An Object-Oriented Approach to the Implementation of Finite Element Methods in C++

G14SCD

MSc Dissertation in

Scientific Computation

2016/17

School of Mathematical Sciences

University of Nottingham

Sammy Petros

Supervisor: Prof. Paul Houston

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

Abstract

An implementation of the finite element method in an object-oriented manner. We introduce the theory underpinning finite element formulations of partial differential equations, before looking in detail at the design and structure of an object-oriented finite element package in C++. Numerical quadrature rules for a variety of element types are then studied theoretically and practically to assist in the assembly of finite element stiffness matrices and load vectors. Finally, a look into the application of the theory and implementation to solve a variety of examples in both one and two dimensions and complete an a posteriori error analysis.

Contents

1 Introduction **5**

2 Weak Derivatives and Sobolev Spaces **7**

2.1 Spaces of Integrable Functions 7

2.2 Weak Derivatives 10

2.3 Sobolev Spaces 12

3 Weak Formulation of Elliptic Boundary Value Problems **17**

3.1 Weak Formulation of a PDE 17

3.2 The Lax-Milgram Theorem 19

3.3 General Elliptic PDE Example 21

4 Construction of the Finite Element Method **26**

4.1 The Finite Element Space 26

4.2 The Finite Element Method 29

4.3 The Elements 31

4.4 Hierarchical Bases 36

5 Implementation in C++ **46**

5.1 General Class Design 46

5.2 Mesh 48

5.3 Element 49

5.4 FE_Solution 53

5.5 QuadratureLibrary 59

5.6 SparseMatrix 60

6 Numerical Quadrature on Elements **67**

6.1 Gauss-Legendre Quadrature 67

6.2 Quadrature on Intervals 70

6.3 Quadrature on Quadrilaterals 71

6.4 Quadrature on Triangles 72

7 Numerical Solution to Finite Element Formulations **74**

7.1 One Dimensional Poisson Equation 74

7.2 A Posteriori Error Analysis for 1D Poisson Equation 84

7.3 Two dimensional Inhomogeneous Modified Helmholtz Equation 86

8 Conclusions **91**

A Code **93**

A.1 Basis Functions 93

A.2 Triangle 96

A.3 Quadrilateral 98

A.4 FE_Solution 99

A.5 QuadratureLibrary 101

A.6 SparseMatrix 106

A.7 Example population of local element stiffness matrices and load vector . . 107

1 Introduction

The finite element method is an extremely versatile and widely used numerical technique. It has many applications in the world of engineering, such as fluid dynamics and the numerical solutions to Navier-Stokes equations. From a theoretical stand point, there is a lot of active research being done into areas such as *hp*-adaptive mesh refinement. There are many powerful packages, written in variety of languages, that exist for evaluating partial differential equations using the finite element method. These include packages such as FEniCS (Logg, Mardal, and Wells, 2012) and deal.ii (Bangerth, Hartmann, and Kanschat, 2007), which are two of the most utilised libraries that exist.

This dissertation aims to provide an introduction to the theory behind the finite element method and apply this knowledge to create an object-oriented finite element package in C++11. In the first section, we give an overview of theoretical aspects such as appropriate functions spaces, norms and inner products, weak derivatives, and linear and bilinear forms. All of this theory is then utilised in Section 3 where we introduce the weak formulation of partial differential equations. Backed up with several examples, we will present a key result pertaining to the existence and uniqueness of solutions of weak formulations. With the knowledge attained in these sections, we shall proceed to Section 4 where we shall discuss the finite element space, the establishment of the finite element method itself, and the varying element types that can be used to make up a triangulation of a domain. As we will see, each of these element types are based on a reference element defined on the domain $[-1, 1]^d$, where d is the dimension of the problem. These reference elements and their attributes will each be looked at in turn, with the choice of the basis functions playing a key part in our presentation.

Once we have covered the key theoretical concepts, we shall apply these to our implementation of the finite element method. A full repository of all the work in this package can be found at `https://github.com/sammyjp/FEM_DISS`, although we will be presenting snippets of code in both our written explanations and in Appendix A. We will break down all of the classes that are required in the implementation, with the inclusion of UML class diagrams for each.

The penultimate section sees a discussion of numerical quadrature techniques, which are

invaluable to the assembly of finite element stiffness matrices and load vectors from an implementation respect. The rules for each element type are looked at in depth and are then linked to the evaluation of basis functions and their gradients on the reference elements introduced in Section 4.3.

Finally, we demonstrate all of the techniques, procedures and rules discussed by solving a selection of examples in both one and two dimensions. In the one dimensional example, we complete an a posterior error analysis for a range of basis function polynomial degrees and mesh discretisation sizes. We present the key results from this analysis and determine whether our implementation follows the theory for the convergence of the finite element method. For the two dimensional problem, we employ a hybrid grid made up of both triangles and quadrilaterals to show the extensive capabilities of the finite element method and our implementation of it.

We then conclude by discussing the material covered and exploring potential extensions to the work done in this dissertation.

2 Weak Derivatives and Sobolev Spaces

In this section, we will outline the theory needed to understand the finite element method detailed in Section 4. Although the main focus of this dissertation is on the implementation of the finite element method, it is important to have a theoretical comprehension so that the computational package can be utlised to its full potential. A thorough theoretical background to the finite element method can be found across Brenner and Scott (2007), Ciarlet (2002) and Braess (2001).

2.1 Spaces of Integrable Functions

Choosing appropriate function spaces is very important when we come to formulating partial differential equations for the application of the finite element method. As we will see later in Section 3.1, we require the functions in the chosen space to be integrable. Here, we shall introduce one of the most fundamental spaces of integrable functions, the Lebesgue space.

2.1.1 Norms and Inner Products

We shall start by giving the definition of a norm and inner product on a general space (Adams and Fournier, 2003).

Definition 2.1.1 (Norm)

A norm on a vector space X is a real-valued function f on X satisfying the following conditions:

(i) $f(x) \geq 0$ *for all* $x \in X$ *and* $f(x) = 0$ *if and only if* $x = 0$.

(ii) $f(cx) = |c|\, f(x)$ *for every* $x \in X$ *and* $c \in \mathbb{C}$.

(iii) $f(x + y) \leq f(x) + f(y)$ *for every* $x, y \in X$.

We denote the norm on X *by* $\|\cdot\|_X$.

A function is said to be a semi-norm if only conditions (ii) and (iii) of a norm hold and it is denoted by $|\cdot|_X$.

Definition 2.1.2 (Inner Product)

If X is a vector space, a function $\langle \cdot, \cdot \rangle_X$ defined by $X \times X$ is called an inner product on X provided that for every $x, y, z \in X$ and $a, b \in \mathbb{C}$

(i) $\langle x, y \rangle_X = \langle y, x \rangle_X$

(ii) $\langle ax + by, z \rangle_X = a \langle x, z \rangle_X + b \langle y, z \rangle_X$

(iii) $\langle x, x \rangle_X = 0$ *if and only if* $x = 0$

It is such that the following norm can be induced from the inner product:

$$\|x\|_X := \sqrt{\langle x, x \rangle_X} \tag{2.1}$$

2.1.2 Lebesgue Spaces

A Lebesgue space is a function space of integrable functions. We will find that these function spaces are an imperative component in the weak formulation of partial differential equations, which we will discuss later in Section 3.

Definition 2.1.3 (Lebesgue Space)

Let f be a real-valued Lebesgue measurable (Brenner and Scott, 2007) function on a domain Ω and let $p \in [1, \infty]$. The Lebesgue space L_p is defined as

$$L_p(\Omega) := \{f : \|f\|_{L_p(\Omega)} < \infty\} \tag{2.2}$$

where the norm associated with the Lebesgue space L_p for $p \in [1, \infty)$ is given by

$$\|f\|_{L_p(\Omega)} := \left(\int_\Omega |f(x)|^p \, dx \right)^{\frac{1}{p}} \tag{2.3}$$

and the norm for $p = \infty$ is given by

$$\|f\|_{L_\infty(\Omega)} := \text{ess sup}\{|f(x)| : x \in \Omega\} \tag{2.4}$$

The Lebesgue space we will be working with most in this dissertation is the L_2 space. We can define the inner product for this space as

$$\langle f, g \rangle_{L_2(\Omega)} := \int_\Omega f(x) g(x) dx. \tag{2.5}$$

2.1.3 The Cauchy-Schwarz Inequality

A key result that will be invaluable to us later on is the Cauchy-Schwarz Inequality. Again, as we will be working predominantly with the Lebesgue space in the case of $p = 2$, we give this result for the L_2 space, although it can be applied to all inner product spaces.

Lemma 2.1.4 (Cauchy-Schwarz Inequality)

Let u and v belong to the L_2 Lebesgue space on a domain Ω, then

$$\left|\langle u, v\rangle_{L_2(\Omega)}\right| \leq \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} . \tag{2.6}$$

Proof (as seen in Houston (2016b)).

Let $\lambda \in \mathbb{R}$, then

$$\begin{aligned}
0 &\leq \|u + \lambda v\|^2_{L_2(\Omega)} \\
&= \langle u + \lambda v, u + \lambda v\rangle_{L_2(\Omega)} \\
&= \langle u, u\rangle_{L_2(\Omega)} + \langle u, \lambda v\rangle_{L_2(\Omega)} + \langle \lambda v, u\rangle_{L_2(\Omega)} + \langle \lambda v, \lambda v\rangle_{L_2(\Omega)} \\
&= \|u\|^2_{L_2(\Omega)} + 2\lambda \langle u, v\rangle_{L_2(\Omega)} + \lambda^2 \|v\|^2_{L_2(\Omega)}
\end{aligned}$$

Hence, we have a quadratic polynomial in λ with real coefficients. As it is non-negative for all $\lambda \in \mathbb{R}$, the discriminant is non-positive, i.e.

$$\left|2 \langle u, v\rangle_{L_2(\Omega)}\right|^2 - 4 \|u\|^2_{L_2(\Omega)} \|v\|^2_{L_2(\Omega)} \leq 0$$

and therefore

$$\begin{aligned}
&\left|2 \langle u, v\rangle_{L_2(\Omega)}\right|^2 \leq 4 \|u\|^2_{L_2(\Omega)} \|v\|^2_{L_2(\Omega)} \\
\iff &4 \left|\langle u, v\rangle_{L_2(\Omega)}\right|^2 \leq 4 \|u\|^2_{L_2(\Omega)} \|v\|^2_{L_2(\Omega)} \\
\iff &\left|\langle u, v\rangle_{L_2(\Omega)}\right| \leq \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)}
\end{aligned}$$

□

Equivalently, the Cauchy-Schwarz inequality in L_2 can be stated as

$$\int_\Omega |u(x)|\,|v(x)|\,dx \leq \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} . \tag{2.7}$$

which will be useful later on in Section 3.2.

Another statement of the Cauchy-Schwarz inequality is for summations in $\mathbb{R}^n$. We give this below.

Lemma 2.1.5 (Cauchy-Schwarz Inequality in $\mathbb{R}^n$)

Let $x, y \in \mathbb{R}^n$. Applying the Cauchy-Schwarz inequality to $\mathbb{R}^n$ gives

$$\sum_{i=1}^{n} x_i y_i \leq \left(\sum_{i=1}^{n} x_i^2 \right)^{\frac{1}{2}} \left(\sum_{i=1}^{n} y_i^2 \right)^{\frac{1}{2}} \tag{2.8}$$

2.2 Weak Derivatives

When we discuss derivatives we are more often than not talking about *classical* derivatives. A function f is differentiable at a point x as long as the following limit exists

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.9}$$

If a function is differentiable at a point then it must also be continuous at that point. We say that a function f is continuously differentiable, or of class $\mathcal{C}^1$, if the derivative $f'(x)$ exists and is a continuous function itself. Likewise, if a function is twice differentiable and both the first and second derivatives are continuous it is said to be of class $\mathcal{C}^2$, etc. If a function is infinitely differentiable and continuous it is said to be of class $\mathcal{C}^\infty$.

A *weak* or *variational* derivative is employed for functions that are non-differentiable but are integrable, for example the function $f(x) = |x|$:

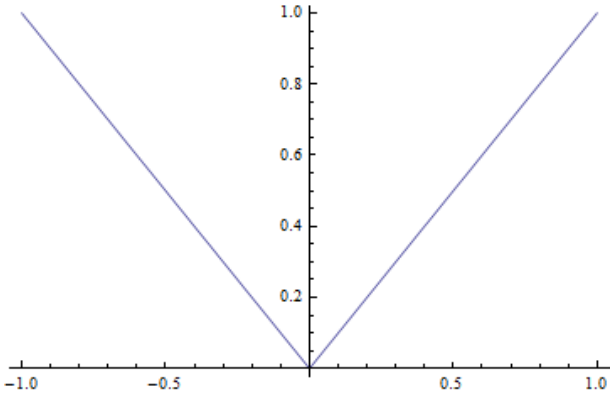


Figure 1: Plot of $f(x) = |x|$.

We can see from the above figure (1) that although the function is continuous, it is non-differentiable at the point $x = 0$.

We now need to introduce the notion of a multi-index (Saint Raymond, 1991).

Definition 2.2.1 (Multi-index)

A multi-index α is an n-tuple of non-negative integers α_i:

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbb{N}^n \tag{2.10}$$

The length of α is given by

$$|\alpha| = \alpha_1 + \alpha_2 + \ldots + \alpha_n \tag{2.11}$$

A multi-index α determines a partial differential operator of order $|\alpha|$. For $\phi \in \mathcal{C}^\infty$,

$$D^\alpha \phi = \left(\frac{\partial}{\partial x_1}\right)^{\alpha_1} \left(\frac{\partial}{\partial x_2}\right)^{\alpha_2} \ldots \left(\frac{\partial}{\partial x_n}\right)^{\alpha_n} \phi = \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}} \phi \tag{2.12}$$

Definition 2.2.2 (Support of a Function)

The support of a real continuous function f on an open set $X \subset \mathbb{R}^n$ is the closure of the set of points in X at which f does not vanish, i.e.

$$supp\ f = \overline{\{x \in X : f(x) \neq 0\}} \tag{2.13}$$

If the support of f is a bounded subset of X then it is said to have *compact support* (Mandelker, 1971). We define the set $\mathcal{C}_0^k$ to be the set of k times differentiable functions that have compact support.

We can now give the formal definition of a weak derivative. The definition we shall use is as seen in Gilbarg and Trudinger (2015).

Definition 2.2.3 (Weak Derivative)

Let u be a locally integrable function on a domain Ω and let α be any multi-index. Then a locally integrable function v is called the α^{th} weak derivative of u if it satisfies

$$\int_\Omega \phi v dx = (-1)^{|\alpha|} \int_\Omega u D^\alpha \phi dx \tag{2.14}$$

for all $\phi \in \mathcal{C}_0^{|\alpha|}(\Omega)$.

Provided v exists, we write $v = D^\alpha u$. A function is said to be k times weakly differentiable if all its weak derivatives exist for orders up to and including k. If u is sufficiently smooth, say $u \in \mathcal{C}^j(\Omega)$, then its weak derivative $D^\alpha u$ of order $|\alpha| \leq j$ coincides with the corresponding partial derivative in the classical pointwise sense (Houston, 2016b).

Example 2.2.4

We shall compute the first ($\alpha = 1$) weak derivative for $f(x) = |x|$ as seen in Figure 1. We shall start with the right hand side of Equation 2.14. For any $\phi \in \mathcal{C}_0^1$,

$$\begin{aligned}
(-1)^1 \int_{-\infty}^{\infty} f(x) D^1 \phi(x) dx &= -\int_{-\infty}^{\infty} |x|\, \phi'(x) dx \\
&= -\int_{-\infty}^{0} -x\phi'(x) dx - \int_{0}^{\infty} x\phi'(x) dx \\
&= \int_{-\infty}^{0} x\phi'(x) dx + \int_{0}^{\infty} -x\phi'(x) dx \\
&= (x \cdot \phi(x))|_{x=-\infty}^{0} - \int_{-\infty}^{0} 1 \cdot \phi(x) dx + (-x \cdot \phi(x))|_{x=0}^{\infty} \\
&\quad - \int_{0}^{\infty} (-1) \cdot \phi(x) dx \\
&= \int_{-\infty}^{0} (-1) \cdot \phi(x) dx + \int_{0}^{\infty} 1 \cdot \phi(x) dx \qquad (\text{as } \phi \in \mathcal{C}_0^1)
\end{aligned}$$

which is now equivalent to the left hand side of Equation 2.14. Hence, we have that the first weak derivative of $f(x)$ is given by

$$v(x) = D^1 f(x) = f'(x) \begin{cases} -1, & x < 0 \\ \\ 1, & x > 0 \end{cases}.$$

□

2.3 Sobolev Spaces

Here, we extend the work seen in Sections 2.1 and 2.2 to introduce Sobolev spaces. These are very highly utilised spaces, as we will see, that are built upon the Lebesgue norm.

Consider an open set $\Omega \subseteq \mathbb{R}^n$, fix $p \in [1, \infty]$ and let m be a non-negative integer (Bressan, 2012).

Definition 2.3.1 (Sobolev Space)

The Sobolev space $W^{m,p}(\Omega)$ is the space of all locally integrable functions $u : \Omega \mapsto \mathbb{R}$ such that, for every multi-index α with $|\alpha| \leq m$, the weak derivative $D^\alpha u$ exists and belongs to the space $L_p(\Omega)$.

We define the subspace $W_0^{m,p}(\Omega) \subseteq W^{m,p}(\Omega)$ to be the closure of $\mathcal{C}_0^\infty$ in $W^{m,p}(\Omega)$.

2.3.1 Sobolev Norms

A Sobolev space $W^{m,p}(\Omega)$ is equipped with its own specific norm. (Brenner and Scott, 2007)

Definition 2.3.2 (Sobolev Norm)

Let $u \in L_1(\Omega)$. The Sobolev norm is defined by

$$\|u\|_{W^{m,p}(\Omega)} := \left(\sum_{|\alpha| \leq m} \|D^\alpha u\|^p_{L_p(\Omega)} \right)^{\frac{1}{p}} \tag{2.15}$$

and similarly the Sobolev infinity norm is defined by

$$\|u\|_{W^{m,\infty}(\Omega)} := \max_{|\alpha| \leq m} \|D^\alpha u\|_{L_\infty(\Omega)} \tag{2.16}$$

The Sobolev semi-norm is given by:

$$|u|_{W^{m,p}(\Omega)} := \left(\sum_{|\alpha| = m} \|D^\alpha u\|^p_{L_p(\Omega)} \right)^{\frac{1}{p}} \tag{2.17}$$

It can be seen that the $W^{0,p}(\Omega)$ space is equal to the $L_p(\Omega)$ space.

2.3.2 Hilbert Spaces

For the majority of this dissertation we shall be using the Sobolev space in the case where $p = 2$, also known as a Hilbert space (Bressan, 2012). We denote this as $H^m(\Omega)$ and it has norm

$$\|u\|_{H^m(\Omega)} := \|u\|_{W^{m,2}(\Omega)} := \left(\sum_{|\alpha| \leq m} \|D^\alpha u\|^2_{L_2(\Omega)} \right)^{\frac{1}{2}} = \left(\sum_{|\alpha| \leq m} \int_\Omega |D^\alpha u|^2_{L_2(\Omega)} \, dx \right)^{\frac{1}{2}} \tag{2.18}$$

and likewise it has semi norm $|u|_{H^m(\Omega)} := |u|_{W^{m,2}(\Omega)}$.

The Hilbert space $H^m(\Omega)$ is equipped with inner product

$$\langle u, v \rangle_{H^m(\Omega)} := \sum_{|\alpha| \leq m} \int_\Omega D^\alpha u D^\alpha v dx. \tag{2.19}$$

Similarly, we define the space $H^m_0(\Omega) := W^{m,2}_0(\Omega)$.

2.3.3 Linear and Bilinear Forms

In order to prove the existence and uniqueness of solutions to finite element problems in Section 3.2, we are going to require the definitions of both linear and bilinear forms. Along with this, an overview of some of the properties that they can possess is also necessary.

Definition 2.3.3 (Linear form)

A linear form (or functional) $y(\cdot)$ on a vector space V is a scalar valued function with the property

$$y(\alpha_1 x_1 + \alpha_2 x_2) = \alpha_1 y(x_1) + \alpha_2 y(x_2) \tag{2.20}$$

for all $x_1, x_2 \in V$ and $\alpha_1, \alpha_2 \in \mathbb{R}$ (Halmos, 1974).

Definition 2.3.4 (Bilinear form)

A scalar valued function $w(\cdot,\cdot)$ on a space V is said to be a bilinear form (or functional) on V if

$$w(\alpha_1 x_1 + \alpha_2 x_2, v) = \alpha_1 w(x_1, v) + \alpha_2 w(x_2, v) \tag{2.21}$$

and

$$w(x, \alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 w(x, v_1) + \alpha_2 w(x, v_2) \tag{2.22}$$

for all $x, v, x_1, x_2, v_1, v_2 \in V$ and $\alpha_1, \alpha_2 \in \mathbb{R}$ (Halmos, 1974).

One of the main properties we will require, which both linear and bilinear forms can hold, is *continuity*. Additionally, bilinear forms may also be *coercive*.

Suppose that H is a real Hilbert space equipped with norm $\|\cdot\|_H$.

(1) A linear form $y(\cdot)$ is said to be *continuous* if there exists a constant $c_0 > 0$ such that

$$|y(x)| \leq c_0 \|x\|_H \tag{2.23}$$

for all $x \in H$.

(2) A bilinear form $w(\cdot,\cdot)$ is said to be *continuous* if there exists a constant $c_1 > 0$ such that

$$|w(x,y)| \leq c_1 \|x\|_H \|y\|_H \tag{2.24}$$

for all $x, y \in H$.

(3) A bilinear form $w(\cdot,\cdot)$ is said to be *coercive* or *V-elliptic* if there exists a constant, $c_2 > 0$, such that

$$w(x,x) \geq c_2 \|x\|_H^2 \tag{2.25}$$

for all $x \in H$.

2.3.4 The Poincaré-Friedrichs Inequality

In order to aid in the manipulation of inequalities when proving continuity (2.24, 2.23) and coercivity (2.25) of linear and bilinear forms, we shall introduce the following lemma (as stated in Brezis (2010)), namely the Poincaré-Friedrichs inequality.

Lemma 2.3.5 (Poincaré-Friedrichs Inequality)

Suppose that $1 \leq p < \infty$ *and* Ω *is a bounded open set. Then there exists a constant* C *(depending on* Ω *and* p*) such that*

$$\|u\|_{L_p(\Omega)} \leq C \|\nabla u\|_{L_p(\Omega)} \quad \forall u \in W_0^{1,p}(\Omega). \tag{2.26}$$

As we are primarily working with the L_2 space throughout this dissertation, we shall only give the proof for the case when $p = 2$ as seen in Houston (2016a). For more on Poincaré and Friedrichs type inequalities see Zheng and Qi (2005).

Proof. Let us consider the special case of a rectangular domain $\Omega = (a,b) \times (c,d)$ in $\mathbb{R}^2$. The proof for general Ω is analogous. For our function $u(x,y)$, it is evident that

$$u(x,y) = u(a,y) + \int_a^x \frac{\partial u}{\partial x}(\xi,y)d\xi = \int_a^x \frac{\partial u}{\partial x}(\xi,y)d\xi, \quad c < y < d.$$

(as $u = 0$ on the boundary)

Hence we have

$$\begin{aligned}
\int_\Omega |u(x,y)|^2 \, dxdy &= \int_a^b \int_c^d \left| \int_a^x \frac{\partial u}{\partial x}(\xi, y) d\xi \right|^2 dydx \\
&\leq \int_a^b \int_c^d \left| \|1\|_{L_2(a,x)} \left\| \frac{\partial u}{\partial x}(\xi, y) \right\|_{L_2(a,x)} \right|^2 dydx \\
&\qquad\qquad \text{(by Cauchy-Schwarz inequality (2.7))} \\
&= \int_a^b \int_c^d \left(\int_a^x 1 d\xi \right) \left(\int_a^x \left| \frac{\partial u}{\partial x}(\xi, y) \right|^2 d\xi \right) dydx \\
&\leq \int_a^b (x-a) dx \left(\int_c^d \int_a^b \left| \frac{\partial u}{\partial x}(\xi, y) \right|^2 d\xi dy \right) \qquad \text{(as } x \leq b\text{)} \\
&= \frac{1}{2}(b-a)^2 \int_\Omega \left| \frac{\partial u}{\partial x}(x,y) \right|^2 dxdy. \\
&\qquad\qquad \text{(as } \xi \text{ is an arbitrary variable of integration)}
\end{aligned}$$

Analogously, we have the result

$$\int_\Omega |u(x,y)|^2 \, dxdy \leq \frac{1}{2}(d-c)^2 \int_\Omega \left| \frac{\partial u}{\partial y}(x,y) \right|^2 dxdy.$$

If we add together these two inequalities, we achieve the result

$$\int_\Omega |u(x,y)|^2 \, dxdy \leq \left(\frac{(b-a)^2 + (d-c)^2}{2} \right) \int_\Omega \left(\left| \frac{\partial u}{\partial x} \right|^2 + \left| \frac{\partial u}{\partial y} \right|^2 \right) dxdy.$$

Taking square roots of both sides we get the inequality in the form of

$$\|u\|_{L_2(\Omega)} \leq C \, \|\nabla u\|_{L_2(\Omega)} \,,$$

where

$$C = \left(\frac{(b-a)^2 + (d-c)^2}{2} \right)^{\frac{1}{2}}.$$

□

3 Weak Formulation of Elliptic Boundary Value Problems

In Section 2.2, we discussed the notion of weak derivatives for functions that may not be classically differentiable. This notion can be applied to the formulation of partial differential equations (PDEs) in order to determine the weak solutions to problems for which classical solutions may not exist. We shall begin this section by discussing how to construct the weak formulation of elliptic boundary value problems. Following on from this we will introduce a key result, the Lax-Milgram theorem, in order to prove the existence and uniqueness of solutions to weakly formulated partial differential equations.

3.1 Weak Formulation of a PDE

The principle behind the weak formulation of a PDE is similar to that of a weak derivative. We shall proceed first by outlining the steps required to obtain the weak formulation of a PDE and then to demonstrate the process we shall work through an example problem with homogeneous Dirichlet boundary conditions.

The steps to obtain the weak formulation are as follows:

Step (1): Multiply PDE by a smooth test function v and integrate both sides over the domain.

Step (2): Apply integration by parts to the left hand side if necessary.

Step (3): Select test function such that it belongs to an appropriate space, depending on boundary conditions.

Step (4): State weak formulation of PDE.

In order to commence with the example problem we must give Green's first identity (Drábek and Holubová, 2014) so we can integrate our equation by parts.

