum_{l=0}^{d_i} U[t+l-1] \cdot \gamma_j^{(s)}(Inverse\_AM(x)).$$

It follows then that for any $1 \leq i \leq n$ (where n is the total number of finite elements in the mesh), where p_i is the degree of the polynomial basis functions of element i, q is the number of quadrature points that the user has defined,

$W = \{w_1, .., w_q\}$, $X = \{x_1, .., x_q\}$ defines the q quassian quadrature weights and associated points for the interval [-1,1] and $q_i = 0.5x_i + 0.5$, $\forall 1 \leq i \leq q$, that

$$\int_{x_i-1}^{x_i} \Big(\frac{h}{p_i \cdot (p_i+1)} \cdot (f - (-\varepsilon u_h'' + b(x)u_h' + c(x)u_h))\Big)^2 dx =$$
$$\int_0^1 \Big(\frac{h}{p_i \cdot (p_i+1)} \cdot \Big(f(AM(\xi)) - \big[-\varepsilon\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}^{(2)}(\xi)\big)$$
$$+ b(AM(\xi))\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}^{(1)}(\xi)\big)$$
$$+ c(AM(\xi))\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}(\xi)\big)\big]\Big)\Big)^2 \; d\xi \cdot |J_{AM}| =$$
$$|J_{AM}| 0.5 \cdot \sum_{i=1}^{q} w_i \Big(\frac{h}{p_i \cdot (p_i+1)} \cdot \Big(f(AM(q_i)) - \big[-\varepsilon\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}^{(2)}(q_i)\big)$$
$$+ b(AM(q_i))\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}^{(1)}(q_i)\big)$$
$$+ c(AM(q_i))\big(\sum_{l=0}^{d_i} U[t+l-1]\gamma_{l+1}(q_i)\big)\big]\Big)\Big)^2.$$

This then gives us a method of computing the value, $\|\frac{h}{p\cdot(p+1)}R(u_h)\|^2_{L^2(a,b)}$, so that we only need to take the square root and multiply by $\frac{1}{2c_0}$ to finally obtain the upper bound on the error

$$\|u - u_h\|_{H^1(a,b)} \leq \frac{1}{2c_0}\|\frac{h}{p\cdot(p+1)}R(u_h)\|_{L^2(a,b)}.$$

After obtaining the error in the FEM solution measured in the $\|\cdot\|_{H^1(a,b)}$ norm, we may compare this value to our error tolerance, so that if the error is larger than the user defined acceptable tolerance, we can perform h-p refinement on the mesh, otherwise the program terminates.

4.6.4 hp mesh refinement

We now explain how the implementation performs h-p refinement on the mesh object Mesh, thus allowing us to re-generate the stiffness matrix A and load vector F from the finite elements associated with the refined mesh. The Mesh object, which we shall call MESH, has a public member function called get_NoOfElements() which returns the total number of finite elements that are contained within the MESH. Once we have this, we can loop from $i = 1$ to $i =$ MESH.get_NoOfElements() and for each i we obtain the values of η_i and $criterior_i$, whereby η_i and $criterior_i$ are the local error indicator and local error indicator criteria of the i_{th} finite element in the mesh, so that if $\eta_i{>}criteria_i$ then the i_{th} finite element needs refining and we store the number i in a vector of integers to signify that element i in the mesh needs refining. Recall that

$$\eta_i = \int_{x_{i-1}}^{x_i} \big(\frac{h}{d_i \cdot (d_i+1)} R(u_h)\big)^2 \; dx$$

and

$$criteria_i = \frac{(2c_0 \cdot tol)^2}{2^{href_i} n}$$

whereby c_0 is the coercivity constant in the Lax Milgram theorem, h is the maximum length of any finite element domain in the mesh, d_i is the degree of the basis polynomials of the i^{th} finite element, tol refers to the user defined tolerance that the finite element solution must adhere (measured in the $\| \cdot \|_{H^1(a,b)}$ norm), $href_i$ is how many times the i^{th} finite element has been h-refined and n is the total number of finite elements in the mesh (MEHS.get_NoOfElements()). $href_i = \log_2(\frac{IUDL}{x_i - x_{i-1}})$, where IUDL is the initial uniform domain length used when the mesh was generated and all of the finite elements had identical lengths. We have already explained how it is possible to compute η_i using the gaussian quadrature rule and the Differente_BF() public member fucntion of the Finite_Element class, and $criteria_i$ is simple to compute (recall that h-refinement performed on an element divides the domain equally into two separate finite element domains). Thus, we loop over each of the finite elements, calculate η_i and $criterior_i$ for each element and if $\eta_i{>}criteria_i$ then we store the integer i in a std::vector<int>, called refinement_vect lets say, so that we have implicitly stored all of the finite elements that require refinement. The process of placing an integer i into the refinement_vect is often called marking element i for refinement.

Recall that the Mesh class has the following public member function

```
void redefineMESH(const int& NoOfElements, const int& NoOfNodes,
                  std::unordered_map<int, Finite_Element>& EC,
                  CSRMatrix& CArray, double& h_);
```

that assigns to each of its (private and public) data members a corresponding value in the functions input argument list. This function then redefines the mesh to match its input arguments exactly. Thus to refine the mesh (the space V_h), we only are required to construct objects that together define the refined mesh and call the redefineMESH() public member function with these newly constructed objects. As can be seen from the input argument list above, the objects that we must create are two integers, one referring to the total number of elements in the refined mesh and the other integer defining the total number of nodes in the refined mesh (which is equivalent to giving the degree of freedom of the redefined mesh). We must also construct all of the Finite_Element objects in the refined mesh and insert these elements into a std::unordered_map<int, Finite_Element> object along with their corresponding integer value representing the mesh number of that particular inserted finite element. The connectivity array for the refined mesh must also be created as well as the h value of the refined mesh (whereby h is the floating point number giving the maximum finite element domain length in the mesh). We shall now explain how each of these objects are created.