Lemma 3.1.1 (Green's First Identity)

Let $\Omega \subset \mathbb{R}^n$ be a bounded open subset with a $\mathcal{C}^1$ boundary $\partial\Omega$. Let $\boldsymbol{\nu}$ denote the unit outward normal vector to the boundary $\partial\Omega$. Then

$$\int_\Omega \Delta u v dx = \int_{\partial\Omega} \frac{\partial u}{\partial \boldsymbol{\nu}} v ds - \int_\Omega \nabla u \cdot \nabla v dx, \tag{3.1}$$

where $\dfrac{\partial u}{\partial \boldsymbol{\nu}} = \boldsymbol{\nu} \cdot \nabla u$ *denotes the derivative with respect to the unit outward normal to* $\partial\Omega$.

Example 3.1.2 (Homogeneous Dirichlet Boundary Condition)

Consider the problem

$$-\Delta u = f, \qquad \text{in } \Omega, \tag{3.2}$$

$$u = 0, \qquad \text{on } \partial\Omega. \tag{3.3}$$

The first step of obtaining the weak formulation of the PDE is to multiply our equation (3.2) by a smooth test function v and integrate over the domain like so

$$-\int_\Omega (\Delta u) v dx = \int_\Omega f v dx. \tag{3.4}$$

Following this, we now need to evaluate the left hand side by parts using Green's first identity (Lemma 3.1.1), giving us

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_{\partial\Omega} \frac{\partial u}{\partial \boldsymbol{\nu}} v ds = \int_\Omega f v dx. \tag{3.5}$$

In order to get rid of the boundary integral we must choose our test function v such that it is equal to zero on the boundary, or in other words, $v \in H_0^1(\Omega)$. This allows us to write

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx. \tag{3.6}$$

We can now state the weak formulation of our PDE problem:

Find u such that $u|_{\partial\Omega} = 0$ and

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx. \tag{3.7}$$

for all v such that $v|_{\partial\Omega} = 0$. □

Hence we have turned our PDE from its classical formulation into its weak formulation, allowing us to proceed to proving the existence and uniqueness of solutions in the following subsection (3.2).

Note: It can be shown using Definitions 2.3.3 and 2.3.4 that the left hand side of (3.7) is a bilinear form on $H_0^1(\Omega)$ and the right hand side is a linear form on $H_0^1(\Omega)$. Therefore,

we can define

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v dx, \tag{3.8}$$

$$\ell(v) = \int_\Omega f v dx, \tag{3.9}$$

allowing us to rewrite our weak formulation as

Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = \ell(v) \tag{3.10}$$

for all $v \in H_0^1(\Omega)$.

3.2 The Lax-Milgram Theorem

It is very important to be able to show the existence and uniqueness of solutions to equations before attempting to solve them. The following result allows us to do this, provided our equation is made up of a bilinear and a linear form (Section 2.3.3).

Theorem 3.2.1 (The Lax-Milgram Theorem)

Let H be a real Hilbert space with inner product $\langle \cdot, \cdot \rangle_H$ and norm $\|\cdot\|_H$. Let $a(\cdot,\cdot)$ be a bilinear form on H and let $\ell(\cdot)$ be a linear functional on H such that:

(a). $a(\cdot,\cdot)$ is coercive,

(b). $a(\cdot,\cdot)$ is continuous,

(c). $\ell(\cdot)$ is continuous.

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

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

for all $v \in H$.

To demonstrate the Lax-Milgram theorem, we shall continue on with the Example 3.1.2 from Section 3.1.

Example 3.2.2 (Homogeneous Dirichlet Boundary Condition cont.)

We have already obtained the weak formulation to our PDE, given by

Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx \tag{3.11}$$

for all $v \in H_0^1(\Omega)$.

where the left hand side is a bilinear form denoted by $a(u, v)$ and the right hand side is a linear form denoted by $\ell(v)$.

We shall begin by proving condition (c) of the Lax-Milgram theorem (3.2.1), which is the continuity (2.23) of our linear form $\ell(v)$.

$$\begin{aligned}
|\ell(v)| &= \left| \int_\Omega f v dx \right| \\
&\leq \|f\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} && \text{(by Cauchy-Schwarz inequality (2.7))} \\
&\leq \|f\|_{L_2(\Omega)} \left(\|v\|^2_{L_2(\Omega)} + \|\nabla v\|^2_{L_2(\Omega)} \right)^{\frac{1}{2}} \\
&= \|f\|_{L_2(\Omega)} \|v\|_{H^1(\Omega)} \\
&= c_0 \|v\|_{H^1(\Omega)} && \text{(where } c_0 = \|f\|_{L_2(\Omega)}\text{)}
\end{aligned}$$

Hence, $\ell(\cdot)$ is continuous on $H^1(\Omega)$.

Next, we shall prove condition (b), that our bilinear form $a(u, v)$ is continuous (2.24).

$$\begin{aligned}
|a(u, v)| &= \left| \int_\Omega \nabla u \cdot \nabla v dx \right| \\
&\leq \|\nabla u\|_{L_2(\Omega)} \|\nabla v\|_{L_2(\Omega)} && \text{(by Cauchy-Schwarz inequality (2.7))} \\
&\leq \left(\|u\|^2_{L_2(\Omega)} + \|\nabla u\|^2_{L_2(\Omega)} \right)^{\frac{1}{2}} \left(\|v\|^2_{L_2(\Omega)} + \|\nabla v\|^2_{L_2(\Omega)} \right)^{\frac{1}{2}} \\
&= c_1 \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)} && \text{(where } c_1 = 1\text{)}
\end{aligned}$$

Hence, $a(\cdot, \cdot)$ is continuous on $H^1(\Omega)$.

Finally, we need to prove condition (a) in order to show that our bilinear form is coercive (2.25).

$$a(u, u) = \int_\Omega |\nabla u|^2 \, dx = \|\nabla u\|^2_{L_2(\Omega)} \tag{3.12}$$

By employing the Poincaré-Friedrichs inequality (Lemma 2.3.5), we also have that

$$Ca(u, u) = C \|\nabla u\|^2_{L_2(\Omega)} \geq \|u\|^2_{L_2(\Omega)} \tag{3.13}$$

If we add together (3.12) and (3.13) we achieve the result

$$\begin{aligned}(1+C)a(u,u) &\geq \|u\|^2_{L_2(\Omega)} + \|\nabla u\|^2_{L_2(\Omega)} \\ &= \|u\|^2_{H^1(\Omega)} \\ \iff a(u,u) &\geq \left(\frac{1}{1+C}\right)\|u\|^2_{H^1(\Omega)} \\ &= c_2\|u\|^2_{H^1(\Omega)} \qquad \text{(where } c_2 = \tfrac{1}{1+C}\text{)}\end{aligned}$$

Hence, our bilinear form is coercive on $H^1(\Omega)$ and therefore by the Lax-Milgram theorem (3.2.1), there exists a unique solution $u \in H^1_0(\Omega) \subseteq H^1(\Omega)$ such that

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

for all $v \in H^1_0(\Omega)$. □

3.3 General Elliptic PDE Example

We will now work through an example of a general elliptic PDE (Houston, 2016a) in $\Omega \subset \mathbb{R}^n$ of the form

$$-\varepsilon\Delta u + \boldsymbol{b}\cdot\nabla u + c(x)u = f(x), \qquad x \in \Omega \tag{3.14}$$

$$u = 0, \qquad x \in \partial\Omega, \tag{3.15}$$

where $\partial\Omega$ denotes the boundary of Ω, ε is a positive constant, $\boldsymbol{b}$ is a vector of functions given by $b_i(x)$ for $i = 1, 2, \ldots, n$ and $b_i(x)$, $c(x)$ and $f(x)$ are all $\mathcal{C}^1$ functions on the closure of Ω (for $i = 1, 2, \ldots, n$).

Firstly, we must multiply our equation by a smooth test function v and then integrate over the domain, giving

$$-\int_\Omega \varepsilon\Delta u v dx + \int_\Omega \boldsymbol{b}\cdot\nabla u v dx + \int_\Omega cuv dx = \int_\Omega fv dx. \tag{3.16}$$

Secondly, we shall integrate the left hand side by parts, using Green's first identity (Lemma 3.1.1), to give

$$\int_\Omega \varepsilon\nabla u\cdot\nabla v dx - \int_{\partial\Omega}\varepsilon\frac{\partial u}{\partial\boldsymbol{\nu}}v ds + \int_\Omega \boldsymbol{b}\cdot\nabla u v dx + \int_\Omega cuv dx = \int_\Omega fv dx. \tag{3.17}$$

Next, we select our test function v such that it is equal to zero on the boundary ($v \in H_0^1(\Omega)$), allowing us to get rid of the boundary integral.

$$\int_\Omega \varepsilon \nabla u \cdot \nabla v dx + \int_\Omega \boldsymbol{b} \cdot \nabla u v dx + \int_\Omega c u v dx = \int_\Omega f v dx. \tag{3.18}$$

Hence, the weak formulation of our PDE (3.14) is as follows:

Find u such that $u|_{\partial\Omega} = 0$ and

$$\int_\Omega \varepsilon \nabla u \cdot \nabla v dx + \int_\Omega \boldsymbol{b} \cdot \nabla u v dx + \int_\Omega c u v dx = \int_\Omega f v dx. \tag{3.19}$$

for all v such that $v|_{\partial\Omega} = 0$.

Our left hand side is a bilinear form (Definition 2.3.4) and our right hand side is a linear form (Definition 2.3.3) so we can define

$$a(u, v) = \int_\Omega \varepsilon \nabla u \cdot \nabla v dx + \int_\Omega \boldsymbol{b} \cdot \nabla u v dx + \int_\Omega c u v dx \tag{3.20}$$

$$\ell(v) = \int_\Omega f v dx. \tag{3.21}$$

Therefore, we can rewrite our weak formulation as

Find $u \in H_0^1(\Omega)$ such that

$$a(u, v) = \ell(v) \tag{3.22}$$

for all $v \in H_0^1(\Omega)$.

Now that we have our weak formulation, we may proceed with proving the existence and uniqueness of a solution to the problem using the Lax-Milgram theorem (3.2.1).
We know from Example 3.2.2 that our linear form $\ell(v)$ is continuous (2.23) on $H^1(\Omega)$, as we showed that

$$|\ell(v)| \leq \|f\|_{L_2(\Omega)} \|v\|_{H^1(\Omega)} = c_0 \|v\|_{H^1(\Omega)}.$$

Next, we must show continuity of our bilinear form $a(u, v)$ by using (2.24).

$$\begin{aligned}
|a(u, v)| &= \left| \int_\Omega \varepsilon \nabla u \cdot \nabla v dx + \int_\Omega \boldsymbol{b} \cdot \nabla u v dx + \int_\Omega c u v dx \right| \\
&\leq \left| \varepsilon \int_\Omega \nabla u \cdot \nabla v dx \right| + \left| \int_\Omega \boldsymbol{b} \cdot \nabla u v dx \right| + \left| \int_\Omega c u v dx \right| \\
&\leq \varepsilon \sum_{i=1}^n \int_\Omega \left| \frac{\partial u}{\partial x_i} \right| \left| \frac{\partial v}{\partial x_i} \right| dx + \sum_{i=1}^n \int_\Omega |b_i(x)| \left| \frac{\partial u}{\partial x_i} \right| |v| \, dx + \int_\Omega |c(x)| \, |u| \, |v| \, dx.
\end{aligned}$$

Let us define

$$C_b = \max_{1\le i\le n} \max_{x\in\bar{\Omega}} |b_i(x)| \quad \text{and} \quad C_c = \max_{x\in\bar{\Omega}} |c(x)| .$$

Then we can write

$$\begin{aligned}
|a(u,v)| &\le \varepsilon \sum_{i=1}^{n} \int_\Omega \left|\frac{\partial u}{\partial x_i}\right| \left|\frac{\partial v}{\partial x_i}\right| dx + C_b \sum_{i=1}^{n} \int_\Omega \left|\frac{\partial u}{\partial x_i}\right| |v|\, dx + C_c \int_\Omega |u|\,|v|\, dx \\
&\le \varepsilon \sum_{i=1}^{n} \left\|\frac{\partial u}{\partial x_i}\right\|_{L_2(\Omega)} \left\|\frac{\partial v}{\partial x_i}\right\|_{L_2(\Omega)} + C_b \sum_{i=1}^{n} \left\|\frac{\partial u}{\partial x_i}\right\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} + C_c \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} \\
&\qquad\qquad \text{(by Cauchy-Schwarz inequality (2.7))} \\
&\le \varepsilon \left(\sum_{i=1}^{n} \left\|\frac{\partial u}{\partial x_i}\right\|^2_{L_2(\Omega)}\right)^{\frac{1}{2}} \left(\sum_{i=1}^{n} \left\|\frac{\partial v}{\partial x_i}\right\|^2_{L_2(\Omega)}\right)^{\frac{1}{2}} \\
&\quad + C_b \left(\sum_{i=1}^{n} 1\right)^{\frac{1}{2}} \left(\sum_{i=1}^{n} \left\|\frac{\partial u}{\partial x_i}\right\|^2_{L_2(\Omega)}\right)^{\frac{1}{2}} \|v\|_{L_2(\Omega)} + C_c \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} \\
&\qquad\qquad \text{(by Cauchy-Schwarz inequality in } \mathbb{R}^n \text{ (2.8))} \\
&= \varepsilon \|\nabla u\|_{L_2(\Omega)} \|\nabla v\|_{L_2(\Omega)} + C_b\sqrt{n} \|\nabla u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} + C_c \|u\|_{L_2(\Omega)} \|v\|_{L_2(\Omega)} \\
&\le \left(\varepsilon + C_b\sqrt{n} + C_c\right) \left(\|u\|^2_{L_2(\Omega)} + \|\nabla u\|^2_{L_2(\Omega)}\right)^{\frac{1}{2}} \left(\|v\|^2_{L_2(\Omega)} + \|\nabla v\|^2_{L_2(\Omega)}\right)^{\frac{1}{2}} \\
&= \left(\varepsilon + C_b\sqrt{n} + C_c\right) \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)} \\
&= c_1 \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)} .
\end{aligned}$$

Therefore, our bilinear form $a(u,v)$ is continuous on $H^1(\Omega)$.

Finally, we shall prove coercivity of our bilinear form. We start with

$$a(u,u) = \int_\Omega \varepsilon |\nabla u|^2 dx + \int_\Omega \boldsymbol{b}\cdot\nabla u u dx + \int_\Omega c |u|^2 dx. \tag{3.23}$$

Now, we must consider two cases for $\boldsymbol{b}$:

Case 1: $\boldsymbol{b} \equiv 0$.

$$a(u,u) = \int_\Omega \varepsilon |\nabla u|^2 dx + \int_\Omega c |u|^2 dx$$

Let us make the assumption that $c(x) \ge 0$ for all $x \in \bar{\Omega}$. Then,

$$a(u,u) \ge \int_\Omega \varepsilon |\nabla u|^2 dx = \varepsilon \|\nabla u\|^2_{L_2(\Omega)} . \tag{3.24}$$

Now, by utilising the Poincaré-Friedrichs inequality (Lemma 2.3.5), we also get the result

$$Ca(u,u) \geq C\varepsilon \left\|\nabla u\right\|^2_{L_2(\Omega)} \geq \varepsilon \left\|u\right\|^2_{L_2(\Omega)}. \tag{3.25}$$

Therefore, adding together (3.24) and (3.25) gives

$$\begin{aligned}(1+C)a(u,u) &\geq \varepsilon\left(\left\|\nabla u\right\|^2_{L_2(\Omega)} + \left\|u\right\|^2_{L_2(\Omega)}\right)\\ &= \varepsilon \left\|u\right\|^2_{H^1(\Omega)}\\ \iff a(u,u) &\geq \frac{\varepsilon}{1+C}\left\|u\right\|^2_{H^1(\Omega)}\\ &= c_2 \left\|u\right\|^2_{H^1(\Omega)}.\end{aligned}$$

Case 2: $\boldsymbol{b} \not\equiv 0$

Manipulating our $\boldsymbol{b}$ integral, we can write

$$\begin{aligned}\int_\Omega \boldsymbol{b}\cdot\nabla u u dx &= \frac{1}{2}\int_\Omega \boldsymbol{b}\cdot\nabla(u^2)dx && \text{(by the chain rule)}\\ &= \frac{1}{2}\int_{\partial\Omega}\boldsymbol{b}\cdot\frac{\partial}{\partial\boldsymbol{\nu}}(u^2)ds - \frac{1}{2}\int_\Omega(\nabla\cdot\boldsymbol{b})\left|u\right|^2 dx\\ &&& \text{(by Green's first identity (Lemma 3.1.1))}\\ &= -\frac{1}{2}\int_\Omega(\nabla\cdot\boldsymbol{b})\left|u\right|^2 dx && (\text{as } u|_{\partial\Omega}=0)\end{aligned}$$

Therefore, substituting back into (3.23), we get

$$\begin{aligned}a(u,u) &= \int_\Omega \varepsilon\left|\nabla u\right|^2 dx - \frac{1}{2}\int_\Omega(\nabla\cdot\boldsymbol{b})\left|u\right|^2 dx + \int_\Omega c\left|u\right|^2 dx\\ &= \int_\Omega \varepsilon\left|\nabla u\right|^2 dx + \int_\Omega\left(c - \frac{1}{2}\nabla\cdot\boldsymbol{b}\right)\left|u\right|^2 dx.\end{aligned}$$

We must make an assumption on the positivity of the second integral in order to prove coercivity:

$$\left(c - \frac{1}{2}\nabla\cdot\boldsymbol{b}\right) \geq 0 \quad \forall x \in \bar{\Omega}. \tag{3.26}$$

Hence, we are left with

$$a(u,u) \geq \int_\Omega \varepsilon\left|\nabla u\right|^2 dx = \varepsilon\left\|\nabla u\right\|^2_{L_2(\Omega)}.$$

Applying the Poincaré-Friedrichs inequality (Lemma 2.3.5) exactly as in case 1, we achieve the result

$$\begin{aligned} a(u,u) &\geq \frac{\varepsilon}{1+C} \|u\|^2_{H^1(\Omega)} \\ &= c_2 \|u\|^2_{H^1(\Omega)} . \end{aligned}$$

Therefore, our bilinear form $a(u,v)$ is coercive on $H^1(\Omega)$ and thus by the Lax-Milgram theorem (3.2.1), there exists a unique solution $u \in H^1_0(\Omega) \subseteq H^1(\Omega)$ such that

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

for all $v \in H^1_0(\Omega)$. □

4 Construction of the Finite Element Method

From the previous sections, we now have an understanding of the theoretical principles needed to proceed. This section will introduce the notion of the finite element space, before discussing the actual finite element method itself. We will then explore a selection of element types in both one and two dimensions, and the varying choices of basis functions that each element offers.

4.1 The Finite Element Space

Now that we know how to achieve the weak formulation of a PDE,

$$a(u, v) = \ell(v) \quad \forall v \in V, \tag{4.1}$$

we need to develop some way of approximating the solution. To do this, we must construct finite-dimensional subspaces V_h of V and formulate similar problems on these subspaces. This is known as the Galerkin method. With any $V_h \subset V$, we then have the *discrete problem*:

Find $u_h \in V_h$ such that

$$a(u_h, v_h) = \ell(v_h) \tag{4.2}$$

for all $v_h \in V_h$.

Through the Lax-Milgram theorem, we know that as $V_h \subset V$, a unique solution u_h exists. We term this the *discrete solution.*

4.1.1 The Finite Element

The definition we shall give for a finite element is that which can be found in Ciarlet (2002).

Definition 4.1.1 (Finite Element)

A finite element in $\mathbb{R}^n$ is a triple $\kappa = (K, P, \Sigma)$ where

(i) K is a bounded, closed subset of $\mathbb{R}^n$ with nonempty interior and piecewise-smooth boundary,

(ii) P is a space of polynomials defined over the set K,

(iii) $\Sigma = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_k\}$ *is a set of linearly independent linear forms defined over the space* P.

Within P, there exists a set of functions ϕ_i, $1 \leq i \leq k$ which satisfy

$$\Sigma_j(\phi_i) = \delta_{ij}, \tag{4.3}$$

for $1 \leq j \leq k$. Here, δ_{ij} represents the standard Kronecker delta:

$$\delta_{ij} = \begin{cases} 1, & \text{for } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{4.4}$$

The linear forms in Σ are known as the *degrees of freedom* (DOFs) of the finite element and the functions ϕ_i, $1 \leq i \leq k$ are known as the basis functions of the finite element.

4.1.2 The Finite Element Mesh

To utilise the finite element method on a domain Ω, we must first partition up the domain into a finite element *triangulation* or *mesh*. We denote this mesh by τ_h and it is made up of a finite number of subsets K in such a way that:

(i) $\bar{\Omega} = \cup_{K \in \tau_h} K$,

(ii) For each distinct $K_1, K_2 \in \tau_h$, it must be that the intersection of the interiors of each subset is equal to the empty set $\emptyset$.

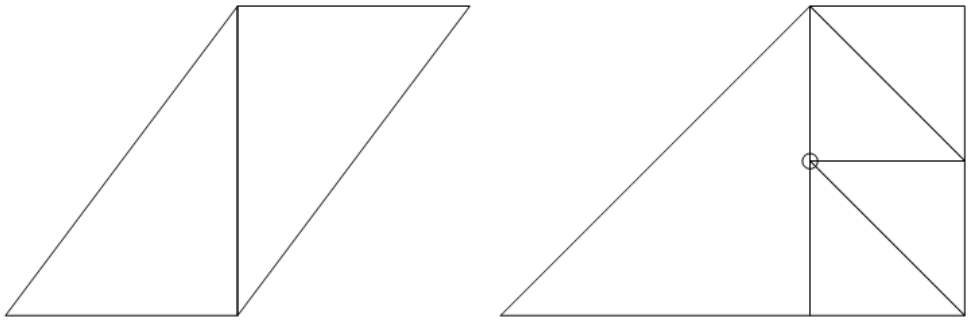
These are the elements on the mesh. The elements must be "arranged" in such a way that satisfies the admissibility property (Braess, 2001).

Definition 4.1.2 (Admissibility of Elements in the Mesh)
A triangulation is said to be admissible if the following properties hold:

(i) If $K_i \cap K_j$ *consists of exactly one point, then it is a shared vertex of* K_i *and* K_j.

(ii) If for $i \neq j$, $K_i \cap K_j$ *consists of more than one point, then* $K_i \cap K_j$ *is a shared edge of* K_i *and* K_j.

We demonstrate this in Figure 2 below.



(a) Admissible (b) Inadmissible

Figure 2: Admissibility of Elements.

The triangulation on the left is admissible but the triangulation on the right is inadmissible due to the hanging node marked by ∘.

Every element in the mesh has a set of vertices associated with it. In order to tell us which vertices make up a specific element, each element has its own connectivity array. This array contains the vertex numbers of the element, making sure they are listed in an anticlockwise order.

Example 4.1.3

In Figure 3 below, we see an example mesh made up of four triangles.

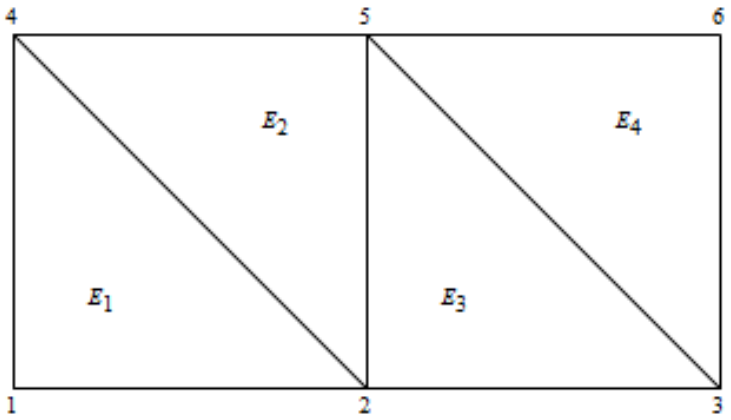


Figure 3: Example of connectivity of elements.

The connectivity arrays for each element are given by

$$\begin{aligned} E_1 &= [1 \;\; 2 \;\; 4] \\ E_2 &= [2 \;\; 5 \;\; 4] \\ E_3 &= [2 \;\; 3 \;\; 5] \\ E_4 &= [3 \;\; 6 \;\; 5]. \end{aligned} \tag{4.5}$$

□

4.1.3 The Finite Element Space

As explained at the beginning of this section, we wish to construct finite dimensional subspaces of our weak formulation space V. Given a triangulation τ_h, made up of non-overlapping elements K, the finite element space $V_h \subset V$ is defined as

$$V_h = \{v \in V : \quad v|_K \in P(K), \quad \forall K \in \tau_h\}, \tag{4.6}$$

where $P(K)$ is the polynomial space associated with each K. Let $N(h)$ be the total number of basis functions ϕ (4.3) on the domain. It is such that the basis functions ϕ_i, $i = 1, \ldots, N(h)$, across all elements, form a basis of the finite element space V_h. As the notation may suggest, h represents the "size" of the discretisation of the domain. The finite element solution theoretically should converge to the true solution as h tends to zero.

4.2 The Finite Element Method

Now that we have defined the finite element space, our goal is to find a solution $u_h \in V_h \subset V$ to our discrete problem (4.2). As discussed above in Section 4.1.3, the basis functions ϕ_i, $i = 1, \ldots, N(h)$ form a basis of V_h. This can be written as $V_h = \text{span}\{\phi_1, \phi_2, \ldots, \phi_{N(h)}\}$. Hence, we can define our finite element solution u_h to be

$$u_h(x) = \sum_{j=1}^{N(h)} U_j \phi_j(x), \tag{4.7}$$

where U_j, $j = 1, \ldots, N(h)$, are so far unknown. In turn, this can then be applied to our discrete problem (4.2) to give the system of equations

$$\sum_{j=1}^{N(h)} a(\phi_j, \phi_i) U_j = \ell(\phi_i), \quad i = 1, \ldots, N(h). \tag{4.8}$$

This can be written in matrix-vector form as

$$AU = F. \tag{4.9}$$

The A matrix is commonly named the finite element *stiffness* matrix and satisfies the property that it is always invertible if the bilinear form a is V-elliptic (coercive) (Ciarlet, 2002). We call F the *load* vector. The size of our system is determined by the number of degrees of freedom in our problem.

Example 4.2.1

Going back to the Poisson equation in Example 3.1.2, we have

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v dx, \tag{4.10}$$

$$\ell(v) = \int_\Omega f v dx, \tag{4.11}$$

where $u, v \in H_0^1(\Omega)$. Our finite element formulation of this problem is given by

Find $u_h \in V_h$ such that

$$a(u_h, v_h) = \ell(v_h) \tag{4.12}$$

for all $v_h \in V_h$, where $V_h \in H_0^1(\Omega)$ is the finite element space for this problem.

Setting up our matrix system, we see that each entry in A is given by

$$A_{ij} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i dx, \tag{4.13}$$

and each entry in F is given by

$$F_i = \int_\Omega f \phi_i dx. \tag{4.14}$$

Applying this to the mesh τ_h of elements K, we can write these integrals as

$$A_{ij} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i dx = \sum_K \int_K \nabla \phi_j \cdot \nabla \phi_i dx, \tag{4.15}$$

$$F_i = \int_\Omega f \phi_i dx = \sum_K \int_K f \phi_i dx. \tag{4.16}$$

Therefore, for each element, we can generate a local stiffness matrix

$$A^K \equiv A^K_{lm} = \int_K \nabla\phi_l \cdot \nabla\phi_m dx, \quad l, m = 1, 2, \ldots, \tag{4.17}$$

where l and m are the local numbering system for the element. They each correspond to an index in the global stiffness matrix. For linear basis functions, this can be determined by looking at the values in the element connectivity array at position l and m. For example, if our connectivity array was [3 5 7] and we were looking at entry $A^K_{1,2}$, then the global entry would be $A_{3,5}$. All that is required to compute the entries of the global matrix is to loop over all the elements in turn and generate their local stiffness matrix. Once this is done the values can be added in to their corresponding entry in the global stiffness matrix. The load vector F is populated in a similar fashion, using a local load vector whose entries are added in to the global load vector. □

To apply boundary conditions (BCs), we must alter the system slightly to satisfy them. For example, if we have homogeneous Dirichlet BCs, the way we implement them into our system is to first zero the rows and columns, in A, associated with the boundary nodes. We then set the diagonal entries of those rows and columns to 1 and finally zero the corresponding entry in the load vector F. If we were to have inhomogeneous Dirichlet BCs then we employ a similar technique but subtract from the load vector the column entries multiplied by the boundary values.

Finally, to compute the finite element solution to a problem, we just need to solve the matrix system to calculate U. Once this is complete, U can just be substituted back in to the definition of the finite element solution u_h (4.7) to give the numerical approximation to the solution of the PDE.

4.3 The Elements

Although there are many element types that can be used, for the purposes of our implementation, we will be focusing on intervals in one dimension and triangles and quadrilaterals in two dimensions. We will be using a local coordinate system (ξ, η) to define the elements, allowing us to easily construct the basis functions and perform numerical integration on the elements later on. For each element we shall supply the relevant mappings

in order to map from the local reference element to a global element in the mesh.

4.3.1 The Interval

In one dimension, we shall be using the reference interval $\hat{I}$ defined on $(-1, 1)$. Figure 4 below depicts this interval.

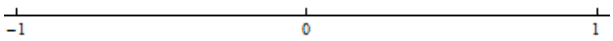


Figure 4: Interval element.

We need to be able to map this to any interval I within our finite element mesh. If we know the start and end point (a, b) of the interval we wish to map to, then we are able to map to any coordinate x within that interval using the mapping $x : \hat{I} \mapsto I$ where

$$x = a + \frac{b-a}{2}(\xi + 1). \tag{4.18}$$

4.3.2 The Quadrilateral

The first two dimensional element we shall consider is the quadrilateral. The reference element that we will use is the quadrilateral $\hat{Q}$ defined on $(-1, 1) \times (-1, 1)$, seen below in Figure 5.

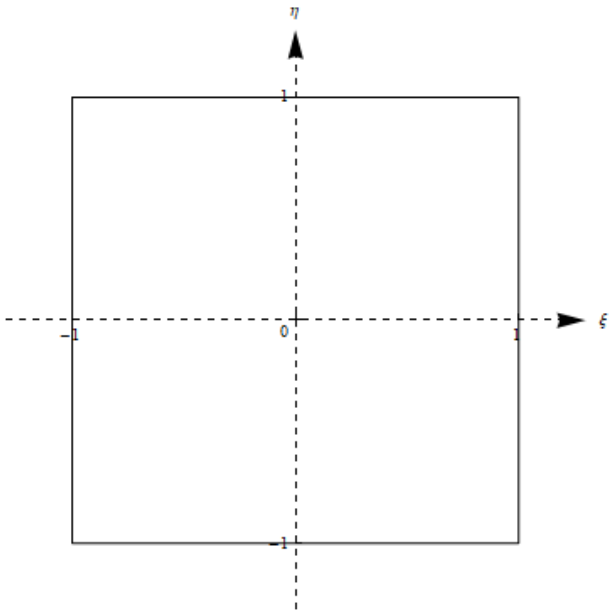


Figure 5: Quadrilateral element.

Similarly to the interval, we need to be able to map this element to any quadrilateral Q in our 2D finite element mesh. We denote the global coordinates by $\boldsymbol{x} = (x, y)$, and let $\boldsymbol{x_1}, \ldots, \boldsymbol{x_4}$ be the coordinates of the four vertices of the global quadrilateral. It is important that these the vertices are labelled in an anticlockwise fashion, in the same manner to the local quadrilateral, as each one should correspond to a vertex on the local quadrilateral.

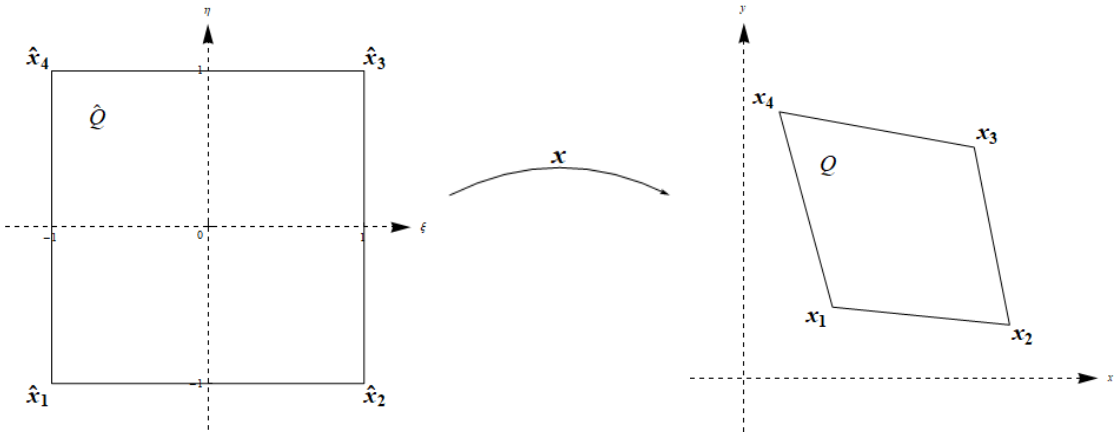


Figure 6: Mapping from reference quadrilateral to global quadrilateral.

For the quadrilateral element we have the mapping $\boldsymbol{x} : \hat{Q} \mapsto Q$ where

$$\boldsymbol{x} = \frac{(1-\xi)(1-\eta)\boldsymbol{x_1} + (1+\xi)(1-\eta)\boldsymbol{x_2} + (1+\xi)(1+\eta)\boldsymbol{x_3} + (1-\xi)(1+\eta)\boldsymbol{x_4}}{4}. \quad (4.19)$$

4.3.3 The Triangle

The final element type we shall consider for our implementation is the triangle. Our reference triangle $\hat{T}$ is defined by

$$\hat{T} = \{(\xi, \eta) \in \mathbb{R}^2 : \quad -1 < \xi, \eta \quad \text{and} \quad \xi + \eta < 0\}, \tag{4.20}$$

as shown in Figure 7 below.

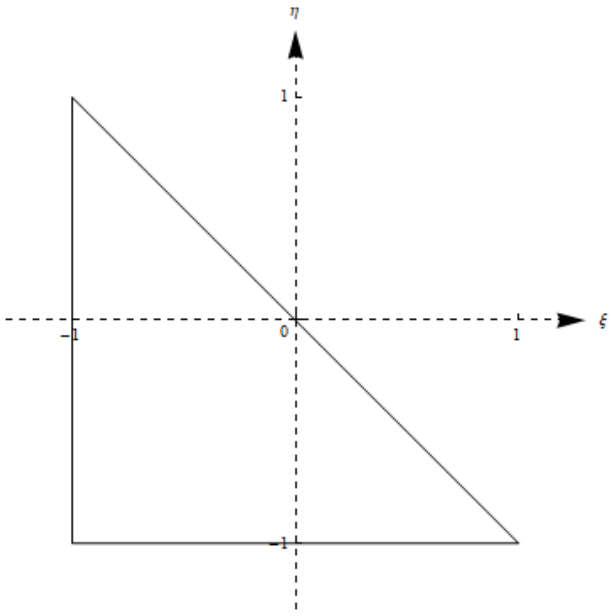


Figure 7: Triangle element.

Once again, we need to map from our reference triangle $\hat{T}$ to a global triangle T in the mesh. Similarly to the quadrilateral, the global triangle nodes are denoted by $\boldsymbol{x_1}, \ldots, \boldsymbol{x_3}$ and must be labelled in an anticlockwise manner. The mapping for any coordinate $\boldsymbol{x}$ in the global triangle is given by

$$\boldsymbol{x} = -\left(\frac{\xi + \eta}{2}\right)\boldsymbol{x_1} + \left(\frac{\xi + 1}{2}\right)\boldsymbol{x_2} + \left(\frac{\eta + 1}{2}\right)\boldsymbol{x_3}. \tag{4.21}$$

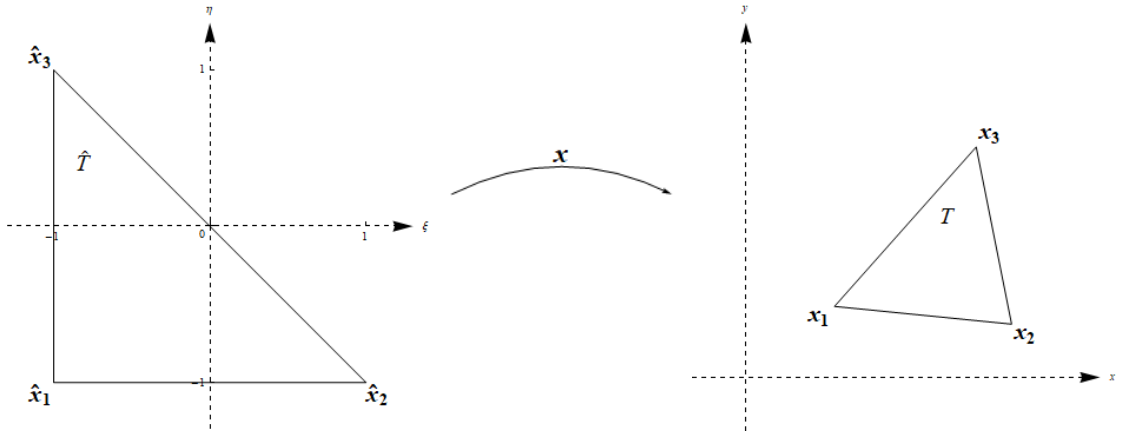


Figure 8: Mapping from reference triangle to global triangle.

4.3.4 Transformation of Basis Functions

We have now introduced the mappings for all the relevant elements in order to map from the local element to the global element. These are all in the form

$$\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{\xi}). \tag{4.22}$$

It is necessary to be able to transform the basis functions and their derivatives for procedures such as integration on our reference element (see Section 6 for more on this). Let us denote the local basis functions by $\varphi(\boldsymbol{\xi})$ and the global basis functions by $\phi(\boldsymbol{x})$. For the basis functions $\phi(\boldsymbol{x})$, we just substitute in the mapping, i.e. $\varphi(\boldsymbol{\xi}) = \phi(\boldsymbol{F}(\boldsymbol{\xi}))$, and evaluate the function. In the case of the derivatives of the basis functions, we need to employ some other techniques.

Firstly, the Jacobian matrix J of the mappings is defined by

$$J_{ij} = \frac{\partial x_i}{\partial \xi_j}. \tag{4.23}$$

The chain rule yields that

$$\frac{\partial \varphi}{\partial \xi_j} = \sum_i \frac{\partial x_i}{\partial \xi_j} \frac{\partial \phi}{x_i}, \quad \text{for } j = 1, 2, 3, \ldots. \tag{4.24}$$

Hence, we can write

$$\begin{bmatrix} \frac{\partial \varphi}{\partial \xi_1} \\ \frac{\partial \varphi}{\partial \xi_2} \\ \frac{\partial \varphi}{\partial \xi_3} \\ \vdots \end{bmatrix} = J^T \begin{bmatrix} \frac{\partial \phi}{x_1} \\ \frac{\partial \phi}{x_2} \\ \frac{\partial \phi}{x_3} \\ \vdots \end{bmatrix} \quad \Longleftrightarrow \quad \begin{bmatrix} \frac{\partial \phi}{x_1} \\ \frac{\partial \phi}{x_2} \\ \frac{\partial \phi}{x_3} \\ \vdots \end{bmatrix} = J^{-T} \begin{bmatrix} \frac{\partial \varphi}{\partial \xi_1} \\ \frac{\partial \varphi}{\partial \xi_2} \\ \frac{\partial \varphi}{\partial \xi_3} \\ \vdots \end{bmatrix}, \tag{4.25}$$

or

$$\nabla \phi = J^{-T} \hat{\nabla} \varphi, \tag{4.26}$$

where $\hat{\nabla} = \left(\frac{\partial}{\partial \xi_1}, \frac{\partial}{\partial \xi_2}, \frac{\partial}{\partial \xi_3}, \dots \right)^T$.

4.4 Hierarchical Bases

We use the term *hierarchical* or *modal* to describe the way we define the higher order basis functions. As the name may suggest, hierarchical basis functions work by essentially "layering" functions on top of one another as the order increases. The basis functions we shall be using are based off of the Lobatto shape functions, which require us to first introduce the Legendre polynomials.

4.4.1 The Legendre Polynomials

The Legendre polynomials are a class of polynomial functions that form an orthogonal basis of the L_2 Space on the interval $(-1, 1)$. There are several ways in which they can be defined but for the purpose of implementation we shall generate them from the following recursive formula (4.27). The notation is not to be confused with that of the Lebesgue space (Definition 2.1.3) and it will be made clear in which context the Legendre polynomials are used.

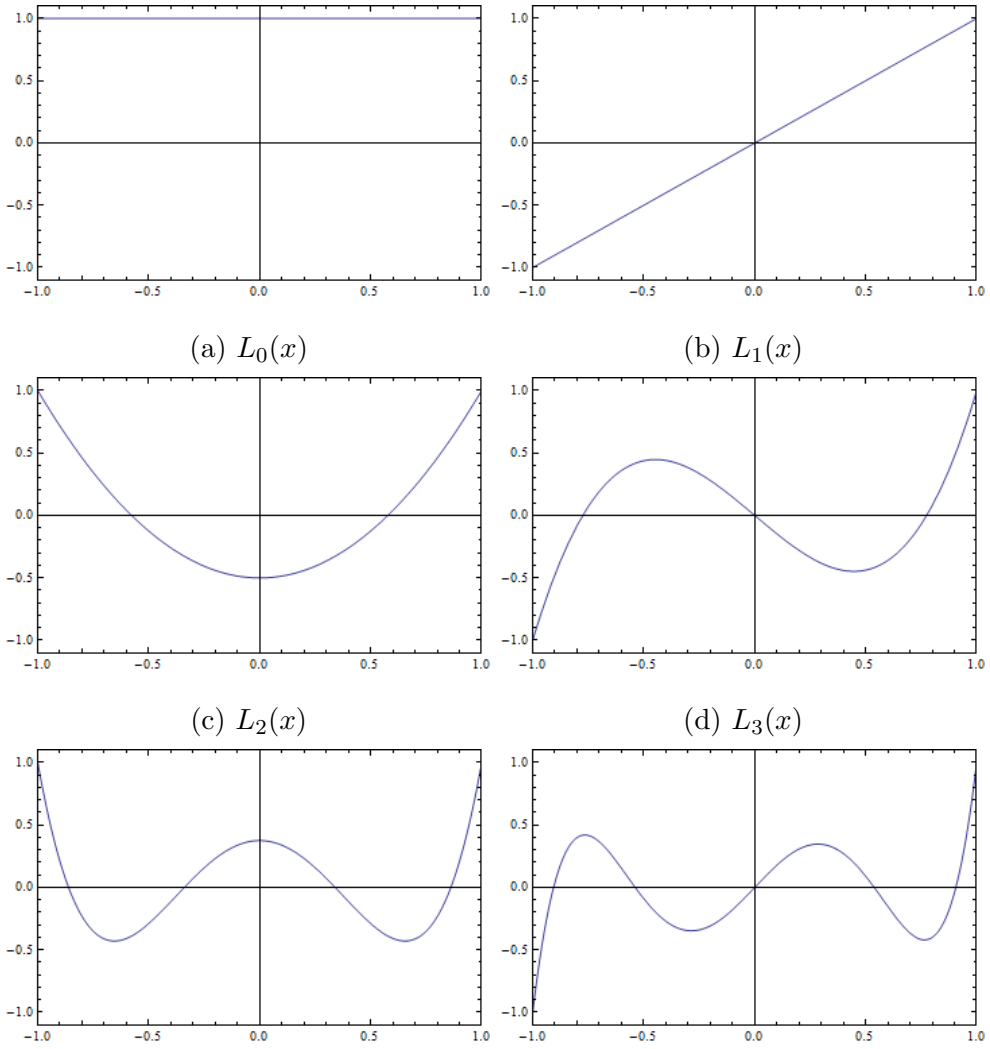
$$\begin{aligned} L_0(x) &= 1, \\ L_1(x) &= x, \\ L_{n+1}(x) &= \frac{2n+1}{n+1} x L_n(x) - \frac{n}{n+1} L_{n-1}(x), \quad n \geq 1. \end{aligned} \tag{4.27}$$

The orthogonality is given by

$$\int_{-1}^{1} L_n(x)L_m(x)dx = \begin{cases} \frac{2}{2n+1}, & \text{if } n = m, \\ 0, & \text{otherwise.} \end{cases} \tag{4.28}$$

For demonstrative and implementation purposes, we shall give the first six Legendre polynomials and illustrate each one with a plot.

$$\begin{aligned} L_0(x) &= 1 \\ L_1(x) &= x \\ L_2(x) &= \frac{1}{2}(3x^2 - 1) \\ L_3(x) &= \frac{1}{2}(5x^3 - 3x) \\ L_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\ L_5(x) &= \frac{1}{8}(63x^5 - 70x^3 + 15x) \end{aligned} \tag{4.29}$$



(a) $L_0(x)$ (b) $L_1(x)$

(c) $L_2(x)$ (d) $L_3(x)$

(e) $L_4(x)$ (f) $L_5(x)$

Figure 9: Plots of the first six Legendre polynomials.

We can see from the plots in Figure 9 that the Legendre polynomials possess the property that

$$L_n(1) = 1 \text{ and } L_n(-1) = (-1)^n, \qquad \forall n \geq 0. \tag{4.30}$$

4.4.2 The Lobatto Shape Functions

We may now use the Legendre polynomials (4.27) to formulate another class of functions, namely the Lobatto shape functions (Šolín, 2006), which will be paramount in the design of the basis functions on our elements.

They are generated using the following formula:

$$\begin{aligned}
\ell_0(x) &= \frac{1-x}{2}, \\
\ell_1(x) &= \frac{1+x}{2}, \\
\ell_n(x) &= \frac{1}{\|L_{n-1}(x)\|_{L_2(-1,1)}} \int_{-1}^{x} L_{n-1}(\xi)d\xi, \qquad \forall n \geq 2.
\end{aligned} \tag{4.31}$$

The L_2 norm of the $(n-1)$th Legendre polynomial on $(-1,1)$ can be written as

$$\begin{aligned}
\|L_{n-1}(x)\|_{L_2(-1,1)} &= \left(\int_{-1}^{1} |L_{n-1}(x)|^2 \, dx \right)^{\frac{1}{2}} \\
&= \left(\frac{2}{2(n-1)+1} \right)^{\frac{1}{2}} \qquad \text{(by orthogonality property (4.28))} \\
&= \sqrt{\frac{2}{2n-1}}.
\end{aligned}$$

Hence we can rewrite the formula for $\ell_n(x)$ as

$$\ell_n(x) = \sqrt{\frac{2n-1}{2}} \int_{-1}^{x} L_{n-1}(\xi)d\xi, \qquad \forall n \geq 2. \tag{4.32}$$

It is obvious from this formula that $\ell_n(-1) = 0$ for all $n \geq 2$. By the orthogonality of the Legendre polynomials (4.28), it is also clear that $\ell(1) = 0$ for all $n \geq 2$ as

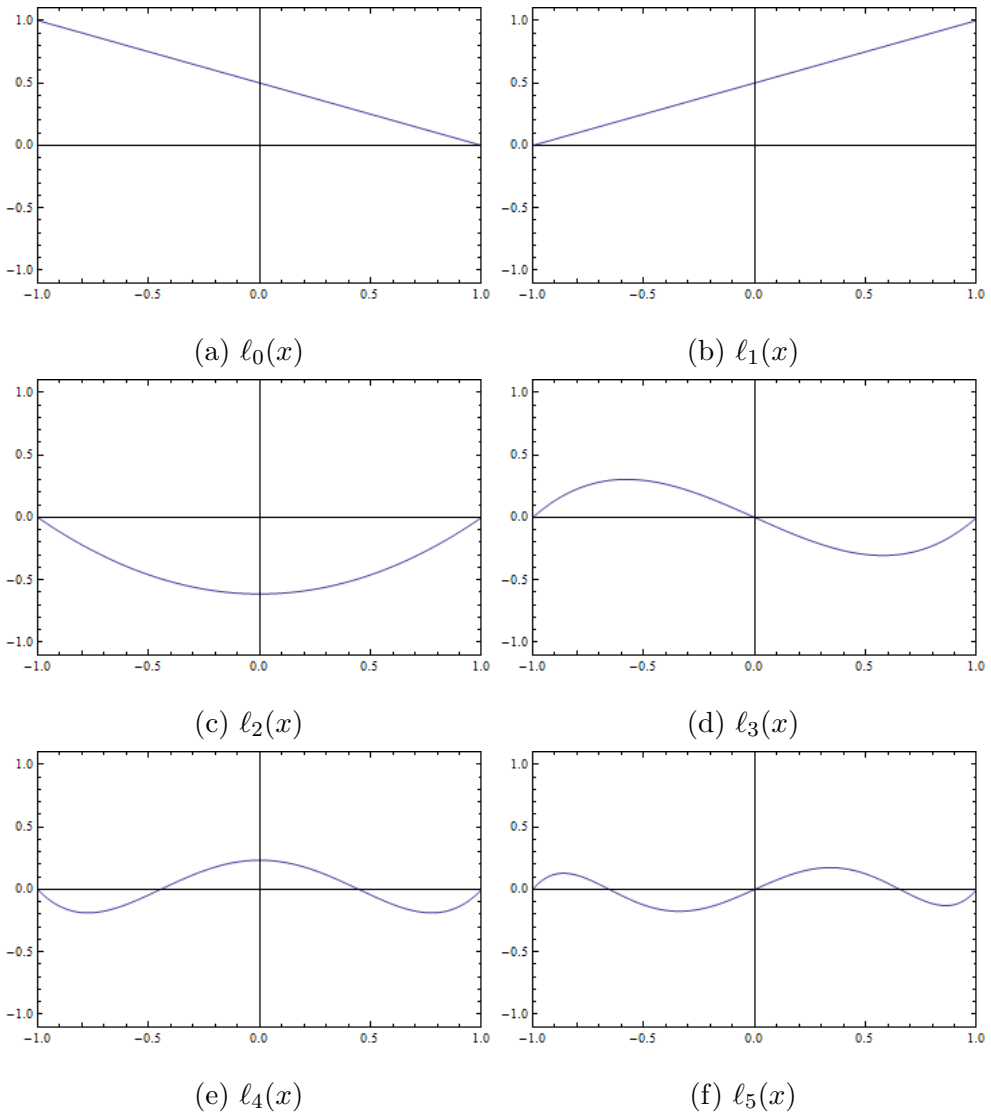
$$\int_{-1}^{1} L_{n-1}(\xi)d\xi = \int_{-1}^{1} L_{n-1}(\xi) \cdot 1 \; d\xi \equiv \int_{-1}^{1} L_{n-1}(\xi) L_0(\xi) d\xi = 0.$$

Evidently, the Lobatto shape functions $\ell_0, \ell_1, \ldots, \ell_p$ form a complete basis of the space $P_p(-1,1)$, the space of polynomials of degree at most p in the interval $(-1,1)$ (Šolín, Segeth, and Doležel, 2003).

Similarly to the Legendre polynomials, we shall list the first 6 Lobatto shape functions

and give their plots.

$$
\begin{aligned}
\ell_0(x) &= \frac{1-x}{2} \\
\ell_1(x) &= \frac{1+x}{2} \\
\ell_2(x) &= \frac{1}{2}\sqrt{\frac{3}{2}}(x^2-1) \\
\ell_3(x) &= \frac{1}{2}\sqrt{\frac{5}{2}}(x^2-1)x \\
\ell_4(x) &= \frac{1}{8}\sqrt{\frac{7}{2}}(x^2-1)(5x^2-1) \\
\ell_5(x) &= \frac{1}{8}\sqrt{\frac{9}{2}}(x^2-1)(7x^2-3)x
\end{aligned}
\tag{4.33}
$$



(a) $\ell_0(x)$ (b) $\ell_1(x)$

(c) $\ell_2(x)$ (d) $\ell_3(x)$

(e) $\ell_4(x)$ (f) $\ell_5(x)$

Figure 10: Plots of the first six Lobatto shape functions.

Now that we have demonstrated the Lobatto shape functions, we can look at our choice of basis functions on interval, triangle and quadrilateral elements. Each element type's basis functions can be either *vertex* (nodal) functions, *edge* functions or *bubble* functions. Vertex functions are such that they are equal to 1 at a vertex and 0 at all the other vertices. Edge functions are 0 at all vertices but act along an edge. Bubble functions are equal to 0 at all the vertices and edges but act on the interior of the element. We will now show how the different functions vary for different elements and visually represent them with plots. As the Lobatto shape functions are defined on $(-1, 1)$, we shall be using the local reference elements of each type, introduced in Section 4.3, to define the basis functions on.