We first declare three vector objects in order to store the connectivity array, CArray, in CSR format (the entries in these vectors shall then be copied over to a CSRMatrix struct that compactly represents the connectivity array). The vectors are thus a std::vector<double> called entries and two std::vector<int> objects called col_no and row_start. Recall that the finite elements in the mesh are numbered based on their domains, whereby the finite element having domain $I = [a, x_1]$, where $\Omega = (a, b)$ is numbered 1 and in general if the finite element k has domain $I_k = [x_{k-1}, x_k]$ then the finite element with domain $I_{k+1} = [x_k, x_{k+1}]$ will be numbered $k + 1$. A std::unordered_map<int, Finite_Elemnt> object, called FEContainer, is also declared to store the Finite_Element objects of the refined mesh in. We first declare an integer type called $NodeNumber$ and assign it a value of 0, this variable keeps track of the number of nodes in the refined mesh. We must also define another int type to keep track of the number of elements in the refined mesh and assign it an initial value of 0. We then iterate over all of the elements in the unrefined/original mesh MESH, looping from i=1 to i=MESH.get_NoOfElements(), whereby generally for the i^{th} element we perform the following steps:

1) Check if the integer i is contained within the refinement_Vect which lists all element numbers to be refined. If i has been marked for refinement go to 2), otherwise go straight to 5).

2) Decide whether to perform h-refinement on the element or p-refinement. If h-refinement has been decided go to 3), if p-refinement has been decided go to 4)

3) Create two Finite_Element objects, called currFE1 and currFE2 for example, such that the domain of currFE1 is $I_{i1} = [x_{i-1}, \frac{x_i+x_{i-1}}{2}]$ and the domain of currFE2 is $I_{i2} = [\frac{x_i+x_{i-1}}{2}, x_i]$, whereby the domain of the i^{th} finite element in the mesh was $I_i = [x_{i-1}, x_i]$. The basis function polynomials of currFE1 and currFE2 all have identical degrees to the basis function polynomials of the i^{th} finite element in the original mesh. Thus currFE1 and currFE2 store the lagrangian finite elements (I_{i1}, d_i) and (I_{i2}, d_i) respectively, whereby d_i was the basis function polynomial degree of the i^{th} finite element in the original unrefined mesh. We then emplace the Finite_Elements objects currFE1 and currFE2 into FEContainer using the element numbering scheme as describe above for the integer value that defines the key that maps to a specific Finite_Element object within FEContainer. Suppose d_i was the degree of the basis function polynomials defined on the i^{th} finite element of the original unrefined mesh, then we set the 1st to $d_i + 1$ columns of the next two rows in the connectivity array to the ordered sets $\{NodeNumber, NodeNumber + 1, .., NodeNumber + d_i\}$ and $\{NodeNumber + d_i, NodeNumber + d_i + 1, .., NodeNumber + d_i + d_i\}$ respectively via appending the vectors entries, col_no and row_start with the appropriate values. We then and set $NodeNumber$ to $NodeNumber + d_i + d_i$. We also have to keep track of the total number of elements that we have added to the FEContainer object (2 have been added during this iteration) as well as the total number of Nodes that are present in the mesh (we have added a total of $2d_i$ nodes to the mesh during this iteration if $i > 1$, otherwise we have added $1 + 2d_i$ nodes if $i = 1$ which occurs when we are h-refining the very first element in the original mesh).

4) Create one Finite_Eement object, named currFE, that has identical domain to the i^{th} finite element in the original mesh. The $d_i + 2$ basis polynomials functions of currFE all have degree $d_i + 1$, whereby all of the basis function polynomials of the i^{th} element in the original unrefined mesh had degree d_i. Thus currFE stores the lagrangian finite element $(I_i, d_i + 1)$, however the mesh number of this finite element in the refined mesh may not necessarily be the same as the integer i, whereby (I_i, d_i) was the i^{th} finite element in the unrefined/original mesh. Once the finite element currFE has

been constructed it must be emplaced with its (refined) mesh number into the FEContainer std::unordered_map<int, Finite_Elemnt> object. We then set the 1st to $d_i + 2$ columns of the next row in the connectivity array to the ordered set $\{NodeNumber, NodeNumber + 1, .., NodeNumber + d_i + 1\}$ and set $NodeNumber$ to be equal to $NodeNumber + d_i + 1$. We must also keep track of the total number of elements and nodes in the refined mesh, whereby we have added 1 new element to the mesh in this iteration. If $i > 1$, we have added $d_i + 1$ new nodes to the mesh, otherwise we have added $d_i + 2$ new nodes to the mesh during this iteration.

5) Create one Finite_Eement object, named currFE that has identical domain and basis polynomial degree to the i^{th} finite element in the original mesh. Emplace this Finite_Element object into FEContainer, whilst keeping track of the total number of elements and nodes in the refined mesh. 1 element will be added to the mesh during this iteration. Let d_i be the basis function polynomial degree of the i^{th} finite element in the original mesh. d_i number of nodes will be added if $i > 1$ during this iteration and and $d_i + 1$ number of nodes will be added if $i = 1$. We then set the 1st to $d_i + 1$ columns of the next row in the connectivity array to the ordered set $\{NodeNumber, NodeNumber + 1, .., NodeNumber + d_i\}$ and set $NodeNumber$ to be equal to $NodeNumber + d_i$.

Applying the above steps 1-5 at each iteration, from i=1 to i=i=MESH.get_NoOfElements(), constructs all of the objects required for the input argument list of the redefineMESH() public member function. That is, the resulting objects std::unordered_map<int, Finite_Elemnt> FEContainer, connectivity array (which can trivially be copied from being represented via thee std::vectors to a single CSRMatrix struct), int NoOfElements, int NoOfNodes, double h define the refined mesh. Whereby h is the maximum domain length of any Finite_Element object contained within FEContainer. Once we have called redefineMESH() with these created objects we can then resolve the discrete weak formulation using the refined space V_h which is represented by the refined Mesh object.

During step 2 it is required to decide between h-refinement and p-refinement of a particular element whereby the element has been marked for refinement, we now describe how this decision process is implemented.

Suppose the i^{th} finite element in the mesh has been marked for refinement and let n = MESH.get_NoOfNode(), so that $\beta = \{\phi_1, .., \phi_n\}$ is a basis for $V_h \subset H^1(\Omega)$. Then if CArray denotes the connectivity array of the mesh, stored within MESH, and d_i denotes the degree of the basis function polynomials defined on the i^{th} finite element then the FEM solution restricted to the i^{th} finite element is given by

$$u_h = \sum_{j=1}^{d_i+1} U[C_{i,j} - 1]BF_j$$

whereby BF_j is the j^{th} basis function polynomial of the i^{th} finite element, defined on I_i and $U \in \Re^n$ is the solution vector of the system $AU = F$ that implicitly defines u_h. The ordering of the basis functions defined on the finite element is identical to the ordering of the polynomials within the vector returned by the (Finite_Element classes) public member function get_BFncs(). Thus the derivative of order s of the FEM solution u_h w.r.t x defined on the finite element is given by

$$u_h^{(s)} = \sum_{j=1}^{d_i+1} U[C_{i,j} - 1]BF_j^{(s)}.$$

As previously detailed, let $\gamma_j^{(s)}$ be the polynomial defined on the reference element [0,1], for each $1 \leq j \leq d_i + 1$, such that if AM is the affine map from $[0, 1]$ to I_i we have that
$\forall \xi \in [0, 1]$

$$\gamma_j^{(s)}(\xi) = \frac{d^{(s)}BF_j}{dx^{(s)}}(AM(\xi)).$$

For any positive integer s, $\gamma_j^{(s)}$ can be obtained by calling the public member function Differentiate_BF() (with appropriate input arguments) of the i^{th} Finite_Element which can be obtained via calling $MESH.find\_FE(i)$. Recall that $I_i = [x_{i-1}, x_i]$ is the domain of the i^{th} finite element. We evaluate the value $F_i[u_h]$, whereby $h_i = x_i - x_{i-1}$ and

$$F_i[u_h] = \|u_h\|^2_{\infty,(x_{i-1},x_i)} \Big[coth(1)\Big(h_i^{-1}\|u_h\|^2_{L^2(x_{i-1},x_i)} + h_i\|u_h'\|^2_{L^2(x_{i-1},x_i)}\Big)\Big]^{-1}.$$

$\|u_h\|^2_{\infty,(x_{i-1},x_i)}$ is the ℓ infinity norm (squared) on the interval (x_{i-1}, x_i) and is computationally approximated via considering 100 uniformly spaced points on the reference element of the function

$$\sum_{j=1}^{d_i+1} U[C_{i,j} - 1]\gamma_j$$

whereby γ_j is the j^{th} basis function polynomial represented on the reference element and U is the solution vector of the system $AU = F$. We have that

$$\|u_h^{(s)}\|^2_{L^2(x_{i-1},x_i)} = 0.5 \cdot |J_{AM}| \cdot \Big(\sum_{j=1}^{gp} w_j \cdot \big(\sum_{k=1}^{d_i+1} U[C_{i,k} - 1]\gamma_k^{(s)}(q_i)\big)^2\Big), \quad (4.4)$$

whereby $\forall 1 \leq i \leq gp$, $gauss_i$ represents the ith gaussian quadrature point on the interval [-1,1] and $q_i = 0.5 gauss_i + 0.5$. We notationally assume in (4.4) that s is a nonnegative integer and that for s=0 $\gamma_j^{(0)} = \gamma_j, \;\; \forall 1 \leq j \leq d_i + 1$.

We show below the function that evaluates $F_i[u_h]$ within the implementation. The pointer coeffVals points to the solution vector U which is stored in dynamically allocated memory.

```
double smoothness_measure(const int& element_index,
                          Mesh& COMPUTATIONALMESH, double* coeffVals)
{
   Finite_Element FE = COMPUTATIONALMESH.find_FE(element_index+1);
   int NoOfBasisFunctions = FE.get_BFDegree()+1;
   int* CArray_row_start = COMPUTATIONALMESH.ConnArray_rowptr();
   double* CArray_entries = COMPUTATIONALMESH.ConnArray_entryptr();
   std::vector< std::vector<double> > localBasisFunctions =
                                                     (FE.get_BFncs());
   std::vector<double> scalars, svar = {1.0}, dorder={1.0};
   int row_start_index = CArray_row_start[element_index]-1;
   for (int j = row_start_index;
                         j<row_start_index+NoOfBasisFunctions; ++j)
   scalars.push_back(coeffVals[ ((int)(CArray_entries[j]))-1 ]);
   std::vector<double> approximate_solution =
                              LinearComb(scalars,localBasisFunctions);
   std::vector< std::vector<double> > localfirst_derivatives;
   for (int j=0; j<(FE.get_BFDegree()+1); ++j)
   localfirst_derivatives.push_back(FE.Differentiate_BF(svar,
                                                       dorder, j+1));
   std::vector<double> approxSol_derivative = LinearComb(scalars,
                                            localfirst_derivatives);
   double lInfinityNormSquared =
                        pow(linfinityNorm(approximate_solution),2.0);
   double l2NormapproxSol = 0.0, l2NormapproxSolderiv=0.0,
   h=((FE.get_VertexPoints())[1][0]-(FE.get_VertexPoints())[0][0]);
   std::vector<double> currPoint;
   for (int j=0; j<GaussianQuadrature::Nodes.size(); ++j) {
    currPoint = {(0.5*GaussianQuadrature::Nodes[j]) + 0.5};
    l2NormapproxSol+= GaussianQuadrature::Weights[j]*
                pow(Eval(currPoint, approximate_solution),2.0);
    l2NormapproxSolderiv+= GaussianQuadrature::Weights[j]*
                pow(Eval(currPoint, approxSol_derivative),2.0);
  }
  l2NormapproxSol*=(0.5*FE.detJ());
  l2NormapproxSolderiv*=(0.5*FE.detJ());
  double coth_1_ = (exp(1.0) + exp(-1.0))/(exp(1.0) - exp(-1.0));
  double smoothnessMeasure = lInfinityNormSquared/(coth_1_*(
        ((1.0/h)*l2NormapproxSol) + (h*l2NormapproxSolderiv) ) );
  return smoothnessMeasure;
}
```