4.4.3 Interval Bases

The interval element is made up only of vertex functions and bubble functions. We shall denote the vertex basis functions for intervals by $\varphi_{\hat{I}}^{v_j}$, $j = 1, 2$ and the bubble basis functions for intervals by $\varphi_{\hat{I}}^{b_k}$, $2 \leq k \leq p$. For intervals, the design of the basis functions is extremely simple. If our chosen polynomial degree p is equal to 1 then we only require the vertex functions. For $p \geq 2$, we "layer on" the bubble functions of degree up to and including p. Each basis function corresponds to a single Lobatto shape function (4.31) i.e:

$$\begin{aligned}
\varphi_{\hat{I}}^{v_1}(\xi) &= \ell_0(\xi), \\
\varphi_{\hat{I}}^{v_2}(\xi) &= \ell_1(\xi), \\
\varphi_{\hat{I}}^{b_2}(\xi) &= \ell_2(\xi), \\
&\vdots \\
\varphi_{\hat{I}}^{b_p}(\xi) &= \ell_p(\xi).
\end{aligned} \tag{4.34}$$

For example, if $p = 3$ then our hierarchical basis functions on the element would look as follows:

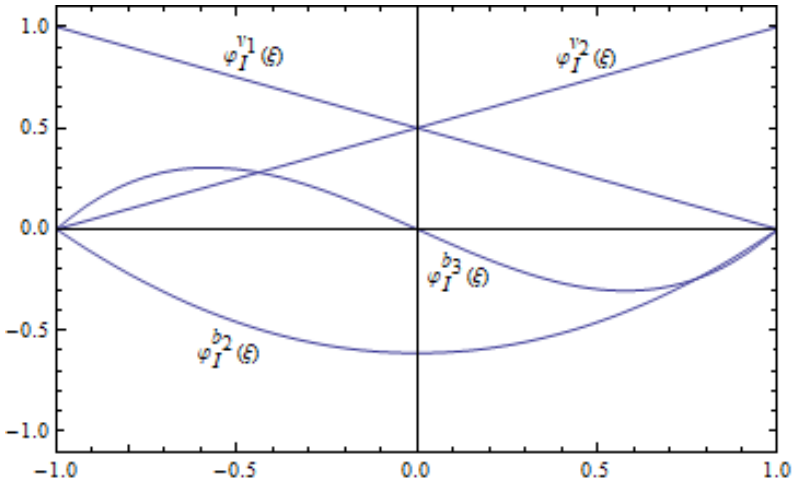


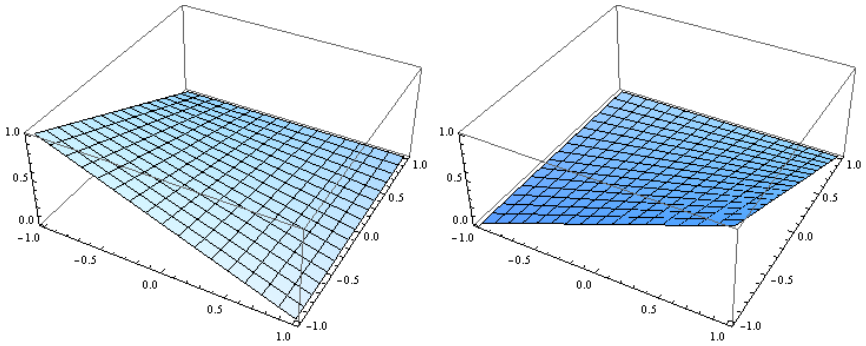
Figure 11: Hierarchical basis functions on the reference interval for $p = 3$.

4.4.4 Quadrilateral Bases

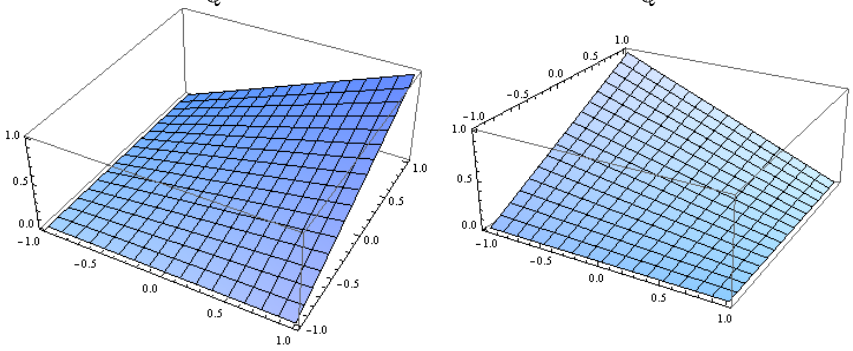
Quadrilateral elements require all three types of basis function. In our implementation, we only look at linear basis functions on our two dimensional elements. Hence we shall give only the vertex basis functions here. Similarly to the interval element, we shall denote the vertex functions for quadrilaterals by $\varphi_{\hat{Q}}^{v_j}$, $j = 1, \ldots, 4$. In order to make up the basis functions we take varying combinations of the Lobatto shape functions. The vertex functions are defined as

$$\begin{aligned}
\varphi_{\hat{Q}}^{v_1}(\xi, \eta) &= \ell_0(\xi)\ell_0(\eta), \\
\varphi_{\hat{Q}}^{v_2}(\xi, \eta) &= \ell_1(\xi)\ell_0(\eta), \\
\varphi_{\hat{Q}}^{v_3}(\xi, \eta) &= \ell_1(\xi)\ell_1(\eta), \\
\varphi_{\hat{Q}}^{v_4}(\xi, \eta) &= \ell_0(\xi)\ell_1(\eta).
\end{aligned} \tag{4.35}$$

Graphically, these look as follows:



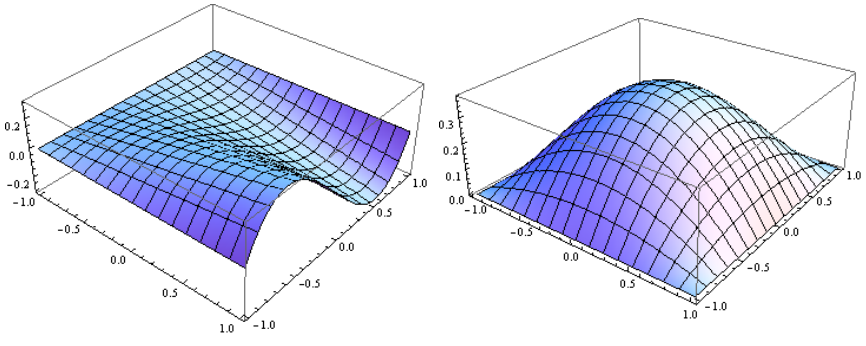
(a) $\varphi^{v_1}_{\hat{Q}}(\xi,\eta)$ (b) $\varphi^{v_2}_{\hat{Q}}(\xi,\eta)$



(c) $\varphi^{v_3}_{\hat{Q}}(\xi,\eta)$ (d) $\varphi^{v_4}_{\hat{Q}}(\xi,\eta)$

Figure 12: Plots of reference quadrilateral vertex basis functions.

Although we are not going to look at higher order bases on two dimensional elements, we shall give a simple example of what an edge and bubble function might look like on the quadrilateral element, for completeness.



(a) $\varphi^{e_2}_{3,\hat{Q}}(\xi,\eta)$ (b) $\varphi^{b_{2,2}}_{\hat{Q}}(\xi,\eta)$

Figure 13: Plots of reference quadrilateral edge basis function for $p = 3$ and bubble function for $p = 2$.

The hierarchical structure of the basis functions in two dimensions decouples the local

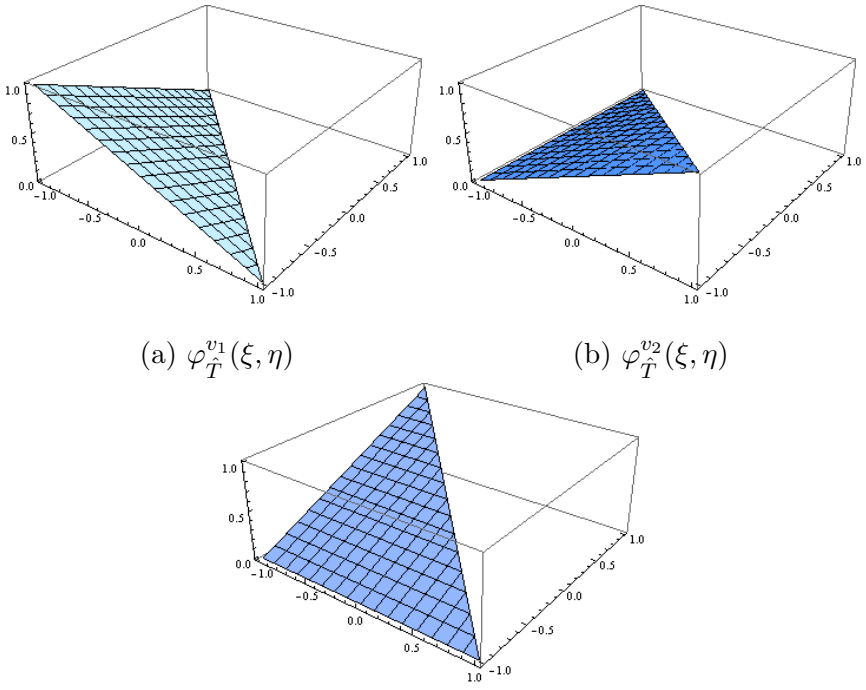
orders of p in the element interior and on the edges, meaning that the p can differ across basis function types. For more information on this see Šolín, Segeth, and Doležel (2003).

4.4.5 Triangle Bases

Once again, we use a similar notation for the basis functions on a triangle element. The vertex functions are denoted by $\varphi_{\hat{T}}^{v_j}$, $j = 1, \ldots, 3$. For the triangle element, these are given by the specially defined functions

$$\begin{aligned} \varphi_{\hat{T}}^{v_1}(\xi, \eta) &= -\frac{\xi + \eta}{2}, \\ \varphi_{\hat{T}}^{v_2}(\xi, \eta) &= \frac{1 + \xi}{2}, \\ \varphi_{\hat{T}}^{v_3}(\xi, \eta) &= \frac{1 + \eta}{2}. \end{aligned} \tag{4.36}$$

Notice that these functions are the coefficients of $\boldsymbol{x_1}, \ldots, \boldsymbol{x_3}$ in the local to global mapping for a triangle, given in Section 4.3.3. These are represented graphically by

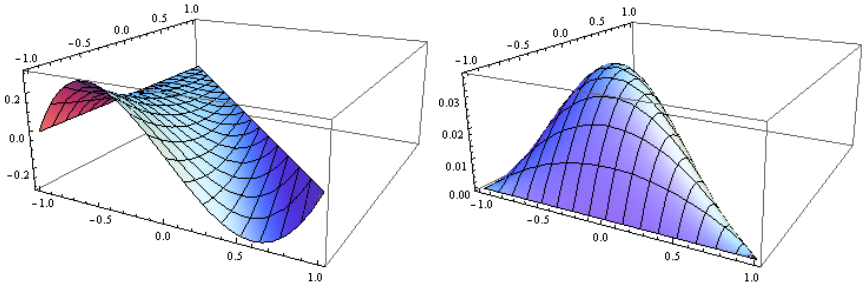


(a) $\varphi_{\hat{T}}^{v_1}(\xi, \eta)$ (b) $\varphi_{\hat{T}}^{v_2}(\xi, \eta)$

(c) $\varphi_{\hat{T}}^{v_3}(\xi, \eta)$

Figure 14: Plots of reference triangle vertex basis functions.

In conjunction with the quadrilateral element, although we are not going to be using higher order functions in two dimensions, we give an example of what the bases may look like for an edge and bubble function on the triangle.



(a) $\varphi^{e_1}_{3,\hat{T}}(\xi,\eta)$ (b) $\varphi^{b_{1,1}}_{\hat{T}}(\xi,\eta)$

Figure 15: Plots of reference Triangle edge basis function for $p = 3$ and the bubble function for $p = 3$.

5 Implementation in C++

The key focus of this dissertation is looking at how an implementation of the finite element method can be developed. There are many different ways that this can be approached, however, our implementation is focused on an object-oriented design using the language C++, specifically C++11. Currently, there are some very extensive libraries written in a variety of languages. Some of these include the deal.ii library (Bangerth, Hartmann, and Kanschat, 2007), which is also written in C++, and FEniCS (Logg, Mardal, and Wells, 2012), which is written in both Python and C++.

This section will discuss the structure of our finite element package and demonstrate, with code snippets, how the implementation was constructed. Later on, in Section 5, we shall show how our package can be used to evaluate a variety of problems in both one and two dimensions.

5.1 General Class Design

We shall start by discussing the core classes that are needed in order to begin the implementation. Following this, we shall take an individual look at the main classes and explore the methods that each contain.

Below in Figure 16, we give a basic UML class diagram for the structure of our implementation.

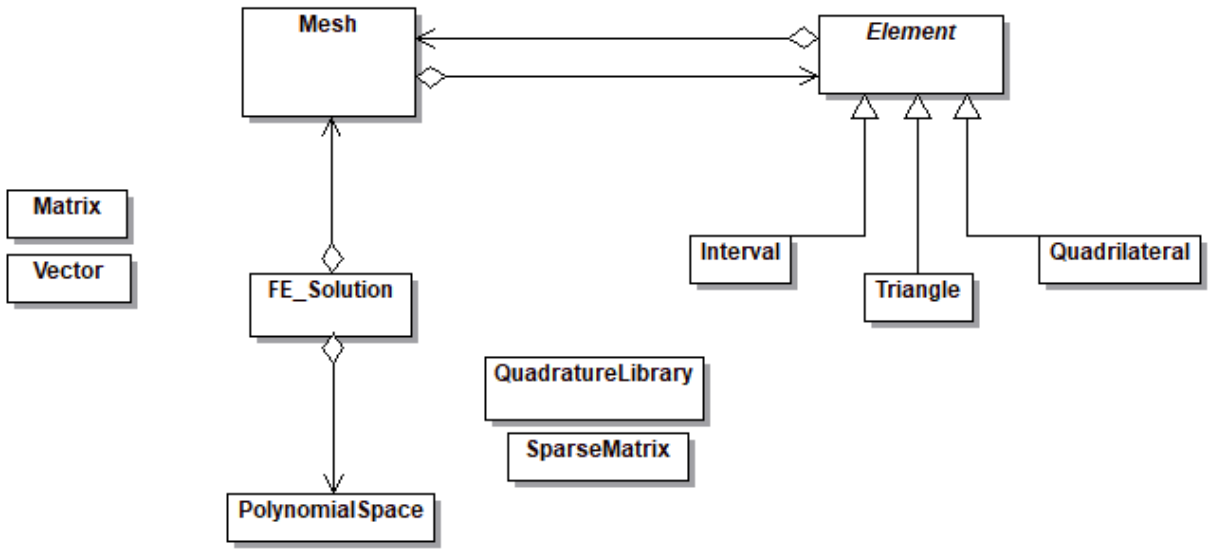


Figure 16: UML class diagram for the whole project.

As we can see, the main class that is required by the other classes (and the class that was developed first) is the *Mesh* class. This class stores everything to do with the mesh triangulation. We shall talk about it in more detail in the next section (5.2). Within this class, we need to store the elements themselves and their attributes. An abstract class, *Element*, was created as a data structure to store all data to do with the elements i.e. the element connectivity array, the element type etc. This takes inspiration from the work of Warburton (1999), which details the storage one might need for such a class. The specific element types then each have their own individual classes which inherit from the Element class. We shall discuss these further in Section 5.3.

The *FE_Solution* class is where the finite element solution is formulated. It contains methods for procedures such as initialising the element degrees of freedom and it has storage for items such as the polynomial degree of the basis functions and for the solution vector. It only requires a reference to the Mesh so that it can access information about the elements. Contained in this class is also the element polynomial space for each element. This information is held within a data structure, *PolynomialSpace*, which stores the basis functions, the gradient of the basis functions and the degrees of freedom on each element for differing polynomial degrees. In Section 5.4 we shall give an insight into the Element and Polynomial Space classes and then in Section 7 we will demonstrate how the finite element solution is generated.

Two stand-alone classes which we also require are a quadrature library to evaluate integrals and a data structure for storing sparse matrices. Similarly to the other classes, we will briefly talk about both later on in Sections 5.5 and 5.6. We shall then proceed to Section 6 which focuses solely on numerical quadrature over the elements, from a theoretical and computational perspective.

In all of our classes, we use the data structures *Vector* and *Matrix* to store arrays instead of the default C++ array. These data structures are an extension of those from Pitt-Francis and Whiteley (2012). Using these instead of the default C++ arrays gives us the flexibility to implement our own specific methods and control the data structure in any way we see fit.

5.2 Mesh

As stated previously, the Mesh class is designed to hold all aspects of the mesh triangulation. This includes details such as the grid points, the number of nodes, the number of elements and the dimension of the problem etc. We can see from the UML representation below (Figure 17) that we have private variables for all of these items.

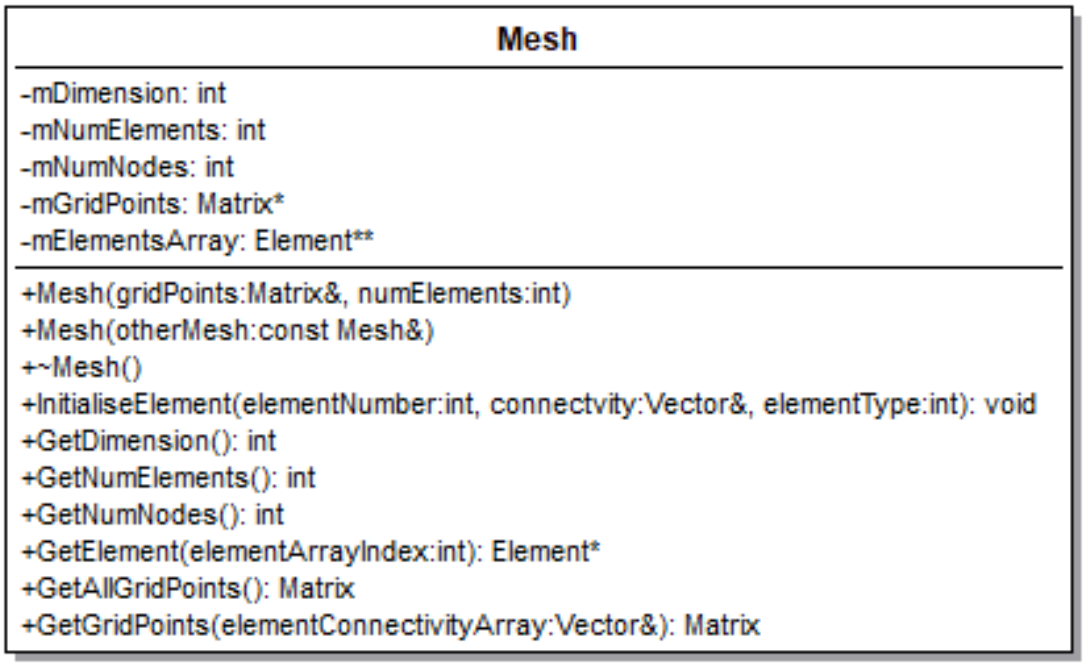


Figure 17: UML representation for the Mesh class.

All of this information is passed in to the specialised constructor of Mesh in the form of

```
Mesh(Matrix& gridPoints, int numElements);
```

The dimension and number of nodes can then be easily computed from the grid points matrix as it is of size $Dimension \times NumNodes$.

We also need to store an array of type Element to hold the attributes for each individual element. In order to initialise the element in each entry of the array we use the method

```
void InitialiseElement(int elementNumber, Vector& connectvity, int elementType);
```

This works by taking an input of the element number, the connectivity array associated with that element and the element type (i.e. interval, triangle or quadrilateral). Running over a switch statement, it then instantiates an object of the relevant element type at the correct position in the element array. This is demonstrated below in Listing 1.

```
void Mesh::InitialiseElement(int elementNumber, Vector& connectvity, int elementType)
{
    switch(elementType)
    {
    case 0:
        {
            mElementsArray[elementNumber - 1] = new Interval(connectvity, *this);
        } break;
    case 1:
        {
            mElementsArray[elementNumber - 1] = new Triangle(connectvity, *this);
        } break;
    case 2:
        {
            mElementsArray[elementNumber - 1] = new Quadrilateral(connectvity, *this);
        } break;
    }
}
```

Listing 1: Method to initialise an object of Element in the element array stored in Mesh.

Other than a copy constructor and a destructor for the class, the rest of our public methods are all getters for the key information that one might require outside of the class.
As discussed, in order for the Mesh class to be properly populated we require the use of the Element class. We will discuss this in the next section, but once this is completed we then have a fully functional mesh handler that could be used for many other projects, not just a finite element implementation.

5.3 Element

The Element class is an abstract class that is devised to hold the attributes of specific element. The only storage required is the element connectivity array associated with the element. Within the class, we declare a public enumerator that holds the different element types.

```
enum ElementType
{
  Interval, Triangle, Quadrilateral,
};
```

Listing 2: Enumerator for the type of element.

This can easily be added to in the future if we wish to implement more elements. A value is associated with each type, starting at 0 (i.e. Interval = 0, Triangle = 1, Quadrilateral = 2). We create individual classes for each of these element types which are derived classes that inherit from Element. These will be discussed later in this section.

We also require a reference to the instance of Mesh that the Element is created within. This then allows us to return the specific grid points for each element by utilising the connectivity array. As the class Element is instantiated within Mesh but we also require the reference to Mesh within Element, a forward declaration (Nackman et al., 2001) of the class Mesh is needed in the Element class. We do not create any new memory for Mesh in Element, we simply hold the reference.

Below in Figure 18 we can see the UML representation for the Element class. All of the methods in the lower box are virtual methods and all the mapping methods and the top two getters are pure virtual. These are pure virtual as they are different for each element type.

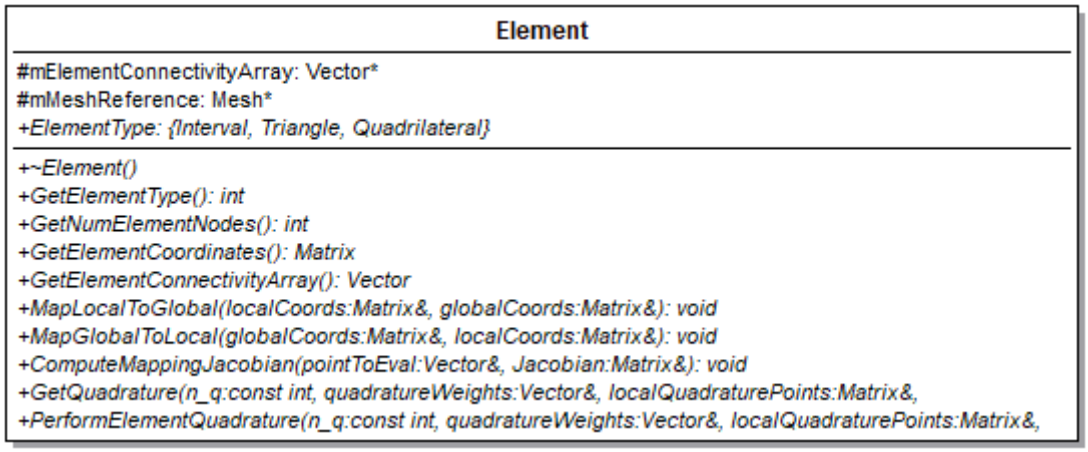


Figure 18: UML representation for the Element class.

The getters for the element coordinates and element connectivity array perform in the same way for every element type, hence they can be implemented in the Element class itself. Similarly, the quadrature methods, which call the member functions from the QuadratureLibrary Class, can be applied to all the element types from one method in the Element class. This is due to the fact that the GetElementType function can be called from within the quadrature method to tell the QuadratureLibrary which quadrature rule is needed.

5.3.1 Interval (Triangle and Quadrilateral)

For the interval element, we create the class Interval. It inherits nearly all its methods and attributes from Element. The only exception is an extra version of the ComputeMappingJacobian method that has an input of an evaluation point of type double instead of a one dimensional Vector. i.e.

```
void ComputeMappingJacobian(double pointToEval, Matrix& Jacobian);
void ComputeMappingJacobian(Vector& pointToEval, Matrix& Jacobian);
```

This is only done for intervals as they are the only one dimensional element, although it does not actually matter for this implementation as the Jacobian of the mapping from local to global intervals only depends on the end points of the interval (see Listing 3). This may not be the case, however, if a different reference element was used.

```
1  void Interval::ComputeMappingJacobian(double pointToEval, Matrix& Jacobian)
2  {
3      Matrix* nodes = new Matrix (GetElementCoordinates());
4      assert(Jacobian.GetNumberOfRows() == Jacobian.GetNumberOfColumns());
5      assert(Jacobian.GetNumberOfRows() == 1);
6      assert(nodes->GetNumberOfRows() == 1);
7      assert(nodes->GetNumberOfColumns() == 2);
8
9      Jacobian(1,1) = 0.5*((*nodes)(1,2) - (*nodes)(1,1));
10
11     delete nodes;
12 }
```

Listing 3: Method to compute the Jacobian of the local to global mapping for an interval.

In order to retrieve the element type, a getter for the element type is easily implemented by utilising the enumerator defined in the element class. This getter can be seen below in Listing 4.

```
1  int Interval::GetElementType() const
2  {
3      return ElementType::Interval;
4  }
```

Listing 4: Method to retrieve the element type for an interval.

For the mapping methods, we require the mapping from local to global intervals that is detailed in Section 4.3.1. This is depicted in Listing 5. We call a number of assert

statements to make sure that the relevant data matches up, also allowing us to easily debug if required.

```
void Interval::MapLocalToGlobal(Matrix& localCoords, Matrix& globalCoords)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(localCoords.GetNumberOfRows() == globalCoords.GetNumberOfRows());
    assert(localCoords.GetNumberOfColumns() == globalCoords.GetNumberOfColumns());
    assert(localCoords.GetNumberOfRows() == 1);
    assert(nodes->GetNumberOfRows() == 1);
    assert(nodes->GetNumberOfColumns() == 2);

    for (int j=1; j<=globalCoords.GetNumberOfColumns(); j++)
    {
        globalCoords(1,j) = (*nodes)(1,1) + 0.5*((*nodes)(1,2)-(*nodes)(1,1))*(
            localCoords(1,j)+1);
    }

    delete nodes;
}
```

Listing 5: Method to map the local coordinates to global coordinates for the Interval element.

The global to local mapping method is not actually used in our implementation but it was programmed anyway as there are situations where it can be useful if the implementation is extended. This mapping was achieved by a simple rearrangement of the local to global mapping, but works in the same way. We detail this below in Listing 6.

```
void Interval::MapGlobalToLocal(Matrix& globalCoords, Matrix& localCoords)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(globalCoords.GetNumberOfRows() == localCoords.GetNumberOfRows());
    assert(globalCoords.GetNumberOfColumns() == localCoords.GetNumberOfColumns());
    assert(globalCoords.GetNumberOfRows() == 1);
    assert(nodes->GetNumberOfRows() == 1);
    assert(nodes->GetNumberOfColumns() == 2);

    for (int j=1; j<=localCoords.GetNumberOfColumns(); j++)
    {
        localCoords(1,j) = ((*nodes)(1,1) + (*nodes)(1,2) - 2*globalCoords(1,j))/((*nodes
            )(1,1) - (*nodes)(1,2));
    }

    delete nodes;
```

```
16 }
```

Listing 6: Method to map the global coordinates to local coordinates for the Interval element.

These methods all work in exactly the same way for the Quadrilateral and Triangle classes. Both these classes inherit directly from the Element class and have no extra methods implemented. For the full code, see Appendix A.2 and A.3.

5.4 FE_Solution

Our FE_Solution class is built to deal with the finite element approximation to a problem. It has private storage for items such as the number of degrees of freedom (DOFs), the specified polynomial degree for the problem and the solution vector. The methods and variables contained in FE_Solution are detailed in the UML diagram in Figure 19 below.

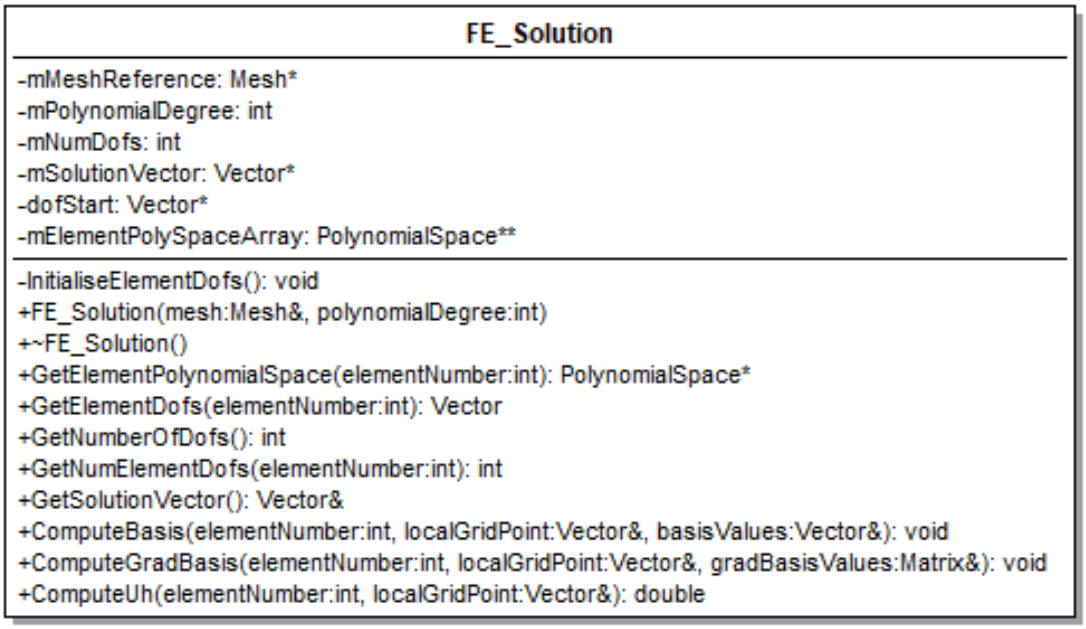


Figure 19: UML representation for the FE_Solution class.

As we can see, other than a specialised constructor and getters for the relevant information, the main methods are those that compute the basis functions on each element, the gradient of the basis functions and an initialiser to initialise the element degrees of freedom. The specialised constructor

```
FE_Solution(Mesh& mesh, int polynomialDegree);
```

takes input arguments of the mesh reference and the polynomial degree. It then uses this information to populate the private variables of the class. In Listing 7 below, we can see the full method for the specialised constructor.

```
1  FE_Solution::FE_Solution(Mesh& mesh, int polynomialDegree)
2  {
3      mMeshReference = &mesh;
4      mPolynomialDegree = polynomialDegree;
5
6      InitialiseElementDofs();
7      mNumDofs = (*dofStart)(dofStart->GetSize()) - 1;
8
9      mSolutionVector = new Vector (mNumDofs);
10 }
```

Listing 7: Specialised constructor for the FE_Solution class.

Although we could write the contents of the InitialiseElementDofs method inside the constructor, we choose to write it in its own private function to better manage the code. The full method for InitialiseElementDOFS can be seen in Appendix A.4. The first part of the initialisation creates a new polynomial space for each element. These are stored in an array of PolynomialSpace classes in a similar way to the Element array in the Mesh class. Once the PolynomialSpace has been created, we proceed to set up and populate the storage for the degrees of freedom. The chosen storage method that we use for the DOFS is specifically designed to relieve memory consumption. We will discuss this now in Section 5.4.1.

5.4.1 Degrees of Freedom Storage

Obviously, we need to have access to the degrees of freedom on each element. We could store this as an array for every individual element but this can be memory intensive and there is a more efficient way of storing the element degrees of freedom. First, we shall discuss the numbering of the DOFs across the domain. This is done by first numbering all the vertex functions by their corresponding node, like so:

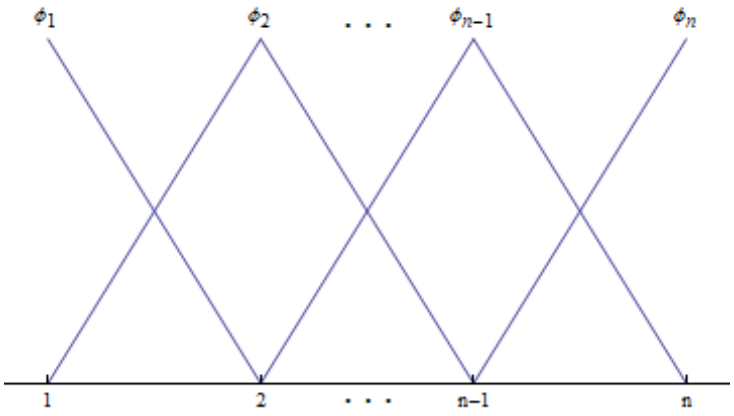


Figure 20: Vertex function DOF numbering on a one dimensional domain.

Once this has been completed, we then add on the higher order basis functions. We do this by numbering all the higher order functions consecutively, on each element in turn. This is depicted in Figure 21 below.

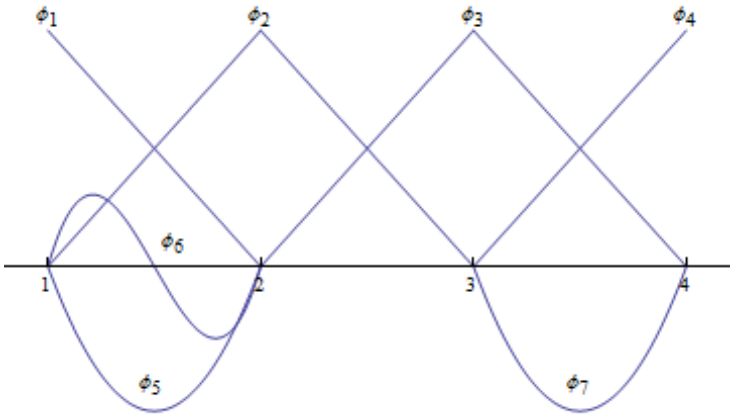


Figure 21: Full DOF numbering on a one dimensional domain.

Now that we have a numbering scheme, we can introduce a storage method for the DOFs (Antonietti et al., 2011). The vertex DOF numbers correspond to the element connectivity arrays, meaning that we only need to call the GetElementConnectivityArray from the Element class for each element to get the vertex DOFs on that element. Hence, no storage at all is required for the vertex DOFs. For the higher order basis DOFs, we introduce a single array called dofStart of length (numElements + 1). This array contains only the numbers of what the first higher order basis function on each individual element is/would be. The final entry in the array is what the next starting number would be if

more DOFs were to be added in.

So, in the case of Figure 21, we would have a dofStart array of length 4, given by

$$\text{dofStart} = [5 \quad 7 \quad 7 \quad 8]. \tag{5.1}$$

This is because on the first element the starting DOF number is 5, on the second if there were to be any basis functions the DOF number would start at 7, and the third element starts at 7 because there are no higher order bases on element 2. As we have one basis function on element 3, this means that the DOF start number for any subsequent elements would begin at number 8. The total number of DOFs can then be calculated to be the last entry in the dofStart array minus 1. This is the length of our solution vector.

Note: In the case of a polynomial degree of 1 across all elements, the dofStart array entries would all be identical. Although it would be redundant here, when working with a number of higher order bases this method of storage can be very memory conserving.

Now, to access the element DOFS, we have the method GetElementDofs. Obviously, we do not have any explicit storage for the DOFs on each element, so we need to construct a vector to return with all the correct DOFS for the specified element.

```
1  Vector FE_Solution::GetElementDofs(int elementNumber)
2  {
3      int connectivitySize = mMeshReference->GetElement(elementNumber)->
           GetElementConnectivityArray().GetSize();
4      Vector elementDofs(connectivitySize + (*dofStart)(elementNumber+1) - (*dofStart)(
           elementNumber));
5
6      for (int i=0; i<connectivitySize; i++)
7      {
8          elementDofs[i] = (mMeshReference->GetElement(elementNumber)->
               GetElementConnectivityArray())[i];
9      }
10     for (int i=connectivitySize; i<elementDofs.GetSize(); i++)
11     {
12         elementDofs[i] = (*dofStart)(elementNumber) + (i-connectivitySize);
13     }
14
15     return elementDofs;
16 }
```

Listing 8: Method to retrieve the degrees of freedom numbers for a specified element.

In Listing 8 above, we show the method that does this. Say we have an element k. We populate the first part of the elementDofs vector with the element connectivity array for k. Following this, we add in the values from the k-th entry in dofStart up to but not including the $(k+1)$-th entry in dofStart. If two consecutive entries in dofStart are the same, then there are no higher order bases on the corresponding element.

Example 5.4.1

Once again, using the example in Figure 21, recall that we have the dofStart array

$$\text{dofStart} = [5 \;\; 7 \;\; 7 \;\; 8]. \tag{5.2}$$

For element 1, we have the connectivity array

$$\text{Connectivity} = [1 \;\; 2]. \tag{5.3}$$

The first entry in the dofStart array is 5 and the 2nd is 7. Hence, we can construct the elementDofs vector as

$$\text{elementDofs} = [1 \;\; 2 \;\; 5 \;\; 6]. \tag{5.4}$$

Similarly, for element 2 we have the elementDofs vector

$$\text{elementDofs} = [2 \;\; 3], \tag{5.5}$$

and for element 3 we have

$$\text{elementDofs} = [3 \;\; 4 \;\; 7]. \tag{5.6}$$

□

5.4.2 Computation of Basis Functions and their Gradients

For this part, we shall introduce the PolynomialSpace class. We can see below, in Figure 22, the UML class diagram for this.

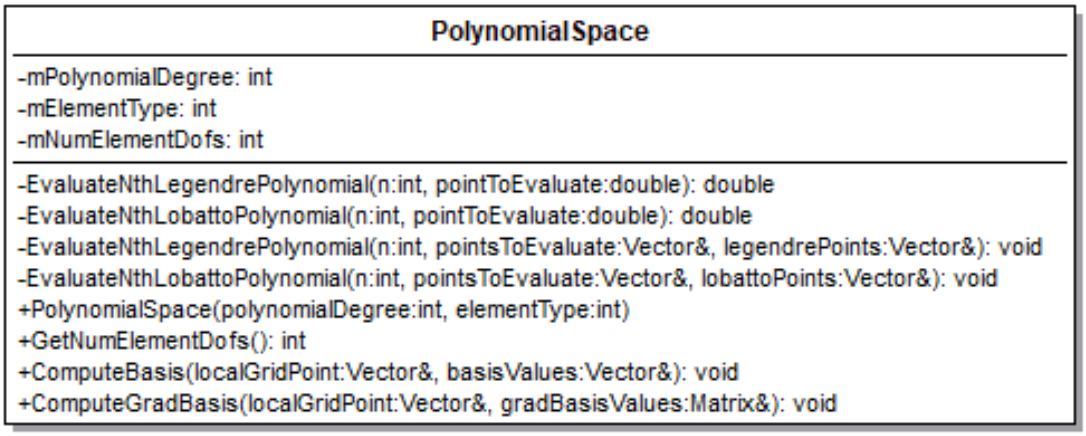


Figure 22: UML representation for the PolynomialSpace class.

The PolynomialSpace class is used to calculate the basis functions and gradients of the basis functions on the reference element. We instantiate a separate In Section 4.4, we detailed the hierarchical basis functions for the three element types. Recall that they are based on the Lobatto shape functions (4.31) which, in turn, are constructed from the Legendre polynomials (4.27). We have implemented methods to compute the values of the nth Legendre and Lobatto functions at a single point and at multiple points in an array. Utilising these, we construct the basis functions and their gradients for each element using a switch statement over the element type. For these methods, see Appendix A.1.

From Figure 19, we can see that there are also methods to compute the basis functions and their gradients in the FE_Solution class. These are the methods that the user will call from outside the class and within them they call the methods from PolynomialSpace. The only difference being that the ComputeGradBasis method then transforms the gradients of the basis functions by multiplying by the inverse transpose of the Jacobian of the mapping from the local to global element, as explained in Section 4.3.4. This is so that integration of the gradient of a basis function can be performed on the reference element.

5.4.3 Computation of the Finite Element Solution u_h

The only other method our class contains is one to compute the finite element solution u_h. This is called once the solution vector U has been calculated by the user. From Section

4.1, we know that the formula for u_h is given by

$$u_h(x) = \sum_{j=1}^{N(h)} U_j \phi_j(x). \tag{5.7}$$

We detail the method below in Listing 9.

```
1  double FE_Solution::ComputeUh(int elementNumber, Vector& localGridPoint)
2  {
3      double Uh = 0;
4
5      Vector* basisValues = new Vector (GetNumElementDofs(elementNumber));
6      ComputeBasis(elementNumber, localGridPoint, *basisValues);
7      Vector* elementDofs = new Vector (GetElementDofs(elementNumber));
8
9      for (int i=1; i<=GetNumElementDofs(elementNumber); i++)
10     {
11         Uh += ((*mSolutionVector)((*elementDofs)(i)))*((*basisValues)(i));
12     }
13
14     delete basisValues;
15     delete elementDofs;
16
17     return Uh;
18 }
```

Listing 9: Method to compute the finite element solution u_h.

5.5 QuadratureLibrary

We use the class QuadratureLibrary to produce everything needed to evaluate integrals numerically. In Section 6, we will look in detail at the theoretical concepts behind quadrature rules and address the implementation. Here we shall briefly introduce the class and its structure. As the class is only a library, we do not need any storage or specialised constructor. The only public method that this class contains is the GetQuadrature method (see Appendix A.5 for full method). This returns the quadrature points and weights for the specified element type. The rest of the private methods can be seen in the UML representation of the class in Figure 23 below.

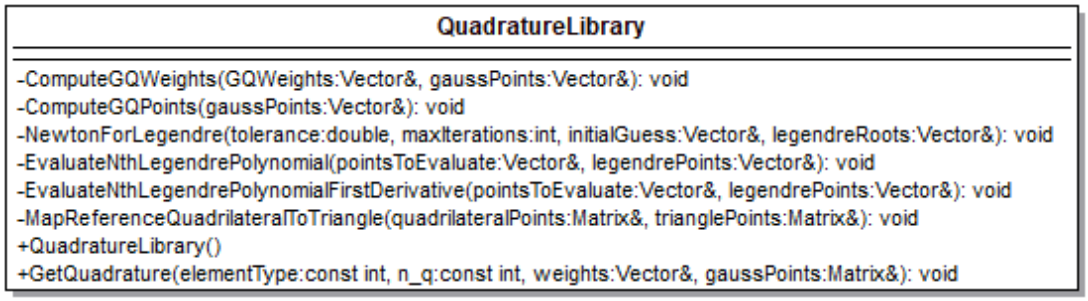


Figure 23: UML representation for the QuadratureLibrary class.

As they are all based on the quadrature theory which we have not yet explained, we shall not go through them here. These methods will be explained in Section 6 and the full code for each can be found either there or in Appendix A.5.

5.6 SparseMatrix

As explained in Section 4.2, once we have formulated our finite element representation of a problem, we have to solve a matrix system of the form

$$AU = F.$$

We will talk more about the assembly of A in Section 7, but we find that in general A contains a large number of zero entries. We could store A as a standard matrix, but when working with very fine meshes and high polynomial degrees, this matrix can become very large. There are a variety of alternative storage methods that can be employed for storing sparse matrices more efficiently (Ketelsen, 2015). For our implementation, we will be using the compressed sparse row (CSR) format. For more on the sparsity patterns and implementation of sparse matrices, see the deal.ii documentation (Bangerth, Hartmann, and Kanschat, 2007).

5.6.1 Compressed Sparse Row

Compressed sparse row works by only storing three one dimensional arrays instead of the full matrix. The first of these arrays stores the non-zero entries in the matrix. The second array works in a similar fashion to the dofStart array that we introduced in Section 5.4.1. It is called the row pointer array and works by indicating at what index in the non-zero

values array the next row of the matrix would begin at. For an $m \times n$ matrix, this array contains $m + 1$ entries. The final array stores the column indices of every entry in the non-zero value array. This, like the non-zero values array, is of length equal to the number of non-zeros. We shall now demonstrate with an example.

Example 5.6.1

Suppose we have the 5×5 matrix

$$M = \begin{pmatrix} 3 & 0 & 9 & 0 & 0 \\ 0 & 1 & 3 & 0 & 7 \\ 0 & 2 & 0 & 0 & 0 \\ 6 & 0 & 0 & 5 & 0 \\ 0 & 0 & 4 & 0 & 0 \end{pmatrix}. \tag{5.8}$$

The number of non zero entries in M is 9. Hence, we can set up our three storage vectors of length 9, 6 and 9 respectively.

$$\text{Values} = [3 \;\; 9 \;\; 1 \;\; 3 \;\; 7 \;\; 2 \;\; 6 \;\; 5 \;\; 4] \tag{5.9}$$

$$\text{Row\_ptr} = [1 \;\; 3 \;\; 6 \;\; 7 \;\; 9 \;\; 10] \tag{5.10}$$

$$\text{Col\_index} = [1 \;\; 3 \;\; 2 \;\; 3 \;\; 5 \;\; 2 \;\; 1 \;\; 4 \;\; 3]. \tag{5.11}$$

The final entry in the Row_ptr array is always one greater than the number of non-zero entries. In order to determine the number of non-zero entries in a given row, we just look at the difference between the consecutive entries in Row_ptr. For example, for the second row in M, we do $6 - 3 = 3$, telling us that there are 3 non-zero entries in row 2. These values correspond to values from the 3rd to the 5th entry in the Values array. □

Compressed sparse row format is only more computationally efficient if we have that

$$\text{Number of non-zeros} < \frac{(m(n-1)-1)}{2}. \tag{5.12}$$

As seen from the matrix M in Example 5.6.1, in the full matrix there are $5 \times 5 = 25$ entries. The total number of entries in the CSR format of M is $9 + 6 + 9 = 24$. We can observe that

$$\text{Number of non-zeros} = 9 < \frac{(5(5-1)-1)}{2} = \frac{19}{2} = 9.5.$$

5.6.2 Implementation of CSR for a Finite Element Stiffness Matrix

Below, in Figure 24, we give the UML representation of our SparseMatrix class. As explained, we have storage for the three vectors that make up the CSR format. Due to our implementation being specifically for a finite element stiffness matrix, our SparseMatrix constructor takes in an argument of type FE_Solution along with the number of elements in the mesh.

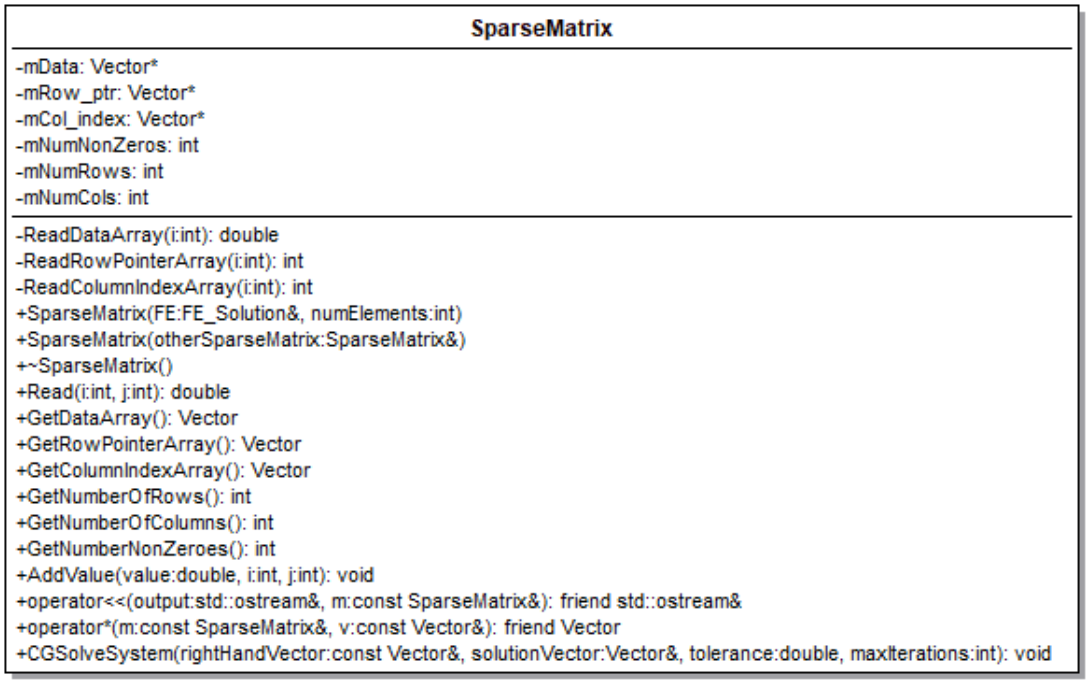


Figure 24: UML representation for the SparseMatrix class.

In order to populate the storage vectors we need to do a "dry run" of the algorithm to determine the sizes of the storage vectors. This works by estimating how many possible non-zero entries there may be in the global stiffness matrix. The full code for the constructor can be found in Appendix A.6 but we will break it down here in small chunks. Algorithm 1 below details the procedure to compute the potential number of non-zero entries that the global stiffness matrix may contain.

Algorithm 1 Compute the potential number of non-zeros in the global stiffness matrix.

Require: An array NNZ of length numberOfDofs.

```
loop over elements
    for each row in the local element stiffness matrix do
        for each column in the local element stiffness matrix do
            NNZ(RowDofNumber) ← NNZ(RowDofNumber) +1
        end for
    end for
end loop
for i = 1 to length(NNZ) do
    NumberNonZeros ← NumberNonZeros + NNZ(i)
end for
```

An array is created to hold the estimated number of non-zeros on each row in the global stiffness matrix. The elements are looped over and on each one the number of non-zeroes for each row in the local element stiffness matrix is computed. Once we have this array populated, the values are summed up to give a final estimation of the total number of non-zeros. This number is always going to be an over-estimation of the actual number of non-zeros due to our algorithm not accounting for duplicated entries.

Now that we have an estimate for the number of non-zeros on each row of the global stiffness matrix, we populate the Row_ptr array using Algorithm 2 below.

Algorithm 2 Populate the Row_ptr array for the global stiffness matrix

Require: The populated NNZ array from Algorithm 1.

```
Row_ptr(1) ← 1
for j = 2 to length(Row_ptr) do
    Row_ptr(j) ← Row_ptr(j − 1) + NNZ(j − 1)
end for
```

The final item to populate is our Col_index array. It is important here that we do not create any duplicate entries. We get around this by introducing a Boolean variable named ColIndexExists. If the appropriate column index has already been inputted then ColIn-

dexExists will be true and so the algorithm leaves the entry as zero and proceeds to the next entry. If not, then the algorithm updates the correct entry in the Col_index array to the appropriate value.

Algorithm 3 Populate Col_index array.

Require: The populated Row_ptr array from Algorithm 2.

Require: A Boolean variable ColIndexExists.

```
1: loop over elements
2:     for each row in the local element stiffness matrix do
3:         for each row in the local element stiffness matrix do
4:             ColIndexExists ← false
5:             for i = Row_ptr(RowDofNumber) to (Row_ptr(RowDofNumber+1)−1) do
6:                 if Col_index(i) = ColDofNumber then
7:                     ColIndexExists ← true
8:                 end if
9:             end for                                  ▷ ColIndexExists check complete
10:            if ColIndexExists = false then
11:                for i = Row_ptr(RowDofNumber) to (Row_ptr(RowDofNumber+1)−1)
   do
12:                    if Col_index(i) = 0 then
13:                        Col_index(i) ← ColDofNumber
14:                        break for loop
15:                    end if
16:                end for
17:            end if                                   ▷ Correct Col_index value set
18:        end for
19:    end for
20: end loop
```

5.6.3 Operator overloading and value inputting

We will need to be able input the values of our global stiffness matrix to our SparseMatrix instance. For this, we use the method AddValue, detailed below in Listing 10.

```
1  void SparseMatrix::AddValue(double value, int i, int j)
2  {
3      assert(i > 0);
4      assert(i < mNumRows+1);
5      assert(j > 0);
6      assert(j < mNumCols+1);
7
8      for (int k=(*mRow_ptr)(i); k<(*mRow_ptr)(i+1); k++)
9      {
10         if((*mCol_index)(k)==j)
11         {
12             (*mData)(k) = value;
13             return;
14         }
15     }
16 }
```

Listing 10: Method to add a value into the SparseMatrix storage format.

The method works by checking to see whether the specified column index matches up to a storage column index for the specified row. If so, then the value is inserted at the relevant entry in the Values array (called mData in our implementation). If the specified column index has not been previously set in the Col_index array then the value is not added in as we have not allocated space for it.

Once we have a fully populated global stiffness matrix, we are going to need to be able to solve the matrix system

$$AU = F$$

to generate our solution vector. There are many solvers that exist for solving such systems of equations. For our implementation, we choose the conjugate gradient method. We will not go into the detail of the method in this dissertation but more information can be found in Quarteroni, Sacco, and Saleri (2010). In this method, the only computation that is required on our matrix is a matrix-vector multiplication. To do this, we will overload the $*$ operator so that a product of type SparseMatrix-Vector can be computed.

```
Vector operator*(const SparseMatrix& m, const Vector& v)
```

The algorithm to compute this is taken from Ketelsen (2015) and is given as follows in Algorithm 4.

Algorithm 4 Compute the matrix-vector product of a matrix stored in CSR format

Require: An $m \times n$ SparseMatrix A and an n-length Vector x.

Require: A new Vector b of length m.

for $i = 1$ to m **do**

$b(i) \leftarrow 0$

for $j = \text{Row\_ptr}(i)$ to $(\text{Row\_ptr}(i+1) - 1)$ **do**

$b(i) \leftarrow b(i) + \text{Values}(j) * x(\text{Col\_index}(j))$

end for

end for

In our implementation, we also overload the insertion operator << so that we can output the full matrix if necessary. All row and column indices are looped over and a call to read the value from the CSR storage is made. This works in a very similar way to the AddTo method but instead of assignment, the value at the specified entry is returned. If there does not exist a value at the indices given, then 0 is returned. The values are outputted in scientific format with a precision of five decimal places, in the same way as the Matrix class from Pitt-Francis and Whiteley (2012).

6 Numerical Quadrature on Elements

When solving simple partial differential equations using the finite element method by hand, it may be so that the integrals in the weak formulation can be solved analytically. More often than not, it is such that the integrals require a numerical approach to achieve a solution. Secondly, for the purposes of implementation, unless the antiderivatives of the integrals are known before hand and have been inputted, we will *have* to employ a numerical technique to compute the integrals.