Once we have obtained the value of $F_i[u_h]$ (whereby if $u_h = 0$ on the i^{th} finite element we let $F_i[u_h] = 1$), since $F_i[u_h]$ is close to 1 if u_h is fairly smooth and if u_h varies strongly over the i^{th} finite element or has a steep derivative $F_i[u_h]$ is closer to zero, p-refinement of the i^{th} finite element is selected if $F_i[u_h]$ is greater than some predefined smoothness parameter (such that $0<smoothness\_param<1$) and is h-refined if $F_i[u_h]$ is less than or equal to the smoothness parameter. Changing the smoothness parameter changes whether the solver is more bias towards h-refinement or p-refinement, with a large smoothness parameter corresponding to h-refinement bias and a low smoothness parameter preferring p-refinement. Thus if one suspects the FEM solution to a particular problem to exhibit a high degree of smoothness or to be quite irregular the smoothness parameter can be set accordingly. If no insight is obvious we can set the smoothness parameter to 0.5 as a default value. We have thus explained how to decide between h-refinement and p-refinement of a particular finite element in the mesh that was marked for refinement previously and in doing so have fully elucidated steps 1-5 that must be applied for each element in the unrefined mesh in order to refine the mesh. When we redefine the mesh we are allowing the possibility of a solution to be computed which more accurately represents the true solution of the weak formulation which is contained within the infinite dimensional space $H^1(\Omega)$. Once the mesh has been refined using the above method, if we re-construct the stiffness matrix and load vector using this refined mesh to obtain the new system $AU = F$, the solution to this system is should give a closer fit to the true weak solution (as a result of the new solution having more degrees of freedom over the entire mesh thus allowing it to more accurately approximate the true weak solution). We can continue to iteratively refine the mesh in this way, obtaining more accurate FEM solutions each time, until we obtain a FEM solution u_h that satisfactorily approximates the true weak solution u.

Chapter 5

Test Problem

In order to test the developed C++ implementation of the h-p finite element method we have chosen a specific test problem of the form (5.1). A comprehensive explanation of the test problem that we have used to verify the C++ implementation can be found in the paper [3]. The test problem takes the form

$$-\varepsilon u'' + cu = 1, \ \forall x \in \Omega = (0,1) \tag{5.1}$$

whereby both ε and c are constant values in $\Re$. For our test problem we have that $\varepsilon = 10^{-5}$ and that the constant c has the value of 1. The test problem required that the solution takes the following values on the boundary, $u(0) = u(1) = 0$, thus we have homogeneous dirichlet boundary conditions. It can be shown that the analytic solution of this problem is defined by

$$u(x) = \frac{exp(\frac{x}{\sqrt{\varepsilon}})}{exp(\frac{1}{\sqrt{\varepsilon}}+1)} - \frac{exp(\frac{-x}{\sqrt{\varepsilon}})exp(\frac{1}{\sqrt{\varepsilon}})}{exp(\frac{1}{\sqrt{\varepsilon}}+1)} + 1 \tag{5.2}$$

whereby ε and c take on the values defined above. We may define the "energy" norm on the space $H^1(\Omega)$, $\forall v \in H^1(\Omega)$ $(\Omega = (0,1))$

$$\|v\|^2_{E,(0,1)} = \varepsilon\|v'\|^2_{L_2(0,1)} + c\|v\|^2_{L_2(0,1)}.$$

It can be shown that using this energy norm to measure the error in the FEM solution ($u - u_h$, whereby u defines the analytic solution given previously and u_h is the FEM solution obtained via the hp FEM), we can obtain the following sharp a posteriori upper bound defined below

$$\|u - u_h\|^2_{E,(0,1)} \leq \sum_{j=1}^{N} \eta^2_{K_j} + \frac{1}{\varepsilon \cdot (p_j(p_j+1))}\|f - \prod f\|^2_{L_2(K_j)} \tag{5.3}$$

where

$$\eta_{K_j} = \frac{1}{\sqrt{\varepsilon \cdot (p_j(p_j+1))}} \|\big(\prod f - (-\varepsilon u_h'' + cu_h)\big)w_j^2\|_{L^2(K_j)}. \qquad (5.4)$$

defines the local error indicator for the j^{th} finite element for this test problem. For more details concerning this a posteriori error bound, see [6, section 3.5.2]. w_j is a function defined on the domain of the j^{th} finite element $I_j = [x_{j-1}, x_j]$ and is defined by $\forall x \in [x_{j-1}, x_j],\ w_j(x) = (x_j - x)(x - x_{j-1})$. N gives the total number of finite elements in the mesh, $K_j = (x_{j-1}, x_j)$ where the j^{th} finite elements domain is given by $I_j = [x_j - 1.x_j]$, the function f refers to the function given on the right hand side of the test problem (5.1) so that in our case $f = 1$. The function given by $\prod f$ defines the L^2 projection of the function f on to the space V_h. The value p_j refers to the degree of the basis function polynomials defined on the j^{th} finite element. Recall that $\beta = \{\phi_1, \phi_2, ..., \phi_n\}$ is an ordered basis for the finite dimensional space V_h. The L^2 projection of the function f on to the space V_h is a function contained within the space V_h, say we denoted this by f_h, such that
$\forall v_h \in V_h$

$$\langle f_h, v_h\rangle_{L_2(\Omega)} = \langle f, v_h\rangle_{L_2(\Omega)}. \qquad (5.5)$$

Similar to how we found a function satisfying the weak formulation, we can find such a function f_h from constructing a matrix M, whereby $M_{i,j} = \langle \phi_j, \phi_i\rangle_{L_2(\Omega)}$ and a vector L, whereby $L_i = \langle f, \phi_i\rangle_{L_2(\Omega)}$. So that M is an nxn matrix and L is a vector contained within the space $\Re^n$ such that if we solve the matrix system $MT = L$ for the solution vector T, then we have that the l^2 projection of the function f is given by the following linear combination of ordered basis functions in the basis β

$$\prod f = f_h = \sum_{i=1}^{n} T_i \phi_i. \qquad (5.6)$$

whereby n defines the dimension of the space V_h, equivalent to the number of basis functions contained within the bordered basis β. We shall not in detail explain how, computationally, we may construct the matrix M and the vector , since this process is very similar to the construction process of how we construct the stiffness matrix A and load vector F to obtain the (iterative) FEM solutions. Thus, we assume that we have obtained the solution vector T that defines the L^2 projection of the function f onto the space V_h, and we shall now , in detail, explain how we can compute the upper bound on the error of the FEM solution measured the "energy" norm given by (5.3).

We first obtain the total number of finite elements within the mesh via using the public member function provided by the Mesh class that is called get_NumberOfElements(). This returns an integer value specifying the total number of finite elements within the mesh, let us call this integer n. We then iterate over each of these finite elements using a standard for loop. During each iterate, from $1 \leq i \leq n$, we wish to compute the value

$$\eta_{K_i}^2 + \frac{1}{\varepsilon \cdot (p_i(p_i+1))} \|f - \prod f\|_{L_2(K_i)}^2,$$

whereby, as previously given ,

$$\eta_{K_i} = \frac{1}{\sqrt{\varepsilon \cdot (p_i(p_i+1))}} \|\big(\prod f - (-\varepsilon u_h'' + c u_h)\big) w_i^2\|_{L^2(K_i)}.$$

We first start via obtaining the polynomials basis function degree for this particular (i^{th}) finite element. This can be obtained via first obtaining the i^{th} finite element in the mesh via using the public member function provided via the Mesh class called find_FE() that takes a 1 starting integer index and returns the corresponding Finite_Element object. Once we have the i^{th} Finite_Element object within the mesh, we can then use the get_BFDegree() public member function provided by the Finite_Element class to obtain the basis function polynomial degree corresponding to finite element i within the mesh. Once we have this basis function degree value, we may compute and store the floating point number $\frac{1}{\varepsilon \cdot (p_i(p_i+1))}$ whereby p_i defines the basis function polynomial degree corresponding to the i^{th} finite element. We then wish to evaluate

$$\|\big(\prod f - (-\varepsilon u_h'' + c u_h)\big) w_i^2\|_{L^2(K_i)}.$$

Let γ_j be defined on the reference element [0,1] such that $\forall 1 \leq j \leq d_i + 1$, (whereby d_i is the basis function polynomial degree corresponding to he i^{th} finite element)

$$\gamma_j(\xi) = \phi_j(AM(\xi)). \;\forall \xi \in [0,1]$$

whereby AM defines the affine map from the reference element [0,1] to the domain of the i^{th} finite element and ϕ_j denotes the j^{th} basis function polynomial of the i^{th} finite element represented on the interval I_i. Similarly, we define $\gamma_j^{(s)}$ (for any positive integer s) on the reference element such that $\forall \xi \in [0,1]$

$$\gamma_j^{(s)}(\xi) = \frac{d^{(s)}\phi_j}{dx^{(s)}}(AM(\xi)).$$

Thus it follows that for any positive integer s, the derivative (of order s) of the finite element solution u_h restricted to the i^{th} finite element can be represented on the reference element as a linear combination of $\gamma_j^{(s)}$ functions. More specifically, we have that

$$\Big(\sum_{j=1}^{d_i+1} U[C_{i,j}-1]\gamma_j^{(s)}(\xi)\Big) = u_h^{(s)}(AM(\xi)),\ \ \forall\xi\in[0,1].$$

The restriction of $\prod f$ to the i^{th} finite element can be represented on the reference element like so,

$$\Big(\sum_{j=1}^{d_i+1} T[C_{i,j}-1]\gamma_j(\xi)\Big) = \prod f(AM(\xi)),\ \ \forall\xi\in[0,1].$$

Thus it follows that
$\forall\xi\in[0,1]$

$$\Big(\Big(\Big(\sum_{j=1}^{d_i+1} T[C_{i,j}-1]\gamma_j(\xi)\Big)+\varepsilon\Big(\sum_{j=1}^{d_i+1} U[C_{i,j}-1]\gamma_j^{(2)}(\xi)\Big)-c\Big(\sum_{j=1}^{d_i+1} U[C_{i,j}-1]\gamma_j(\xi)\Big)\Big)(w_i(AM(\xi))^2)\Big)^2$$
$$\Big(\Big(\prod f(AM(\xi)) - (-\varepsilon u_h(AM(\xi))'' + cu_h(AM(\xi)))\Big)(w_i(AM(\xi)))^2\Big)^2$$

so that we have, via the gaussian quadrature rule (whereby we assume that we have a total number of gq quadrature points $\{q_1,..,q_{gq}\}$ and associated weights $\{w_1,..,w_{gq}\}$ for the the interval [0,1]) that

$$\|\Big(\prod f - (-\varepsilon u_h'' + cu_h)\Big)w_i^2\|^2_{L^2(K_i)} =$$
$$0.5\cdot|J_{AM}|\cdot\sum_{k=1}^{gp} w_k\Big(\Big(\Big(\sum_{j=1}^{d_i+1} T[C_{i,j}-1]\gamma_j(q_k)\Big)+\varepsilon\Big(\sum_{j=1}^{d_i+1} U[C_{i,j}-1]\gamma_j^{(2)}(q_k)\Big)-$$
$$c\Big(\sum_{j=1}^{d_i+1} U[C_{i,j}-1]\gamma_j(q_k)\Big)\Big)(w_i(AM(q_k))^2)\Big)^2.$$