A quadrature rule is a form of approximation for a definite integral of a function. Quadrature rules work by taking a weighted sum of functions points evaluated at a specific set of abscissas (points) across the domain of integration (Epperson, 2013). Conventionally, the domain of integration for the rule itself is taken to be $[-1, 1]$.

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i)$$

In order to numerically compute integrals on a general domain, we must map from the general domain onto the reference domain; similarly to the mapping of elements in Section 4.3.

6.1 Gauss-Legendre Quadrature

There are many quadrature rules which require different classes of orthogonal polynomials, such as Gauss-Jacobi quadrature, Chebyshev-Gauss quadrature and Newton-Cotes quadrature (Quarteroni, Sacco, and Saleri, 2010). For our implementation we shall be using one of the most common quadrature rules, Gauss-Legendre quadrature. As can be inferred from the name, the rule requires the use of the Legendre polynomials. We met these in Section 4.4.1 when we were introducing the hierarchical basis functions for each element type.

To recap, they are produced by using the following recursive formula:

$$\begin{aligned} L_0(x) &= 1, \\ L_1(x) &= x, \\ L_{n+1}(x) &= \frac{2n+1}{n+1} x L_n(x) - \frac{n}{n+1} L_{n-1}(x), \quad n \geq 1. \end{aligned} \tag{6.1}$$

Similarly, their derivatives can also be generated with a recursive formula:

$$\begin{aligned} L'_0(x) &= 0, \\ L'_1(x) &= 1, \\ L'_{n+1}(x) &= \frac{2n+1}{n} x L'_n(x) - \frac{n+1}{n} L'_{n-1}(x), \quad n \geq 1. \end{aligned} \tag{6.2}$$

The quadrature points (abscissas) x_i, $i = 1, \ldots, n$ for the nth order quadrature rule are given by the roots of the nth order Legendre polynomial $L_n(x)$. Hence we need to be able to solve the equation

$$L_n(x) = 0 \tag{6.3}$$

for x. To do this we employ Newton's method, (Epperson, 2013). In our implementation, an initial guess of the Chebyshev nodes (Cheney and Kincaid, 2012) was used in the Newton iteration, those of which are given by

$$x_i^{(0)} = -\cos\left(\frac{2i-1}{2n}\pi\right), \tag{6.4}$$

for $i = 1, \ldots, n$.

The weights w_i, $i = 1, \ldots, n$ for the Gauss-Legendre quadrature rule are defined by

$$w_i = \frac{2}{(1-x_i^2)[L'_n(x_i)]^2}, \tag{6.5}$$

for $i = 1, \ldots, n$.

Below, we can observe the design of the C++ methods used to obtain the quadrature points and weights (for the NewtonForLegendre method see Appendix A.5). A blank vector of length n, which will hold the quadrature points, is passed in to the ComputeGQ-Points function. The initial guess is defined and then the points are iterated upon until the 2-norm of the difference between the previous guess and the current guess is below the specified tolerance, or the amount of iterations exceeds the stated maximum.

```
1  void QuadratureLibrary::ComputeGQPoints(Vector& gaussPoints)
2  {
3      double tolerance = 1e-12;
4      int maxIterations = 1000;
5      Vector* initialGuess = new Vector (gaussPoints.GetSize());
6
7      for (int i=1; i<=gaussPoints.GetSize(); i++)
```

```
8       {
9           (*initialGuess)[i-1] = -cos((((2.0*i)-1.0)/(2.0*gaussPoints.GetSize()))*Pi);
10      }
11
12      NewtonForLegendre(tolerance, maxIterations, (*initialGuess), gaussPoints);
13      delete initialGuess;
14  }
```

Listing 11: Function to compute Gauss quadrature points for a vector of length n.

Once the quadrature points have been calculated, they are then passed into the method for computing the quadrature weights, along with a blank vector to hold the weights. The derivative of the nth Legendre polynomial is then calculated at the points and finally the formula for the weights (6.5) is evaluated.

```
1   void QuadratureLibrary::ComputeGQWeights(Vector& GQWeights, Vector& gaussPoints)
2   {
3       assert(GQWeights.GetSize() == gaussPoints.GetSize());
4
5       Vector* legendreDeriv = new Vector(GQWeights.GetSize());
6
7       EvaluateNthLegendrePolynomialFirstDerivative(gaussPoints, (*legendreDeriv));
8
9       for (int i=0; i<GQWeights.GetSize(); i++)
10      {
11         GQWeights[i] = 2.0/((1.0-pow(gaussPoints[i],2.0))*(pow((*legendreDeriv)[i],2.0)));
12      }
13
14      delete legendreDeriv;
15  }
```

Listing 12: Function to compute Gauss quadrature weights for a vector of length n.

Now that the points and weights have been defined for the Gauss-Legendre quadrature rule, we can show how they are used to approximate integrals on the three element types we are focusing on in our implementation. A detailed explanation of the formulation of the Gauss-Legendre quadrature rule on these elements can be found in Deng (2010).

6.2 Quadrature on Intervals

For the standard quadrature interval $[-1, 1]$ the quadrature rule is given by

$$\int_{-1}^{1} g(\xi)dx \approx \sum_{i=1}^{N} w_i g(\xi_i). \tag{6.6}$$

When extending this to a general interval $[a, b]$, we need to employ the mapping (4.18) introduced in Section 4.3.1, given by

$$x = a + \frac{b-a}{2}(\xi + 1). \tag{6.7}$$

The only other thing we require in order to change our variable of integration is the Jacobian of the mapping defined by $J_{ij} = \frac{\partial x_i}{\partial \xi_j}$, which in this case is

$$J(\xi) = \left(\frac{b-a}{2}\right). \tag{6.8}$$

Now, we have, for a function $G(x)$:

$$\int_a^b G(x)dx = \int_{-1}^{1} G\left(x(\xi)\right) |J(\xi)| \, d\xi = \frac{b-a}{2} \int_{-1}^{1} G\left(a + \frac{b-a}{2}(\xi+1)\right) d\xi. \tag{6.9}$$

Therefore, we can write

$$\int_a^b G(x)dx \approx \frac{b-a}{2} \sum_{i=1}^{N} w_i G\left(a + \frac{b-a}{2}(\xi_i + 1)\right). \tag{6.10}$$

This is the Gauss-Legendre quadrature rule for a general interval $[a, b]$. It is exact for polynomials of degree $2N - 1$ (Dow, 1998).

As mentioned in Section 4.3, the design of our reference elements and their basis functions was chosen so that it would make numerical integration on the elements simpler. If we were to compute the basis functions on the global elements instead of just the reference element, then in order to perform the integration we would have to map to the reference element anyway. Hence, we are essentially cutting down on computations by using a reference element. The only thing that is required to perform the integration of a basis function on an element is the local basis function and the determinant of the mapping Jacobian, which is stored in our Element class. Therefore, we perform quadrature of a basis function ϕ on our global element by using

$$\int_I \phi(x)dx = \int_{\hat{I}} \varphi(\xi) \, |J(\xi)| \, d\xi \approx \frac{b-a}{2} \sum_{i=1}^{N} w_i \varphi(\xi_i), \tag{6.11}$$

where φ is the corresponding local basis function to ϕ.

6.3 Quadrature on Quadrilaterals

We shall first look at quadrature on the local quadrilateral defined on $[-1,1]^2$. Say, for a function $g(\xi,\eta)$, we wish to integrate on this domain i.e.

$$\int_{-1}^{1}\int_{-1}^{1} g(\xi,\eta)d\xi d\eta. \tag{6.12}$$

If we apply the quadrature rule to the inner integral we get

$$\int_{-1}^{1}\int_{-1}^{1} g(\xi,\eta)d\xi d\eta \approx \int_{-1}^{1}\left(\sum_{i=1}^{M} w_i g(\xi_i,\eta)\right)d\eta. \tag{6.13}$$

Next, employing numerical quadrature with respect to η we achieve the result

$$\int_{-1}^{1}\int_{-1}^{1} g(\xi,\eta)d\xi d\eta \approx \sum_{i=1}^{M}\sum_{j=1}^{N} w_i\hat{w}_j g(\xi_i,\eta_j), \tag{6.14}$$

where η_j and $\hat{w}_j$ are the Nth order Gauss-Legendre quadrature abscissas and weights in the η direction. Although M and N can be chosen separately, they are normally taken so that $M=N$, giving the rule

$$\int_{-1}^{1}\int_{-1}^{1} g(\xi,\eta)d\xi d\eta \approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j g(\xi_i,\xi_j). \tag{6.15}$$

Now, we need to be able to perform integration on a general quadrilateral Q. For a function $G(x,y)$, we require the evaluation of

$$\iint_{Q} G(x,y)dxdy. \tag{6.16}$$

To do this, we use of the mapping from the local element to the global element introduced in Section 4.3.2:

$$\boldsymbol{x}(\xi,\eta) = \frac{(1-\xi)(1-\eta)\boldsymbol{x_1} + (1+\xi)(1-\eta)\boldsymbol{x_2} + (1+\xi)(1+\eta)\boldsymbol{x_3} + (1-\xi)(1+\eta)\boldsymbol{x_4}}{4}, \tag{6.17}$$

where $\boldsymbol{x}(\xi,\eta) = \begin{pmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{pmatrix}$ are the global coordinates corresponding to (ξ,η) and $\boldsymbol{x_1},\ldots,\boldsymbol{x_4}$ are the coordinates of the nodes of the global quadrilateral. Similarly to the interval element, we need the determinant of the mapping Jacobian:

$$|J(\xi,\eta)| = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix} \tag{6.18}$$

Hence, we can formulate the rule for a general quadrilateral as

$$\begin{aligned}\iint_Q G(x,y)dxdy &= \iint_{\hat{Q}} G(x(\xi,\eta),y(\xi,\eta))\left|J(\xi,\eta)\right| d\xi d\eta \\ &\approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j G(x(\xi_i,\xi_j),y(\xi_i,\xi_j))\left|J(\xi_i,\xi_j)\right|.\end{aligned} \tag{6.19}$$

As discussed in the previous section (6.2), the computations can be cut down by our use of the reference element Q. Therefore, we can perform quadrature on a basis function ϕ on Q by using

$$\iint_Q \phi(x,y)dxdy = \iint_{\hat{Q}} \varphi(\xi,\eta)\left|J(\xi,\eta)\right| d\xi d\eta \approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \varphi(\xi_i,\xi_j)\left|J(\xi_i,\xi_j)\right|. \tag{6.20}$$

6.4 Quadrature on Triangles

For the triangle element we are going to be using a tensor-product type Gauss quadrature rule; the principles of which can be found in Mousavi and Sukumar (2010), an extension on the findings of Duffy (1982).

In conjunction with the interval and quadrilateral elements, we shall start by demonstrating quadrature on the reference triangle $\hat{T}$ as depicted in Figure 7, defined by

$$\hat{T} = \{(\xi,\eta)\in\mathbb{R}^2 : \quad -1 < \xi,\eta \quad \text{and} \quad \xi+\eta < 0\}. \tag{6.21}$$

The tensor-product type quadrature works by first mapping this reference triangle onto the reference quadrilateral $\hat{Q}$ and then performing the numerical integration. We introduce the mapping

$$\boldsymbol{\xi}(a,b) = \begin{pmatrix}\xi(a,b)\\ \eta(a,b)\end{pmatrix} = \begin{pmatrix}-\frac{1}{2}+\frac{1}{2}a-\frac{1}{2}b-\frac{1}{2}ab\\ b\end{pmatrix}, \tag{6.22}$$

and its Jacobian

$$J(a,b) = \begin{pmatrix}\frac{1-b}{2} & -\frac{1+a}{2}\\ 0 & 1\end{pmatrix}. \tag{6.23}$$

Taking the determinant of the mapping Jacobian gives

$$|J(a,b)| = \begin{vmatrix}\frac{1-b}{2} & -\frac{1+a}{2}\\ 0 & 1\end{vmatrix} = \frac{1-b}{2}, \tag{6.24}$$

which is what we need to map the domain of integration. Hence, for a function $g(\xi, \eta)$ we have the integral

$$\int_{-1}^{1}\int_{-1}^{-\eta} g(\xi,\eta)d\xi d\eta = \int_{-1}^{1}\int_{-1}^{-\xi} g(\xi,\eta)d\eta d\xi = \int_{-1}^{1}\int_{-1}^{1} g(\xi(a,b),\eta(a,b))\,|J(a,b)|\,dadb. \tag{6.25}$$

If we let $G(a,b) = (g(\xi(a,b),\eta(a,b))\,|J(a,b)|)$, then we can write this as

$$\int_{-1}^{1}\int_{-1}^{1} G(a,b)dadb. \tag{6.26}$$

Now, employing the quadrature rule for the reference quadrilateral we achieve the result

$$\iint_{\hat{T}} g(\xi,\eta)d\xi d\eta = \iint_{\hat{Q}} G(a,b)dadb \approx \sum_{i=1}^{N}\sum_{i=1}^{N} w_i w_j G(a_i,a_j). \tag{6.27}$$

For a general triangle T all that is required is to map to the local triangle $\hat{T}$ then employ the same rule as above in (6.27). Once again, say we have a function $f(x,y)$ and we wish to evaluate

$$\iint_{T} f(x,y)dxdy. \tag{6.28}$$

In Section 4.3.3 we gave the mapping for the reference element to the global element:

$$\boldsymbol{x} = -\left(\frac{\xi+\eta}{2}\right)\boldsymbol{x_1} + \left(\frac{\xi+1}{2}\right)\boldsymbol{x_2} + \left(\frac{\eta+1}{2}\right)\boldsymbol{x_3}. \tag{6.29}$$

With the determinant of the mapping Jacobian

$$|J(\xi,\eta)| = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix}, \tag{6.30}$$

we can change the domain of integration as follows:

$$\iint_{T} f(x,y)dxdy = \iint_{\hat{T}} f(x(\xi,\eta),y(\xi,\eta))\,|J(\xi,\eta)|\,d\xi d\eta. \tag{6.31}$$

Now, if we let $F(\xi,\eta) = f(x(\xi,\eta),y(\xi,\eta))\,|J(\xi,\eta)|$, we can apply the mapping to the reference quadrilateral and use the rule (6.27) we established. This gives us the final result

$$\iint_{T} f(x,y)dxdy = \iint_{\hat{T}} F(\xi,\eta)d\xi d\eta = \iint_{\hat{Q}} \hat{F}(a,b)dadb \approx \sum_{i=1}^{N}\sum_{i=1}^{N} w_i w_j \hat{F}(a_i,a_j), \tag{6.32}$$

where $\hat{F}(a,b) = (F(\xi(a,b),\eta(a,b))\,|J(a,b)|)$.

7 Numerical Solution to Finite Element Formulations

Now that we have covered the theory and implementation of the finite element method, we will show how our implementation can be used to solve partial differential equations in both one and two dimensions. This will be demonstrated using several examples, in which we will combine all aspects of our finite element method knowledge, along with our code, to give a thorough theoretical and practical evaluation. Once we have generated a solution to the problems, we shall do an a posteriori error analysis to show the effect that changing our mesh size h and polynomial degree p has on the convergence of our solution.

7.1 One Dimensional Poisson Equation

For the first problem, we are going to look at the one dimensional Poisson equation on $x \in [0, 1]$, of the form:

$$-u''(x) = -2\pi^2 \cos(2\pi x), \quad x \in (0, 1), \tag{7.1}$$

$$u(0) = u(1) = 0. \tag{7.2}$$

This has analytic solution

$$u(x) = \sin^2(\pi x). \tag{7.3}$$

In Example 3.1.2 and 3.2.2 in Section 3, we gave the weak formulation of the general Poisson equation and proved that a unique solution exists by using the Lax-Milgram theorem (3.2.1). The weak formulation for (7.1) is given by

Find $u \in H_0^1(0, 1)$ such that

$$\int_0^1 u'(x)v'(x)dx = \int_0^1 -2\pi^2 \cos(2\pi x)v(x)dx, \tag{7.4}$$

for all $v(x) \in H_0^1(0, 1)$.

We now need to construct a mesh τ_h on our domain. For this example, we will be using a uniform mesh with step size h. Hence, we can generate our mesh with Algorithm 5 below.

Algorithm 5 Uniform mesh generation on an interval $[a, b]$ with step size h.

Require: numGridPoints $\leftarrow \frac{b-a}{h} + 1$

  for $i = 1$ to numGridPoints **do**

    gridPoint$(i) \leftarrow h(b-a)(i-1)$

  end for

Require: numElements $\leftarrow \frac{b-a}{h}$

  for $j = 1$ to numElements **do**

    for $k = 1$ to numNodesPerElement (2) **do**

      Connectivity$(j, k) \leftarrow j + (k-1)$

    end for

  end for

Let us start with a step size of $h = 0.08\dot{3}$, which corresponds to a mesh with 12 elements and 13 grid points. In our code, we use a *dimension* $\times$ *numGridPoints* matrix named Grid to hold the grid points, so in this case Grid is 1×13. We run Algorithm 5 to generate the grid and then instantiate a Mesh object called myMesh.

```
Mesh* myMesh = new Mesh(*Grid, numElements);
```

The next step is to initialise each element in the mesh by using the InitialiseElement method from the Mesh class (as explained in Section 5.2). This is demonstrated below in Listing 13.

```
1  for (int i=1; i<=Connectivity->GetNumberOfRows(); i++)
2  {
3      Vector* Con_temp = new Vector (Connectivity->GetRowAsVector(i));
4      myMesh->InitialiseElement(i, *Con_temp, 0);
5      delete Con_temp;
6  }
```

Listing 13: Initialisation of Interval elements on domain.

Now that we have a mesh τ_h, we can formulate our finite element space $V_h \in H_0^1(0,1)$. In this example, we are going to be using a polynomial degree of $p = 2$ for all elements. Using the definition of the finite element space from Section 4.1.3, we define V_h to be

$$V_h = \{v \in H_0^1(0,1) : \quad v|_K \in P_2(K), \quad \forall K \in \tau_h\}, \tag{7.5}$$

where $P_2(K)$ is the space of quadratic polynomials associated with each K. Hence, our finite element formulation of the problem can be stated as

Find $u_h \in V_h$ such that

$$\int_0^1 u_h'(x)v_h'(x)dx = \int_0^1 -2\pi^2 \cos(2\pi x)v_h(x)dx, \tag{7.6}$$

for all $v_h(x) \in V_h$.

Utilising the fact that $V_h = \text{span}\{\phi_1, \phi_2, \ldots, \phi_{25}\}$, we set up our matrix system $AU = F$, where A has entries

$$A_{ij} = \int_0^1 \phi_i'(x)\phi_j'(x)dx, \quad i, j = 1, 2, \ldots, 25, \tag{7.7}$$

and F has entries

$$F_i = \int_0^1 -2\pi^2 \cos(2\pi x)\phi_i(x)dx, \quad i = 1, 2, \ldots, 25. \tag{7.8}$$

Next, we need to instantiate our finite element space, FE_Solution, along with our global stiffness matrix A and load vector F. As discussed in Section 5.6, our A matrix is of type SparseMatrix, which uses the FE_Solution object to estimate where the non-zero entries of A may lie.

```
FE_Solution* FE = new FE_Solution(*myMesh, polynomialDegree);
SparseMatrix* A = new SparseMatrix(*FE, numElements);
Vector* F = new Vector (FE->GetNumberOfDofs());
```

We now come on to the assembly of our matrix-vector system. Section 4.2 showed that we can write the integrals over the whole domain, as the sum of integrals across the elements.

$$A_{ij} = \int_0^1 \phi_i'(x)\phi_j'(x)dx = \sum_K \int_K \phi_i'(x)\phi_j'(x)dx, \quad i, j = 1, 2, \ldots, 25, \tag{7.9}$$

$$F_i = \int_0^1 -2\pi^2 \cos(2\pi x)\phi_i(x)dx = \sum_K \int_K -2\pi^2 \cos(2\pi x)\phi_i(x)dx, \quad i = 1, 2, \ldots, 25. \tag{7.10}$$

On each element, we can then compute the local element stiffness matrix and load vector. As we are working with $p = 2$, there are only three basis functions on each one. Hence, we can write the entries of the local element stiffness matrix A^K as

$$A^K_{lm} = \int_K \phi_l'(x)\phi_m'(x)dx = \int_{x_{k-1}}^{x_k} \phi_l'(x)\phi_m'(x)dx, \quad l, m = 1, 2, 3, \tag{7.11}$$

where x_{k-1} and x_k are the nodes of the corresponding element. We can then map this integral to the local element using the transformations discussed in Section 4.3.4 and Section 6.2.

$$\int_{x_{k-1}}^{x_k} \phi_l'(x)\phi_m'(x)dx = \int_{\hat{I}} \left(J^{-T}\varphi_l'(\xi)\right)\left(J^{-T}\varphi_m'(\xi)\right)|J|\,d\xi, \quad l,m = 1,2,3. \tag{7.12}$$

The Jacobian of the mapping, in this case, is simply $J = \left(\frac{x_k - x_{k-1}}{2}\right)$. Therefore, we have

$$A_{lm}^K = \frac{2}{x_k - x_{k-1}} \int_{-1}^{1} \varphi_l'(\xi)\varphi_m'(\xi)d\xi, \quad l,m = 1,2,3. \tag{7.13}$$

Similarly, the local load vector F^K is populated with entries given by

$$\begin{aligned} F_l^K &= \int_K -2\pi^2 \cos(2\pi x)\phi_l(x)dx \\ &= \int_{\hat{I}} -2\pi^2 \cos\left(2\pi\left(x_{k-1} + \frac{x_k - x_{k-1}}{2}(\xi+1)\right)\right)\phi_l(\xi)\,|J|\,d\xi \\ &= -\pi^2(x_k - x_{k-1})\int_{-1}^{1} \cos\left(2\pi\left(x_{k-1} + \frac{x_k - x_{k-1}}{2}(\xi+1)\right)\right)\varphi_l(\xi)d\xi, \quad l = 1,2,3 \end{aligned} \tag{7.14}$$

For a uniform grid, we have that $h = x_k - x_{k-1}$. Hence, we can write these both as

$$A_{lm}^K = \frac{2}{h}\int_{-1}^{1} \varphi_l'(\xi)\varphi_m'(\xi)d\xi, \quad l,m = 1,2,3, \tag{7.15}$$

$$F_l^K = -\pi^2 h \int_{-1}^{1} \cos\left(2\pi\left(x_{k-1} + \frac{h}{2}(\xi+1)\right)\right)\varphi_l(\xi)d\xi, \quad l = 1,2,3. \tag{7.16}$$

As detailed in Section 4.4, the local basis functions we shall be using are the first three Lobatto shape functions, given by

$$\varphi_1(\xi) = \varphi_{\hat{I}}^{v_1}(\xi) = \frac{1-\xi}{2}, \quad \varphi_2(\xi) = \varphi_{\hat{I}}^{v_2}(\xi) = \frac{1+\xi}{2}, \quad \varphi_3(\xi) = \varphi_{\hat{I}}^{b_2}(\xi) = \frac{1}{2}\sqrt{\frac{3}{2}}(\xi^2 - 1). \tag{7.17}$$

We also require their derivatives, given by

$$\varphi_1'(\xi) = -\frac{1}{2}, \quad \varphi_2'(\xi) = \frac{1}{2}, \quad \varphi_3'(\xi) = \sqrt{\frac{3}{2}}\xi. \tag{7.18}$$

Now, although we can evaluate these integrals analytically, in our implementation this is not possible, therefore we have to employ the quadrature rules discussed in Section 6.2. For this example, we choose a rule of order $n_q = 5$. We loop over each element in turn

and compute the entries to the local element stiffness matrix and load vector using the quadrature rule. Once this is complete, the values of the local element stiffness matrix are added the values to the global stiffness matrix. We shall show the general code for the kth element. The first part is to get the quadrature for the specific element and instantiate our local matrix and vector. This can be seen in Listing 14 below.

```
1    int n_q = 5;
2
3    int numElementDofs;
4
5    Vector* quadratureWeights = new Vector (n_q);
6    Matrix* localQuadraturePoints = new Matrix (1,n_q);
7    Matrix* globalQuadraturePoints = new Matrix (1,n_q);
8
9    numElementDofs = FE->GetNumElementDofs(k);
10   Matrix* A_loc = new Matrix (numElementDofs, numElementDofs);
11   Vector* f_loc = new Vector (numElementDofs);
12
13   myMesh->GetElement(k)->GetQuadrature(n_q, *quadratureWeights, *localQuadraturePoints,
         *globalQuadraturePoints);
```

Listing 14: Set up of the quadrature rule, local element stiffness matrix and load vector, on the kth element.

With the quadrature points and weights now returned, we loop over each point and compute the function value at that point. This is then multiplied by the corresponding weight and added to the weighted sum. To evaluate the basis functions and their derivatives at the appropriate quadrature point, we use our methods from the FE_Solution class, detailed in Section 5.4.2. As the code for this part is a little lengthy, we only include the key lines in Listing 15 below. For the full code see Appendix A.7.

```
14   for (int q=1; q<=n_q; q++)
15   {
16       FE->ComputeBasis(k, (*localQuadraturePoints)(1,q), *basisValues);
17       FE->ComputeGradBasis(k, (*localQuadraturePoints)(1,q), *basisGrad);
18
19       for (int i=1; i<=A_loc->GetNumberOfRows(); i++)
20       {
21           for (int j=1; j<=A_loc->GetNumberOfColumns(); j++)
22           {
23               (*A_loc)(i,j) += (*basisGrad)(1,i)*(*basisGrad)(1,j)*(*quadratureWeights)
                     (q);
```

```
24              }
25          }
26          for (int i=1; i<=f_loc->GetSize(); i++)
27          {
28              (*f_loc)(i) += (*basisValues)(i)*(*quadratureWeights)(q) * (-2.0*pow(Pi,2.0)*
                    cos(2.0*Pi*(*globalQuadraturePoints)(1,q)));
29          }
30      }
```

Listing 15: Evaluation of the local stiffness matrix and load vector entries using numerical quadrature, for the kth element.