whereby C is the connectivity array, U is the solution vector that defines the finite element solution, T is the solution vector (of the system $MT = L$) that defines the L^2 projection of f and ε and c are the constants defined in the test problem. γ_j has been previously defined and can be obtained via calling the get_BFncs() public member function of the i^{th} finite element in the mesh to return a vector consisting of the ordered basis polynomials of the finite element represented on the reference element. Thus γ_j is the j^{th} polynomial (stored as a std::vector<double> object) in the vector returned by the get_BFncs() public member function of the i^{th} finite element. Similarly $\gamma_j^{(2)}$ can be obtained from calling the Differentiate_BF() public member function of the i^{th} finite element with input parameters specifying the return of the second derivative of the j^{th} basis function, represented on the reference element. As previously stated $|J_{AM}|$ is the determinant of the jacobian matrix of the affine map that maps from the reference element to the i^{th} finite element and can be obtained from calling the public member function $detJ()$ of the i^{th} finite element.

We store the value of η_{K_i} for each element, as these values will be used if further mesh refinement is required. We can then calculate $\|f - \prod f\|^2_{L_2(K_i)}$, since via the gaussian quadrature rule we have that

$$\|f-\prod f\|^2_{L_2(K_i)} = 0.5\cdot|J_{AM}|\sum_{j=1}^{gq} w_j\Big(f(AM(q_j))-(\sum_{k=1}^{d_i+1} T[C_{i,k}-1]\gamma_k(q_j))\Big)^2.$$

This then allows us to evaluate $\|u-u_h\|^2_{E,(0,1)}$ as we iterate over each finite element, calculating $\eta^2_{K_i} + \frac{1}{\varepsilon\cdot(p_i(p_i+1))}\|f-\prod f\|^2_{L_2(K_i)}$ for each element. Since the test problem uses the energy norm to measure the error in our FEM solution, we use a different strategy for deciding which elements need to be refined (see [8] for more details). Given that we have the local error indicator, η_{K_i} for each finite element stored in a vector, we choose to refine every finite element, j, such that (whereby N defines the total number of finite elements in the mesh)

$$\max_{1\le i\le N}\eta_{K_i} \le \theta^{-1}\eta_{K_j}.$$

whereby $0<\theta<1$ is a element marking parameter that we take to be 0.5 in the test problem. We then refine the element using 0.5 as the smoothness measure parameter using the procedure detailed in the previous chapter.

Since the analytic solution is available and differentiable, it was possible to compute the actual energy norm of the error $(u-u_h)$ as well as the upper bound of the error $(u-u_h)$ which was obtained using the aforementioned method after each iterative mesh refinement step. The smoothness measure parameter and element marking parameter were both set to 0.5 and the algorithm was run for 20 iterations, so that 20 increasingly accurate FEM solutions were obtained as a result of iterative hp mesh refinement. The initial uniform finite element domain length was 0.25 and the initial polynomials basis function degree was 1 for all the four finite elements on the initial domain. Plots showing the error estimate and actual true error of each iterative FEM solution and the final mesh after 20 iterations are displayed below.

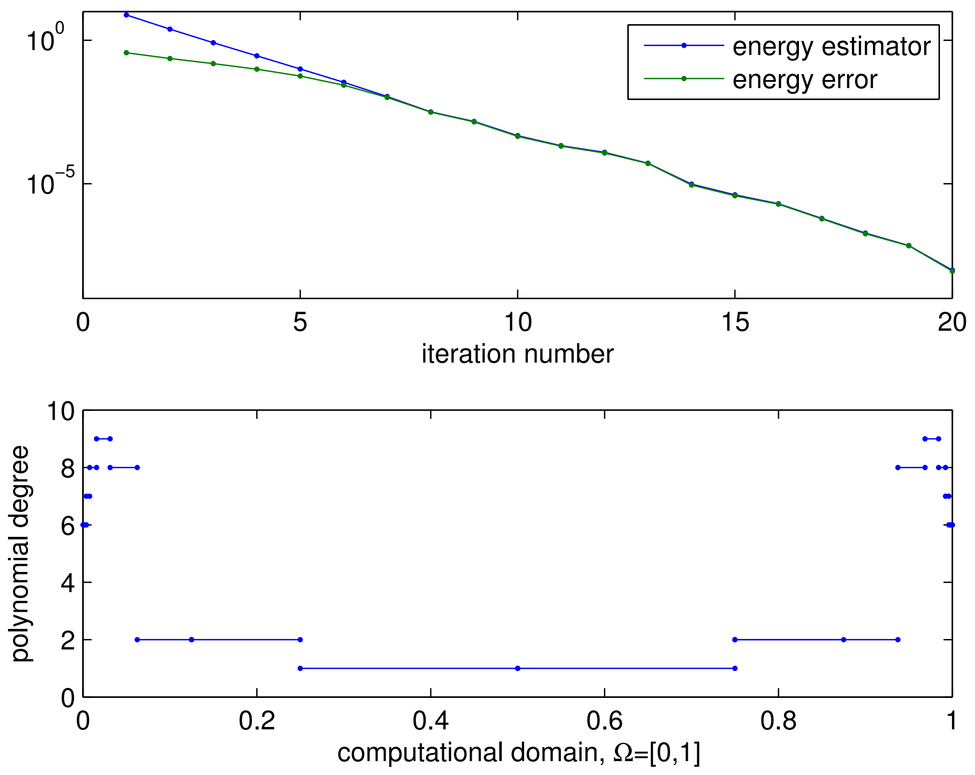


Figure 5.1: hp-mesh after 19 refinement steps (bottom) and performance of hp-FEM (top).