We shall now demonstrate the compilation of the local stiffness matrix and load vector, analytically. Let us choose the 1st element in our mesh to show this. The nodes of this element are given by $x_{k-1} = 0$ and $x_k = 0.08\dot{3}$. Hence, we have

$$A^1_{lm} = \frac{2}{h}\int_{-1}^{1} \varphi'_l(\xi)\varphi'_m(\xi)d\xi, \quad l,m = 1,2,3, \tag{7.19}$$

$$F^1_l = -\pi^2 h \int_{-1}^{1} \cos\left(\pi h(\xi+1)\right)\varphi_l(\xi)d\xi, \quad l = 1,2,3. \tag{7.20}$$

So,

$$\begin{aligned}
A^1_{11} &= \frac{2}{h}\int_{-1}^{1}\left(-\frac{1}{2}\right)\left(-\frac{1}{2}\right)d\xi = \frac{2}{h}\int_{-1}^{1}\frac{1}{4}d\xi = \frac{2}{h}\left[\frac{1}{4}\xi\right]_{-1}^{1} = \frac{1}{h},\\
A^1_{12} &= \frac{2}{h}\int_{-1}^{1}\left(-\frac{1}{2}\right)\left(\frac{1}{2}\right)d\xi = \frac{2}{h}\int_{-1}^{1}-\frac{1}{4}d\xi = \frac{2}{h}\left[-\frac{1}{4}\xi\right]_{-1}^{1} = -\frac{1}{h},\\
A^1_{13} &= \frac{2}{h}\int_{-1}^{1}\left(-\frac{1}{2}\right)\left(\sqrt{\frac{3}{2}}\xi\right)d\xi = \frac{2}{h}\int_{-1}^{1}-\frac{1}{2}\sqrt{\frac{3}{2}}\xi d\xi = \frac{2}{h}\left[-\frac{1}{4}\sqrt{\frac{3}{2}}\xi^2\right]_{-1}^{1} = 0,\\
&\vdots
\end{aligned} \tag{7.21}$$

leading to the local element stiffness matrix

$$A^1 = \frac{1}{h}\begin{pmatrix} 1 & -1 & 0\\ -1 & 1 & 0\\ 0 & 0 & 2\end{pmatrix} = \begin{pmatrix} 12 & -12 & 0\\ -12 & 12 & 0\\ 0 & 0 & 24\end{pmatrix}. \tag{7.22}$$

Similarly, for the load vector we have

$$F_1^1 = -\pi^2 h \int_{-1}^{1} \cos\left(\pi h(\xi+1)\right)\left(\frac{1-\xi}{2}\right) d\xi = -0.803848,$$
$$F_2^1 = -\pi^2 h \int_{-1}^{1} \cos\left(\pi h(\xi+1)\right)\left(\frac{1+\xi}{2}\right) d\xi = -0.766949, \tag{7.23}$$
$$F_3^1 = -\pi^2 h \int_{-1}^{1} \cos\left(\pi h(\xi+1)\right)\left(\frac{1}{2}\sqrt{\frac{3}{2}}(\xi^2-1)\right) d\xi = 0.644224.$$

Now that we have our local element stiffness matrix and load vector, we need to add in the values to the corresponding entries in the the global stiffness matrix and load vector. In order to do this, we require the DOF numbers for the element we are working on. The procedure for attaining these was explained in Section 5.4.1.

The entries in the local element stiffness matrices correspond to the entries in the global stiffness matrix by

$$A_{lm}^K \xrightarrow{+} A_{\mathrm{DOF}_l^K, \mathrm{DOF}_m^K} \tag{7.24}$$

for all K. Again, using element 1 in our mesh, we give the DOF numbers for this element by

$$\mathrm{DOF}^1 = [1 \;\; 2 \;\; 14]. \tag{7.25}$$

Hence, we can add the entries of A^1 into the corresponding entries of A in the following manner:

$$\begin{pmatrix} A_{1,1}^1 & A_{1,2}^1 & A_{1,3}^1 \\ A_{2,1}^1 & A_{2,2}^1 & A_{2,3}^1 \\ A_{3,1}^1 & A_{3,2}^1 & A_{3,3}^1 \end{pmatrix} \xrightarrow{+} \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,14} \\ A_{2,1} & A_{2,2} & A_{2,14} \\ A_{14,1} & A_{14,2} & A_{14,14} \end{pmatrix}. \tag{7.26}$$

Likewise, the local load vector entries are added in to the global load vector by

$$F_l^K \xrightarrow{+} F_{\mathrm{DOF}_l^K}. \tag{7.27}$$

So, for the first element in our mesh,

$$\begin{pmatrix} F_1^1 \\ F_2^1 \\ F_3^1 \end{pmatrix} \xrightarrow{+} \begin{pmatrix} F_1 \\ F_2 \\ F_{14}. \end{pmatrix} \tag{7.28}$$

In our implementation, we use the AddValue method (explained in Section 5.6) to populate our global stiffness matrix with the corresponding values. The code for this and the population of the global load vector is given in Listing 16 below.

```
    for (int i=1; i<=A_loc->GetNumberOfRows(); i++)
    {
        for (int j=1; j<=A_loc->GetNumberOfColumns(); j++)
        {
            A->AddValue(A->Read((FE->GetElementDofs(k))(i), (FE->GetElementDofs(k))(j)) +
                 (*A_loc)(i,j), (FE->GetElementDofs(k))(i), (FE->GetElementDofs(k))(j));
        }
    }
    for (int i=1; i<=f_loc->GetSize(); i++)
    {
        (*F)((FE->GetElementDofs(k))(i)) += (*f_loc)(i);
    }
```

Listing 16: Population of global stiffness matrix and load vector for the kth element.

The last step required, before solving the system, is applying the boundary conditions (BCs) of the problem (7.2). In this case, we have that $u(0) = u(1) = 0$. As explained in Section 4.2, we zero the first and thirteenth row and column in A, set the entries $A_{11} = A_{13,13} = 1$ and set $F_1 = F_{13} = 0$. In our implementation, we do this as follows in Listing 17.

```
    for (int i=1; i<=A->GetNumberOfRows(); i++)
    {
        for (int j=1; j<=A->GetNumberOfColumns(); j++)
        {
            if (i==1 || i==myMesh->GetNumNodes() || j==1 || j==myMesh->GetNumNodes())
            {
                if (i==j)
                {
                    A->AddValue(1, i, j);
                }
                else
                {
                    A->AddValue(0, i, j);
                }
            }
        }
    }
    for (int i=1; i<=F->GetSize(); i++)
    {
        if (i==1 || i==myMesh->GetNumNodes())
        {
            (*F)(i) = 0;
        }
```

```
24      }
```

Listing 17: Application of homogeneous Dirichlet boundary conditions.

Hence, we have our final system of equations. The global stiffness matrix for the general Poisson equation with homogeneous Dirichlet boundary conditions is of the form

$$A = \frac{1}{h}\begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 & & & & \\ 0 & 2 & -1 & & \ddots & \vdots & & & & \\ \vdots & -1 & \ddots & \ddots & & \vdots & & 0 & & \\ \vdots & & \ddots & 2 & -1 & \vdots & & & & \\ \vdots & \ddots & & -1 & 2 & 0 & & & & \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & 0 & & & \\ & & & & & 0 & 2 & 0 & \cdots & 0 \\ & & 0 & & & & 0 & \ddots & \ddots & \vdots \\ & & & & & & \vdots & \ddots & \ddots & 0 \\ & & & & & & 0 & \cdots & 0 & 2 \end{pmatrix}. \tag{7.29}$$

We solve the system in our implementation using the method CGSolveSystem. This was briefly discussed in Section 5.6.3.

```
A->CGSolveSystem(*F, FE->GetSolutionVector(), 1e-9, 1000);
```

Here, we specify a convergence tolerance of $1e-9$ and a maximum iteration count of 1000. As our solution vector U is of length 25, we won't specify the whole vector in one. The nodal solution points are the first 13 entries in U and the higher order basis solution points are the final 12. Hence, we will give U as these two vectors separately, i.e.

$$U = \begin{pmatrix} U_{\text{nodal}} \\ U_{\text{higher}} \end{pmatrix}, \quad \text{where} \quad U_{\text{nodal}} = \begin{pmatrix} 0.00000e+000 \\ 6.69873e-002 \\ 2.50000e-001 \\ 5.00000e-001 \\ 7.50000e-001 \\ 9.33013e-001 \\ 1.00000e+000 \\ 9.33013e-001 \\ 7.50000e-001 \\ 5.00000e-001 \\ 2.50000e-001 \\ 6.69873e-002 \\ 0.00000e+000 \end{pmatrix}, U_{\text{higher}} = \begin{pmatrix} 2.68427e-002 \\ 1.96502e-002 \\ 7.19247e-003 \\ -7.19247e-003 \\ -1.96502e-002 \\ -2.68427e-002 \\ -2.68427e-002 \\ -1.96502e-002 \\ -7.19247e-003 \\ 7.19247e-003 \\ 1.96502e-002 \\ 2.68427e-002 \end{pmatrix} \tag{7.30}$$

The nodal solution will always be the same for whichever polynomial degree that we choose as there will always be the linear basis functions, for all p. It is the finite element solution u_h whose accuracy will be increased as p is increased. Recall, from Section 4.2, that we define u_h to be

$$u_h(x) = \sum_{j=1}^{25} U_j \phi_j(x) = \sum_{K} \sum_{l=1}^{3} U_{DOF_l^K} \varphi_l(\xi). \tag{7.31}$$

We now have a finite element solution. Below, in Figure 25, we give the final plot for our finite element solution u_h, evaluated at the original grid points.

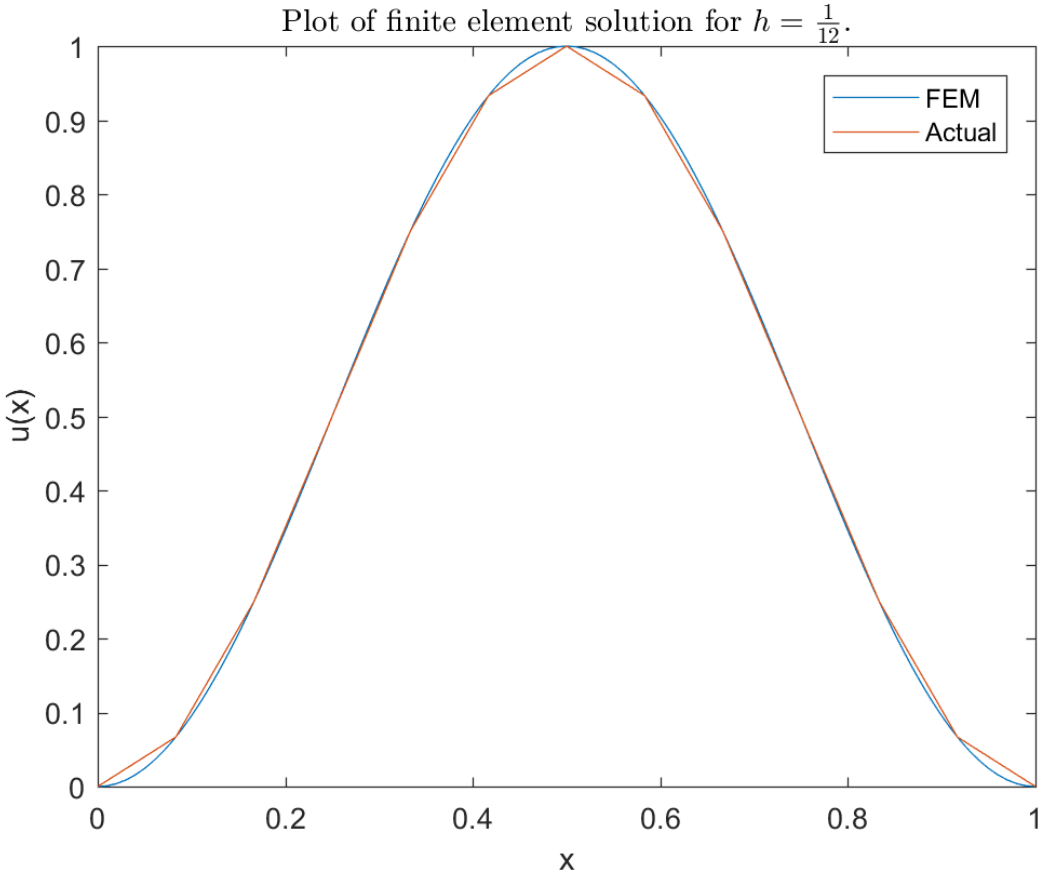


Figure 25: Plot of the finite element solution to our problem at the grid points.

7.2 A Posteriori Error Analysis for 1D Poisson Equation

Now, to demonstrate convergence of the finite element method, we shall do an error analysis on the same problem. A thorough explanation of a posteriori finite element error analysis can be found in Ainsworth and Oden (1997). Theoretically, the finite element method solution u_h should converge to the true solution u with order $\mathcal{O}\left(h^{p+1}\right)$ with respect to the L_2 norm.

To complete our analysis, we run the same code as demonstrated in Section 7.1 but instead we look at all polynomial degrees from $p = 1$ to $p = 4$, with varying h values. At the end of each run, we take the L_2 norm of the difference between the true solution $u(x) = \sin^2(\pi x)$ and the finite element solution u_h. As we discussed in Section 2.1.2, we

define the L_2 norm as

$$\|u - u_h\|_{L_2(0,1)} = \left(\int_0^1 |u - u_h|^2 \, dx \right)^{\frac{1}{2}}. \tag{7.32}$$

To evaluate the error in our code, we first perform numerical quadrature on $\|u - u_h\|^2_{L_2(0,1)}$ and then take the square root to give the L_2 error. We do this like so, in Listing 18.

```
1   double Error_L2 = 0;
2
3   for (int k=1; k<=myMesh->GetNumElements(); k++)
4   {
5       Vector* quadratureWeights = new Vector (n_q);
6       Matrix* localQuadraturePoints = new Matrix (1,n_q);
7       Matrix* globalQuadraturePoints = new Matrix (1,n_q);
8
9       myMesh->GetElement(k)->GetQuadrature(n_q, *quadratureWeights, *
            localQuadraturePoints, *globalQuadraturePoints);
10      for (int q=1; q<=n_q; q++)
11      {
12          Error_L2 += (pow((pow(sin(Pi*(*globalQuadraturePoints)(1,q)),2.0)) - (FE->
                ComputeUh(k, (*localQuadraturePoints)(1,q))),2.0) * (*quadratureWeights)(
                q));
13      }
14
15      delete quadratureWeights;
16      delete localQuadraturePoints;
17      delete globalQuadraturePoints;
18  }
19
20  Error_L2 = sqrt(Error_L2);
```

Listing 18: Computation of L_2 error.

The error values for the varying polynomial degrees p and step size h are saved to a file which we then use this to produce the error plot in Figure 26 below. This is a plot of the dth root of the number of DOFs versus Error (where d is the dimension of the problem) and we give it on a logarithmic scale.

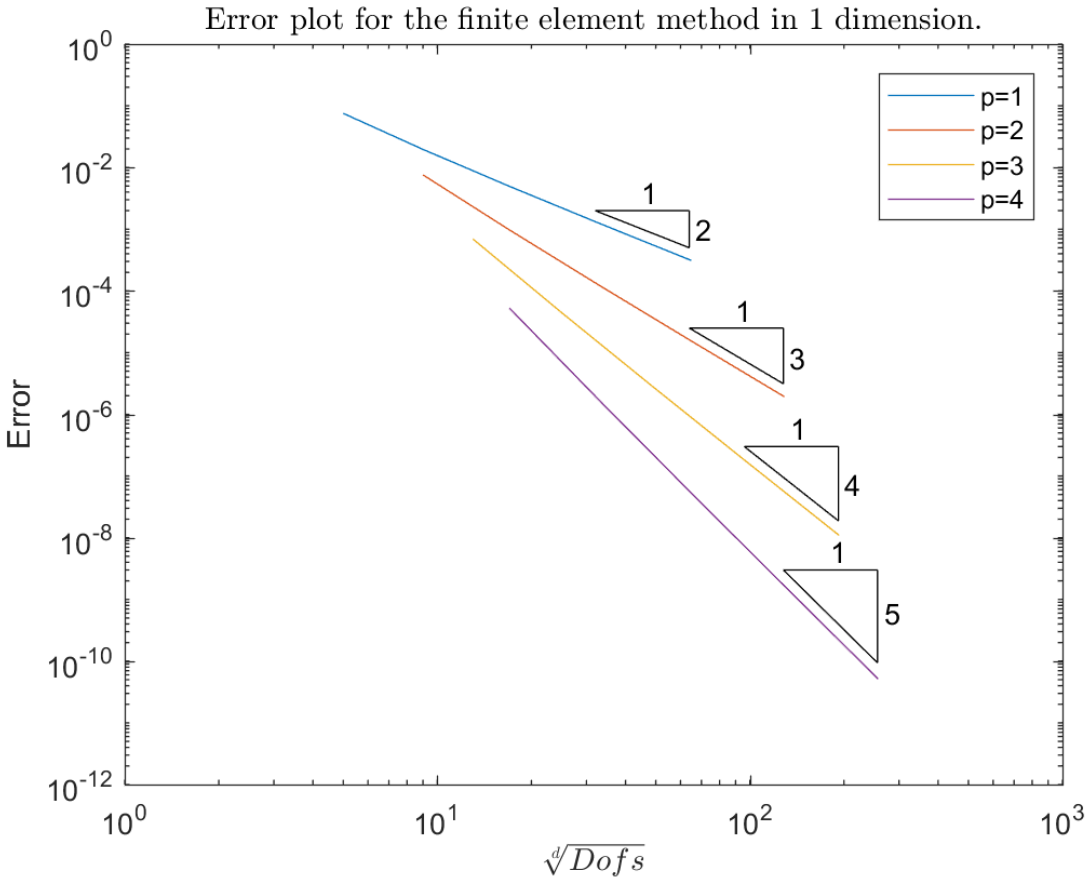


Figure 26: Error plot to show the convergence of the finite element method for varying polynomial degrees and step size h.

We can clearly see from the plot that the L_2 error of our problem, $\|u - u_h\|_{L_2(0,1)}$, is of order $\mathcal{O}\left(h^{p+1}\right)$. Hence, we can conclude that our implementation conforms to the theory behind the finite element method.

7.3 Two dimensional Inhomogeneous Modified Helmholtz Equation

We have now demonstrated our how to formulate, assemble and compute a finite element solution for a one dimensional problem, from both an analytical and computational respect. The principles behind this work in exactly the same way for two dimensional problems. For this example, we are going to be looking at the inhomogeneous modified

Helmholtz equation on $\Omega = [0,1] \times [0,1]$, given by

$$-\Delta u(x,y) + u(x,y) = (1 + 2\pi^2)\sin(\pi x)\sin(\pi y), \quad x, y \in (0,1), \tag{7.33}$$

$$u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0. \tag{7.34}$$

The analytic solution to this problem is

$$u(x,y) = \sin(\pi x)\sin(\pi y). \tag{7.35}$$

Recall, in Section 3.3, that we determined the weak formulation for a general elliptic PDE and proved, using the Lax-Milgram theorem (3.2.1), the existence and uniqueness of a finite element solution u_h. Hence, we can write the weak formulation of our problem (7.33) as

Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u \cdot \nabla v d\boldsymbol{x} + \int_\Omega uv d\boldsymbol{x} = \int_\Omega (1 + 2\pi^2)\sin(\pi x)\sin(\pi y) v d\boldsymbol{x}. \tag{7.36}$$

for all $v \in H_0^1(\Omega)$.

We shall not go into as much detail as the example in Section 7.1 due to the assembly and code being comparatively similar. To demonstrate the extent of our implementation, for this problem we will use is a hybrid grid made up of both quadrilateral and triangle elements. An example of what this grid will look like is given in Figure 27 below.

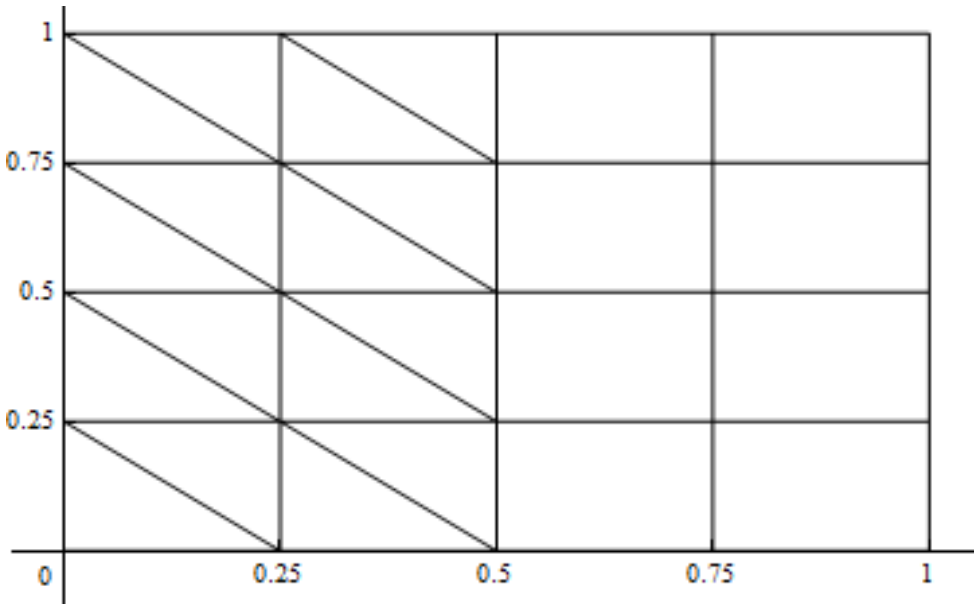


Figure 27: Example of a hybrid uniform grid with step size $h = 0.25$.

The grid points are still generated in the same way as before, except that the algorithm is extended to two dimensions. The only thing that changes is the initialisation of the elements, due to the change in element types. If the specified h value for the grid discretisation yields an odd number of elements across the rows, then we make the middle column part of the triangular mesh. The way we set up the connectivity arrays in our code is to have a separate global connectivity matrix for the quadrilateral elements and one for the triangle elements. Once these are populated for all the elements, we use the code below in Listing 19 to initialise the elements of the mesh.

```
1    int elementCounter = 1;
2    for (int i=1; i<=ConnectivityTriangles->GetNumberOfRows(); i++)
3    {
4        Vector* Con_temp = new Vector (ConnectivityTriangles->GetRowAsVector(i));
5        myMesh->InitialiseElement(elementCounter, *Con_temp, 1);
6        delete Con_temp;
7        elementCounter++;
8    }
9    for (int i=1; i<=ConnectivityQuads->GetNumberOfRows(); i++)
10   {
11       Vector* Con_temp = new Vector (ConnectivityQuads->GetRowAsVector(i));
12       myMesh->InitialiseElement(elementCounter, *Con_temp, 2);
13       delete Con_temp;
14       elementCounter++;
15   }
```

Listing 19: Initialisation of elements on a hybrid grid.

For this example, we shall choose an initial grid size of $h = 0.2$. This means that we have a mesh τ_h made up of 30 triangles on the left hand side and 10 quadrilaterals on the right hand side, with 36 grid points in total. We define our finite element space for this problem to be

$$V_h = \{v \in H_0^1(\Omega) : \quad v|_K \in P_1(K), \quad \forall K \in \tau_h\}, \tag{7.37}$$

where $P_1(K)$ is the space of linear polynomials associated with each K. Hence, we can give our finite element formulation as

Find $u_h \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h d\boldsymbol{x} + \int_\Omega u_h v_h d\boldsymbol{x} = \int_\Omega (1 + 2\pi^2) \sin(\pi x) \sin(\pi y) v_h d\boldsymbol{x}. \tag{7.38}$$

for all $v_h \in V_h$.

Again, employing the knowledge that $V_h = \text{span}\{\phi_1, \phi_2, \ldots, \phi_{36}\}$, we can set up our matrix system $AU = F$, where the entries of A are determined by

$$A_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j d\boldsymbol{x} + \int_\Omega \phi_i\phi_j d\boldsymbol{x}, \quad i,j = 1,2,\ldots,36, \tag{7.39}$$

and the entries in F are given by

$$F_i = \int_\Omega (1+2\pi^2)\sin(\pi x)\sin(\pi y)\phi_i d\boldsymbol{x}, \quad i = 1,2,\ldots,36. \tag{7.40}$$

Employing the mappings and transformations detailed in Sections 4.3, 6.4 and 6.3, we can give the entries of the local stiffness matrix and load vector, for each element, by

$$A^K_{lm} = \int_K \left(\left(J^{-T}\hat{\nabla}\varphi_l\right)\cdot\left(J^{-T}\hat{\nabla}\varphi_m\right) + \varphi_l\varphi_m\right)|J|\, d\boldsymbol{\xi}, \quad l,m = 1,2,3,(4), \tag{7.41}$$

$$F^K_l = \int_K (1+2\pi^2)\sin(\pi\cdot x(\xi,\eta))\sin(\pi\cdot y(\xi,\eta))\varphi_l\,|J|\, d\boldsymbol{\xi}, \quad l = 1,2,3,(4), \tag{7.42}$$

where the Jacobian matrices J and mappings $x(\xi,\eta), y(\xi,\eta)$ correspond to the element type of the K-th element. Recall the local basis functions from Section 4.4, given by

$\hat{Q}$	$\hat{T}$
$\varphi_1 = \varphi^{v_1}_{\hat{Q}}(\xi,\eta) = \ell_0(\xi)\ell_0(\eta),$	$\varphi_1 = \varphi^{v_1}_{\hat{T}}(\xi,\eta) = -\dfrac{\xi+\eta}{2},$
$\varphi_2 = \varphi^{v_2}_{\hat{Q}}(\xi,\eta) = \ell_1(\xi)\ell_0(\eta),$	$\varphi_2 = \varphi^{v_2}_{\hat{T}}(\xi,\eta) = \dfrac{1+\xi}{2},$
$\varphi_3 = \varphi^{v_3}_{\hat{Q}}(\xi,\eta) = \ell_1(\xi)\ell_1(\eta),$	$\varphi_3 = \varphi^{v_3}_{\hat{T}}(\xi,\eta) = \dfrac{1+\eta}{2}.$
$\varphi_4 = \varphi^{v_4}_{\hat{Q}}(\xi,\eta) = \ell_0(\xi)\ell_1(\eta).$	

Solving the populated system using our implementation, we get the solution vector U which we can in turn use to compute our finite element solution u_h. We visualise the final solution as a contour plot in Figure 28 below.

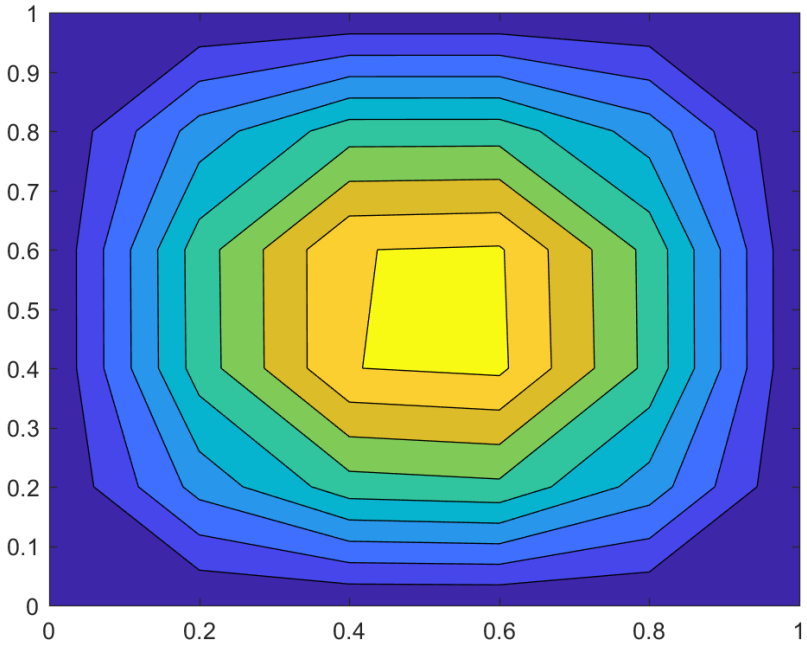


Figure 28: Contour plot of the finite element solution with a hybrid grid, for $h = 0.2$.