As displayed in the figure above, we can see an exponential convergence of the iterative hp FEM algorithm towards the true weak solution u, which lives in the infinite dimensional space $H^1(\Omega)$. The "energy" error estimator $\sum_{j=1}^{N} \eta_{K_j}^2 + \frac{1}{\varepsilon\cdot(p_j(p_j+1))}\|f - \prod f\|_{L_2(K_j)}^2$ is shown to be a sharp upper bound of the error in the FEM approximation given by the "energy" error $\|u-u_h\|_{E,(0,1)}^2$, this is expected from the result given previously (5.3). The hp FEM algorithm can only display this exponential convergence rate up to machine precision, and after 20 iterations the algorithm was taking a significant amount of time to process the next FEM solution so that we terminated program execution after 20 iterates. Recall that an element, once marked for refinement, is h-refined if the obtained approximate solution or its derivative varies significantly over the element and is p-refined if the approximate solution is sufficiently smooth. To obtain a more visual understanding of why the adaptive algorithm refined the mesh in the above manner, we consider the analytic solution u below:

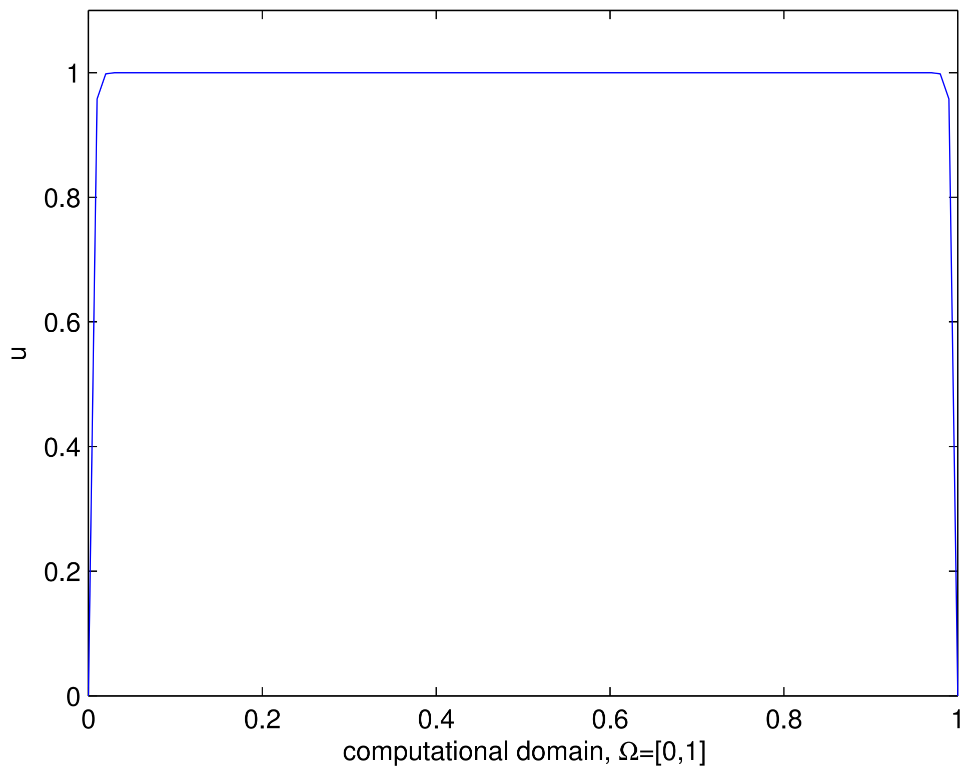


Figure 5.2: Exact solution, u, of (21) with $\varepsilon = 10^{-5}$, $\ c = 1$.

Due to the analytic function varying severely at each end of the domain, we would expect a mesh capable of accurate analytic solution approximation to have finite elmenets of relatively small domain length at these locations as a result of repeated h refinement of the mesh being applied during past iterations to attempt to capture the irregularity in the analytic solution. Similarly in the area of the domain where the analytic solution has a constant value of one, we would expect the mesh to have been refined much less than at the ends of the domain, since this constant valued behaviour of the analytic solution can be captured exactly through finite elements having basis function polynomial degrees of one, $(I_k, 1)$, which was the initial basis function degree of all polynomials in the mesh. Due to the ability of the initial mesh being able to capture the behaviour of the analytic solution exactly on the two finite elements (each of domain length 0.25) either side of the point 0.5, the mesh has not been refined on these two elements adjacent to the middle of the domain. In order to sufficiently capture the behaviour of the analytic solution in the domain where its value changes abruptly from one, the basis function polynmials defined on the finite elements at these locations must be of high order, this can be seen in the figure detailing the mesh after 19 iterations above.

Chapter 6

Conclusion and Further Work

The original intention of this dissertation project was to develop a freely accessible hp-FEM code and provide a description of the finite element method and developed C++ hp-FEM implementation. The hp-FEM code can be applied to the following relatively general PDE problem:

$$-\varepsilon \cdot \frac{d^2u}{dx^2} + b(x) \cdot \frac{du}{dx} + c(x) \cdot u = f \qquad \forall x \in \Omega = (a, b), \tag{6.1}$$

whereby ε>0 and b, c, and f are as described in (3.1), and nonhomogeneous or homogeneous dirichlet boundary conditions are defined on the boundary of the domain. Although one may consider this problem rather restrictive, the implementation has been developed in such a way that it would be very reasonable to extend the code to be applicable to a more general class of PDE problems. As previously outlined the finite element method is used in many practical scenarios to obtain approximate solutions to physically relevant PDE problems that occur in the sciences and engineering.

The hp-FEM code that was developed can successfully apply the hp-FEM to problems of the above type. In order to specify the particular PDEproblem to be solved, the user must input information into two separate files. One of these files is named Computational Domain.dat and is used to input general properties of the problem, such as boundary conditions, specification of the boundary domain and initial mesh parameters. The other input file is named PDE_Definition.cpp and contains the definitions of the parameter ε and functions b, c, $nonhomogeneousDirichlet()$, $analytic\_solution()$, $analytic\_solution\_deriv()$ and f , which together implicitly defines the PDE problem. Once these files have been amended by the user, the user must then compile and link all of the relevant files together before running the final executable. Dependent upon the information provided by the user within the two input files, the hp-FEM code will either repeatedly refine the mesh (using h-p refinement) until a solution is found that adheres to a certain (user defined) tolerance or the code will obtain a FEM solution of the PDE problem using the initial mesh that was defined by the user within the Computational Domain.dat file. The exact process of how the implementation decides upon h or p refinement for a specific marked element is explained in detail within previous chapters (The hp-FEM and C++ Implementation chapters). After the implementation has obtained a FEM solution for a particular mesh iteration, a sharp a posteriori upper bound on the error induced is outputted to the terminal to give an indication of how well the current mesh has approximated the true solution. The developed hp-FEM code was verified using a test problem that had an exact (boundary layer) solution. For this test case the hp-FEM code refined the initial mesh appropriately to capture the behaviour of the boundary layer exact solution and as a result a good level of performance (exponential convergence) up to machine precision was obtained.

The hp-FEM C++ implementation and associated fundamental underlying mathematics has been described sufficiently well to allow independent individuals to amend the present implementation or start to develop their own implementation for their own needs. As such this written dissertation along with the hp FEM code shall be made publicly accessible (see appendix for details of location).

As alluded to previously there currently exists two input files that the user must amend before compiling, linking, and executing the code. Thus in order to currently use the implementation a user must have a certain level of knowledge with regards the C++ language. Ideally the user should be able to use the code without needing to compile or link any files, just by running one single program. Out of the two current input files one of these is simply read (Computational.dat) whilst the other is compiled and linked during the formation of the final executable (PDE_Definition.cpp). The functions defined in PDE_Definition.cpp currently all take one floating point number as input, say x, and return a floating point number out which is some function of the input argument. Thus it is possible for the user to represent these functions using a simple protocol of human readable syntax and semantics in the Computational Domain.dat file (as opposed to the user writing these functions into the PDE_Definitions.cpp file using the C++ language before compiling and linking relevant files to solve the problem), the main executable/program could then parse these definitions using a consistent protocol and write to the PDE_Definition.cpp file these definitions before invoking system calls to compile and link all of the relevant files together and run the final executable. Implementing this facility would allow the code to be more user friendly, as only 1 input file would have to be amended by the user and zero knowledge of C++ or compiler/linker usage would be needed. A fundamental component in implementing this functionality is parsing an arbitrary single variable function and writing this function to the file (PDE_Definition.cpp) using the C++ language. An algorithm that could be used to achieve this is the shunting-yard algorithm. The shunting yard algorithm takes as input a mathematical expression and reads the expression one token at a time from left to right, where a token is either a parenthesis, operator ($+, -, *, /, \wedge$), number, or function. The shunting algorithm defines a process of reading and storing tokens in such a manor as to be able to reconstruct the original arbitrary mathematical expression, the expression can be stored and reconstructed using RPN (Reverse Polish Notation). Below is a code snippet that declares the Token class within a header file that could be used in an expression parsing function.

```
class Token
{
  public:
    // Constructor
    Token(const int& Token_TypeSpecifier,
        const std::string& stringVal,
        const double& numericVal);
    // Default Constructor
    Token();
    // Copy Constructor
    Token(const Token& tCOPY);
    // Assignent Operator
    Token& operator=( Token tASSIGN);
    // get data members
    int get_type() const;
    double get_numeric() const;
    std::string get_string() const;
    // Determine token type
    bool is_operation_token() const;
    bool is_numeric_token() const;
    bool is_function_token() const;
    bool is_parenthesis_token() const;
    // Destructor
    Token();

  private:
    int token_type;
    double numeric_value;
    std::string string_value;
};
```

whereby the token_type integer private data member would define whether the token is a parenthesis, operator, number, or function token and the numeric_value and string_value private data members would specify the specific token value (for the numeric token type and function and operator token types).

References

[1] R.A.Adams, J.F.Fournier, 2009, Sobolev Spaces, Singapore, Elsevier
[2] D.Bau and L.N.Trefethen, 1997, Numerical Linear Algebra (p.285-292), Philadelphia, SIAM
[3] S.C.Brenner and L.R.Scott, 2007, The Mathematical Theory of Finite Element Methods, New Delhi, Springer
[4] P.G.Ciarlet, 1979, The Finite Element Method for Elliptic Problems, New York, North-Holland
[5] K.Irvine, 2014, Assembly language for 86 PROCESSORS, New Jersey, Pearson Education Inc.
[6] C.Schwab, 1998 , p and hp FEM Theory and Application to Solidand Fluid Mechanics, Oxford, Oxford University Press
[7] B.Stroustrup, 2008, Programming principles and practice using C++, Boston, Pearson Education Inc.
[8] Thomas.J.Wihler, 2010, An hp-adaptive strategy based on continuous Sobolev embeddings, Journal of Computational and Applied Mathematics 235(2011)27312739, Elsevier
[9] O.C.Zienkiewicz, 2013, The Finite Element Method for Fluid Dynamics, Oxford, Butterworth-Heinemann
[10] O.C.Zienkiewicz, 2005, The Finite Element Method for Solid and Structural Mechanics (Volume 2), Oxford, Butterworth-Heinemann (Elsevier imprint)

Appendix

Attached on this page is a CD-RW containing the hp-FEM implementation developed by the author. When the author or the associated academic institution no longer requires exclusive hp-FEM code or dissertation access (which shall be approximately after 01/11/2016), all documents/codes that were used in the production/development of the dissertation/hp-FEM code shall be placed in the public domain (https://github.com/efidoalo/hp-FEM-cpp).