To demonstrate the effect that refining our grid has on the solution, we provide another contour plot (Figure 29) of the same problem but with a grid size of $h = 0.125$.

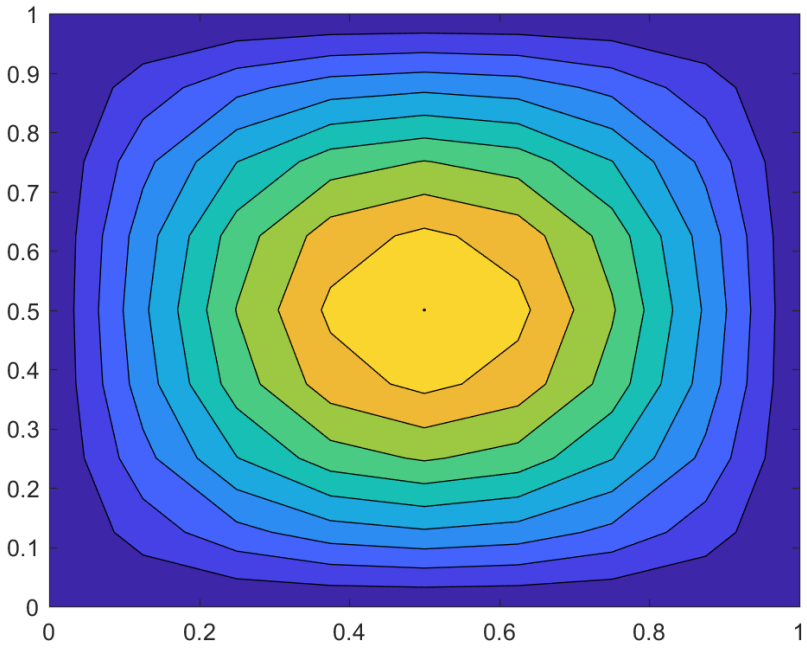


Figure 29: Contour plot of the finite element solution with a hybrid grid, for $h = 0.125$.

8 Conclusions

Throughout this dissertation, we have encountered a number of theoretical and practical techniques that the finite element method employs to solve partial differential equations. We started by studying the theory that underpins the formulation of finite element problems, such as Sobolev spaces, weak formulations and the existence and uniqueness of solutions. The majority of this inital theory, seen in Sections 2 and 3, is reviewed intensively in Brenner and Scott (2007). After exploring this, we moved on to Section 4, where we introduced the finite element method itself and discussed the choice of basis functions for varying element types. Although we only discussed linear bases for two dimensional elements, a step for the future would be to look at higher order bases, as seen in Šolín, Segeth, and Doležel (2003).

Section 5 saw the design and development of our implementation of the finite element method. We looked in depth at the structure of each class and how they interact with the other classes. Our implementation mirrors aspects of a selection of other existing finite element libraries, such as AptoFEM (Antonietti et al., 2011), deal.ii (Bangerth, Hartmann, and Kanschat, 2007) and FEniCS (Logg, Mardal, and Wells, 2012). As we are unable to evaluate integrals analytically within our code, we employ numerical quadrature techniques to approximate the entries in the finite element stiffness matrix and load vector. These techniques are discussed in Section 6 from both a theoretical and implementation perspective.

Finally, we brought together all the knowledge from these sections to demonstrate how the finite element method can be utilised to solve partial differential equations in both one and two dimensions. An a posteriori error analysis was undertaken and yielded important results showing that our implementation follows the theoretical error convergence rates detailed in Ainsworth and Oden (1997).

There are many possible extensions that can be made to the work achieved in this project, mainly with respect to our implementation. For all of our examples, we had to write a mesh generator for the specified domain. This can be particularly challenging for triangulations such as the hybrid grid used in Section 7.3. Ideally, we would like to be able

to have a pre-generated mesh, stored in a file format such as .VTK. A next step for our implementation could be creating a .VTK file reader to automatically import the grid points and the element connectivity arrays. Another obvious shortcoming is the implementation of only linear basis functions for the two dimensional elements. Although the basis functions themselves are easy to program, a new class designed to hold information about the interactions between the edges of adjacent elements would be required. This is because when working with higher order bases in two dimensions, the orientation of the element can present issues with conformity between elements.

The next logical step, other than the extensions already detailed, would be to look at development for three dimensional problems. Some new element types such as bricks, tetrahedrons and pyramids would need to introduced and have designated classes to store all their attributes and mappings. These are detailed in Warburton (1999) and Šolín, Segeth, and Doležel (2003). When working in three dimensions, each element contains faces as well as edges and vertices. Similarly to higher order bases in two dimensions, another class would be required to hold this data.

A final furthering of the work we have seen in this project is the theory and development behind *hp*-adaptive mesh refinement. This looks at dynamically refining the mesh coarseness and polynomial degree on specific elements in the domain to give an enhanced approximation to the solution in those areas. This is particularly applicable to domains such as an L shape, where the inner vertex can present difficulties for coarser triangulations.

Overall, the finite element method is a highly versatile and extensive tool and this dissertation has only just begun to scratch the surface of the theory and range of applications that it has to offer.

A Code

A.1 Basis Functions

```
void PolynomialSpace::EvaluateNthLegendrePolynomial(int n, Vector& pointsToEvaluate,
    Vector& legendrePoints)
{
    assert(pointsToEvaluate.GetSize() == legendrePoints.GetSize());
    assert(n >= 0);

    Vector* pn_prev = new Vector(pointsToEvaluate.GetSize());
    (*pn_prev) = 1;
    Vector* pn = new Vector(pointsToEvaluate);
    Vector* pn_next = new Vector(pointsToEvaluate.GetSize());

    if(n==0)
    {
        (*pn) = (*pn_prev);
    }

    if (n>1)
    {
        for (int i=1; i<n; i++)
        {
            for (int j=0; j<pn->GetSize(); j++)
            {
                (*pn_next)[j] = (pointsToEvaluate[j])*((*pn)[j])*((2.0*i)+1.0)/(i+1.0) -
                    ((*pn_prev)[j])*(i/(i+1.0));
            }
            (*pn_prev) = (*pn);
            (*pn) = (*pn_next);
        }
    }

    legendrePoints = (*pn);

    delete pn_prev;
    delete pn;
    delete pn_next;
}
```

```
void PolynomialSpace::EvaluateNthLobattoPolynomial(int n, Vector& pointsToEvaluate,
    Vector& lobattoPoints)
{
    assert(pointsToEvaluate.GetSize() == lobattoPoints.GetSize());
    assert(n >= 0);
```

```
5
6       if (n==0)
7       {
8           for (int i=0; i<pointsToEvaluate.GetSize(); i++)
9           {
10              lobattoPoints[i] = (1.0 - pointsToEvaluate[i])/2.0;
11          }
12      }
13      else if (n==1)
14      {
15          for (int i=0; i<pointsToEvaluate.GetSize(); i++)
16          {
17              lobattoPoints[i] = (1.0 + pointsToEvaluate[i])/2.0;
18          }
19      }
20      else
21      {
22          Vector* legendrePoints_prev = new Vector(pointsToEvaluate.GetSize());
23          Vector* legendrePoints = new Vector(pointsToEvaluate.GetSize());
24
25
26          EvaluateNthLegendrePolynomial(mNumElementDofs - 1, pointsToEvaluate, *
                legendrePoints_prev);
27          EvaluateNthLegendrePolynomial(mNumElementDofs, pointsToEvaluate, *legendrePoints)
                ;
28
29          for (int i=0; i<pointsToEvaluate.GetSize(); i++)
30          {
31              lobattoPoints[i] = (mNumElementDofs-1) * ((*legendrePoints_prev)[i] - (
                    pointsToEvaluate[i]*(*legendrePoints)[i]));
32          }
33
34          delete legendrePoints_prev;
35          delete legendrePoints;
36      }
37  }
```

```
1   void PolynomialSpace::ComputeBasis(Vector& localGridPoint, Vector& basisValues)
2   {
3       assert(basisValues.GetSize() == mNumElementDofs);
4
5       switch(mElementType)
6       {
7       case 0:
8           {
9               for (int i=1; i<=mNumElementDofs; i++)
```

```
            {
                basisValues(i) = EvaluateNthLobattoPolynomial(i-1, localGridPoint(1));
            }
        } break;
    case 1:
        {
            /* LINEAR BASES ONLY */
            assert (mPolynomialDegree == 1);

            basisValues(1) = -(localGridPoint(1)+localGridPoint(2))/2.0;
            basisValues(2) = (1+localGridPoint(1))/2.0;
            basisValues(3) = (1+localGridPoint(2))/2.0;
        } break;
    case 2:
        {
            /* LINEAR BASES ONLY */
            assert (mPolynomialDegree == 1);

            basisValues(1) = EvaluateNthLobattoPolynomial(0, localGridPoint(1))*
                EvaluateNthLobattoPolynomial(0, localGridPoint(2));
            basisValues(2) = EvaluateNthLobattoPolynomial(1, localGridPoint(1))*
                EvaluateNthLobattoPolynomial(0, localGridPoint(2));
            basisValues(3) = EvaluateNthLobattoPolynomial(1, localGridPoint(1))*
                EvaluateNthLobattoPolynomial(1, localGridPoint(2));
            basisValues(4) = EvaluateNthLobattoPolynomial(0, localGridPoint(1))*
                EvaluateNthLobattoPolynomial(1, localGridPoint(2));

        } break;
    }
}
```

```
void PolynomialSpace::ComputeGradBasis(Vector& localGridPoint, Matrix& gradBasisValues)
{
    assert(gradBasisValues.GetNumberOfColumns() == mNumElementDofs);

    switch(mElementType)
    {
    case 0:
        {
            assert(gradBasisValues.GetNumberOfRows() == 1);

            gradBasisValues(1,1) = -0.5;
            gradBasisValues(1,2) = 0.5;

            for (int i=3; i<=mNumElementDofs; i++)
            {
```

```
16                  gradBasisValues(1,i) =  sqrt((2.0*(i-1) - 1.0)/2.0)*
                        EvaluateNthLegendrePolynomial(i-2, localGridPoint(1));
17              }
18          } break;
19      case 1:
20          {
21              /* LINEAR BASES ONLY */
22              assert(mPolynomialDegree == 1);
23
24              gradBasisValues(1,1) = -0.5;
25              gradBasisValues(2,1) = -0.5;
26              gradBasisValues(1,2) = 0.5;
27              gradBasisValues(2,2) = 0;
28              gradBasisValues(1,3) = 0;
29              gradBasisValues(2,3) = 0.5;
30          } break;
31      case 2:
32          {
33              /* LINEAR BASES ONLY */
34              assert (mPolynomialDegree == 1);
35
36              gradBasisValues(1,1) = -0.5*EvaluateNthLobattoPolynomial(0, localGridPoint(2)
                    );
37              gradBasisValues(2,1) = -0.5*EvaluateNthLobattoPolynomial(0, localGridPoint(1)
                    );
38              gradBasisValues(1,2) = 0.5*EvaluateNthLobattoPolynomial(0, localGridPoint(2))
                    ;
39              gradBasisValues(2,2) = -0.5*EvaluateNthLobattoPolynomial(1, localGridPoint(1)
                    );
40              gradBasisValues(1,3) = 0.5*EvaluateNthLobattoPolynomial(1, localGridPoint(2))
                    ;
41              gradBasisValues(2,3) = 0.5*EvaluateNthLobattoPolynomial(1, localGridPoint(1))
                    ;
42              gradBasisValues(1,4) = -0.5*EvaluateNthLobattoPolynomial(1, localGridPoint(2)
                    );
43              gradBasisValues(2,4) = 0.5*EvaluateNthLobattoPolynomial(0, localGridPoint(1))
                    ;
44          } break;
45      }
46  }
```

A.2 Triangle

```
1   int Triangle::GetElementType() const
2   {
```

```
    return ElementType::Triangle;
}
```

```
void Triangle::MapLocalToGlobal(Matrix& localCoords, Matrix& globalCoords)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(localCoords.GetNumberOfRows() == globalCoords.GetNumberOfRows());
    assert(localCoords.GetNumberOfColumns() == globalCoords.GetNumberOfColumns());
    assert(localCoords.GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfColumns() == 3);

    for (int i=1; i<=globalCoords.GetNumberOfRows(); i++)
    {
        for (int j=1; j<=globalCoords.GetNumberOfColumns(); j++)
        {
            globalCoords(i,j) = -((localCoords(1,j) + localCoords(2,j))/2.0)*(*nodes)(i
                ,1) + ((localCoords(1,j) + 1)/2)*(*nodes)(i,2) + ((localCoords(2,j) + 1)
                /2.0)*(*nodes)(i,3);
        }
    }

    delete nodes;
}
```

```
void Triangle::MapGlobalToLocal(Matrix& globalCoords, Matrix& localCoords)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(globalCoords.GetNumberOfRows() == localCoords.GetNumberOfRows());
    assert(globalCoords.GetNumberOfColumns() == localCoords.GetNumberOfColumns());
    assert(globalCoords.GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfColumns() == 3);

    for (int j=1; j<=localCoords.GetNumberOfColumns(); j++)
    {
        localCoords(1,j) = ((*nodes)(1,3)*(-2.0*globalCoords(2,j)+(*nodes)(2,1)+(*nodes)
            (2,2)) + (*nodes)(1,1)*(2.0*globalCoords(2,j)-(*nodes)(2,2)-(*nodes)(2,3)) -
            (2.0*globalCoords(1,j)-(*nodes)(1,2))*((*nodes)(2,1)-(*nodes)(2,3)))/
                        ((*nodes)(1,3)*((*nodes)(2,1)-(*nodes)(2,2)) + (*nodes)(1,1)
                            *((*nodes)(2,2)-(*nodes)(2,3)) + (*nodes)(1,2)*((*nodes)
                            (2,3)-(*nodes)(2,1)));

        localCoords(2,j) = ((*nodes)(1,1)*(-2.0*globalCoords(2,j)+(*nodes)(2,2)+(*nodes)
            (2,3)) + (*nodes)(1,2)*(2.0*globalCoords(2,j)-(*nodes)(2,1)-(*nodes)(2,3)) -
            (2.0*globalCoords(1,j)-(*nodes)(1,3))*((*nodes)(2,1)-(*nodes)(2,2)))/
```

```
                        ((*nodes)(1,3)*((*nodes)(2,1)-(*nodes)(2,2)) + (*nodes)(1,1)
                            *((*nodes)(2,2)-(*nodes)(2,3)) + (*nodes)(1,2)*((*nodes)
                            (2,3)-(*nodes)(2,1)));
    }

    delete nodes;
}
```

```
void Triangle::ComputeMappingJacobian(Vector& pointToEval, Matrix& Jacobian)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(Jacobian.GetNumberOfRows() == Jacobian.GetNumberOfColumns());
    assert(Jacobian.GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfColumns() == 3);

    Jacobian(1,1) = ((*nodes)(1,2) - (*nodes)(1,1))/2.0;
    Jacobian(1,2) = ((*nodes)(1,3) - (*nodes)(1,1))/2.0;
    Jacobian(2,1) = ((*nodes)(2,2) - (*nodes)(2,1))/2.0;
    Jacobian(2,2) = ((*nodes)(2,3) - (*nodes)(2,1))/2.0;

    delete nodes;
}
```

A.3 Quadrilateral

```
int Quadrilateral::GetElementType() const
{
    return ElementType::Quadrilateral;
}
```

```
void Quadrilateral::MapLocalToGlobal(Matrix& localCoords, Matrix& globalCoords)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(localCoords.GetNumberOfRows() == globalCoords.GetNumberOfRows());
    assert(localCoords.GetNumberOfColumns() == globalCoords.GetNumberOfColumns());
    assert(localCoords.GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfColumns() == 4);

    for (int i=1; i<=globalCoords.GetNumberOfRows(); i++)
    {
        for (int j=1; j<=globalCoords.GetNumberOfColumns(); j++)
        {
```

```
            globalCoords(i,j) = 0.25*((*nodes)(i,1)*((1-localCoords(1,j))*(1-localCoords
                (2,j))) + (*nodes)(i,2)*((1+localCoords(1,j))*(1-localCoords(2,j)))+
                                (*nodes)(i,3)*((1+localCoords(1,j))*(1+localCoords(2,j)))
                                     + (*nodes)(i,4)*((1-localCoords(1,j))*(1+localCoords
                                    (2,j))));
        }
    }

    delete nodes;
}
```

```
void Quadrilateral::ComputeMappingJacobian(Vector& pointToEval, Matrix& Jacobian)
{
    Matrix* nodes = new Matrix (GetElementCoordinates());
    assert(Jacobian.GetNumberOfRows() == Jacobian.GetNumberOfColumns());
    assert(Jacobian.GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfRows() == 2);
    assert(nodes->GetNumberOfColumns() == 4);

    Jacobian(1,1) = 0.25*(-(*nodes)(1,1)*(1-pointToEval(2)) + (*nodes)(1,2)*(1-
        pointToEval(2)) + (*nodes)(1,3)*(1+pointToEval(2)) - (*nodes)(1,4)*(1+pointToEval
        (2)));
    Jacobian(1,2) = 0.25*(-(*nodes)(1,1)*(1-pointToEval(1)) - (*nodes)(1,2)*(1+
        pointToEval(1)) + (*nodes)(1,3)*(1+pointToEval(1)) + (*nodes)(1,4)*(1-pointToEval
        (1)));
    Jacobian(2,1) = 0.25*(-(*nodes)(2,1)*(1-pointToEval(2)) + (*nodes)(2,2)*(1-
        pointToEval(2)) + (*nodes)(2,3)*(1+pointToEval(2)) - (*nodes)(2,4)*(1+pointToEval
        (2)));
    Jacobian(2,2) = 0.25*(-(*nodes)(2,1)*(1-pointToEval(1)) - (*nodes)(2,2)*(1+
        pointToEval(1)) + (*nodes)(2,3)*(1+pointToEval(1)) + (*nodes)(2,4)*(1-pointToEval
        (1)));

    delete nodes;
}
```

A.4 FE_Solution

```
void FE_Solution::InitialiseElementDofs()
{
    mElementPolySpaceArray = new PolynomialSpace* [mMeshReference->GetNumElements()];
    dofStart = new Vector(mMeshReference->GetNumElements() + 1);

    (*dofStart)[0] = mMeshReference->GetNumNodes() + 1;

    for (int i=0; i<mMeshReference->GetNumElements(); i++)
```

```
    {
        mElementPolySpaceArray[i] = new PolynomialSpace(mPolynomialDegree, mMeshReference
            ->GetElement(i+1)->GetElementType());

        if (mMeshReference->GetElement(i+1)->GetElementType() == 0)
        {
            (*dofStart)[i+1] = (*dofStart)[i] + (mPolynomialDegree - 1);
        }
        else if (mMeshReference->GetElement(i+1)->GetElementType() == 1)
        {
            /* LINEAR BASES ONLY */
            (*dofStart)[i+1] = (*dofStart)[i];
        }
        else if (mMeshReference->GetElement(i+1)->GetElementType() == 2)
        {
            /* LINEAR BASES ONLY */
            (*dofStart)[i+1] = (*dofStart)[i];
        }
    }
}
```

```
void FE_Solution::ComputeGradBasis(int elementNumber, double localGridPoint, Matrix&
    gradBasisValues)
{
    /* For dimension 1 */
    GetElementPolynomialSpace(elementNumber)->ComputeGradBasis(localGridPoint,
        gradBasisValues);

    Matrix* jacobian = new Matrix (mMeshReference->GetDimension(), mMeshReference->
        GetDimension());

    mMeshReference->GetElement(elementNumber)->ComputeMappingJacobian(localGridPoint, *
        jacobian);

    gradBasisValues = gradBasisValues*(1.0/(*jacobian)(1,1));

    delete jacobian;
}
```

```
void FE_Solution::ComputeGradBasis(int elementNumber, Vector& localGridPoint, Matrix&
    gradBasisValues)
{
    /* For dimension 2 */
    GetElementPolynomialSpace(elementNumber)->ComputeGradBasis(localGridPoint,
        gradBasisValues);

```

```
    Matrix* jacobian = new Matrix (mMeshReference->GetDimension(), mMeshReference->
        GetDimension());
    mMeshReference->GetElement(elementNumber)->ComputeMappingJacobian(localGridPoint, *
        jacobian);

    Matrix* jacobianInverseTranspose = new Matrix (*jacobian);

    (*jacobianInverseTranspose)(1,1) = (*jacobian)(2,2);
    (*jacobianInverseTranspose)(1,2) = -(*jacobian)(2,1);
    (*jacobianInverseTranspose)(2,1) = -(*jacobian)(1,2);
    (*jacobianInverseTranspose)(2,2) = (*jacobian)(1,1);

    (*jacobianInverseTranspose) = (*jacobianInverseTranspose)*(1.0/jacobian->
        CalculateDeterminant());
    delete jacobian;

    for (int j=1; j<=gradBasisValues.GetNumberOfColumns(); j++)
    {
        Vector* gradTemp = new Vector(gradBasisValues.GetColumnAsVector(j));

        (*gradTemp) = (*jacobianInverseTranspose)*(*gradTemp);

        for (int i=1; i<=gradBasisValues.GetNumberOfRows(); i++)
        {
            gradBasisValues(i,j) = (*gradTemp)(i);
        }

        delete gradTemp;
    }

    delete jacobianInverseTranspose;
}
```

A.5 QuadratureLibrary

```
void QuadratureLibrary::EvaluateNthLegendrePolynomial(Vector& pointsToEvaluate, Vector&
    legendrePoints)
{
    Vector* pn_prev = new Vector(pointsToEvaluate.GetSize());
    (*pn_prev) = 1;
    Vector* pn = new Vector(pointsToEvaluate);
    Vector* pn_next = new Vector(pointsToEvaluate.GetSize());

    if (pointsToEvaluate.GetSize()>1)
    {
```

```
        for (int i=1; i<pointsToEvaluate.GetSize(); i++)
        {
            for (int j=0; j<pn->GetSize(); j++)
            {
                (*pn_next)[j] = (pointsToEvaluate[j])*((*pn)[j])*((2.0*i)+1.0)/(i+1.0) -
                    ((*pn_prev)[j])*(i/(i+1.0));
            }
            (*pn_prev) = (*pn);
            (*pn) = (*pn_next);
        }
    }

    legendrePoints = (*pn);

    delete pn_prev;
    delete pn;
    delete pn_next;
}
```

```
void QuadratureLibrary::EvaluateNthLegendrePolynomialFirstDerivative(Vector&
    pointsToEvaluate, Vector& legendrePoints)
{
    Vector* pn_prev = new Vector(pointsToEvaluate.GetSize());
    (*pn_prev) = 0;
    Vector* pn = new Vector(pointsToEvaluate.GetSize());
    (*pn) = 1;
    Vector* pn_next = new Vector(pointsToEvaluate.GetSize());

    if (pointsToEvaluate.GetSize()>1)
    {
        for (int i=1; i<pointsToEvaluate.GetSize(); i++)
        {
            for (int j=0; j<pn_next->GetSize(); j++)
            {
                (*pn_next)[j] = (pointsToEvaluate[j])*((*pn)[j])*(((2.0*i)+1.0)/i) - ((*
                    pn_prev)[j])*((i+1.0)/i);
            }
            (*pn_prev) = (*pn);
            (*pn) = (*pn_next);
        }
    }

    legendrePoints = (*pn);

    delete pn_prev;
    delete pn;
```

```
    delete pn_next;
}
```

```
void QuadratureLibrary::MapReferenceQuadrilateralToTriangle(Matrix& quadrilateralPoints,
    Matrix& trianglePoints)
{
    assert(trianglePoints.GetNumberOfRows() == quadrilateralPoints.GetNumberOfRows());
    assert(trianglePoints.GetNumberOfColumns() == quadrilateralPoints.GetNumberOfColumns
        ());
    assert(trianglePoints.GetNumberOfRows() == 2);

    for (int j=1; j<=trianglePoints.GetNumberOfColumns(); j++)
    {
        trianglePoints(1,j) = 0.5*(-1 + quadrilateralPoints(1,j) - quadrilateralPoints(2,
            j) - quadrilateralPoints(1,j)*quadrilateralPoints(2,j));
        trianglePoints(2,j) = quadrilateralPoints(2,j);
    }
}
```

```
void QuadratureLibrary::NewtonForLegendre(double tolerance, int maxIterations, Vector&
    initialGuess, Vector& legendreRoots)
{
    assert(initialGuess.GetSize() == legendreRoots.GetSize());

    int iterationCounter = 0;
    double error;
    Vector* error_vec = new Vector (legendreRoots.GetSize());
    Vector* legendreEval = new Vector (legendreRoots.GetSize());
    Vector* legendreDeriv = new Vector (legendreRoots.GetSize());

    do
    {
        EvaluateNthLegendrePolynomial(initialGuess, *legendreEval);
        EvaluateNthLegendrePolynomialFirstDerivative(initialGuess, *legendreDeriv);

        for (int i=0; i<legendreRoots.GetSize(); i++)
        {
            legendreRoots[i] = initialGuess[i] - (*legendreEval)[i]/(*legendreDeriv)[i];
        }

        (*error_vec) = legendreRoots - initialGuess;
        initialGuess = legendreRoots;

        error = error_vec->CalculateNorm(2);
        iterationCounter++;
    } while (error >= tolerance && iterationCounter <= maxIterations);
```

```
27
28      delete error_vec;
29      delete legendreEval;
30      delete legendreDeriv;
31  }
```

```
1   void QuadratureLibrary::GetQuadrature(const int elementType, const int n_q, Vector&
        weights, Matrix& gaussPoints)
2   {
3       int n;
4       assert(gaussPoints.GetNumberOfColumns() == n_q);
5       assert(weights.GetSize() == n_q);
6
7       switch(elementType)
8       {
9       case 0:
10          {
11              assert(gaussPoints.GetNumberOfRows() == 1);
12
13              n = n_q;
14
15              Vector* gaussVec = new Vector(n);
16              Vector* weightVec = new Vector(n);
17              ComputeGQPoints(*gaussVec);
18              ComputeGQWeights(*weightVec, *gaussVec);
19
20              for (int j=1; j<=n; j++)
21              {
22                  gaussPoints(1,j) = (*gaussVec)(j);
23              }
24
25              weights = (*weightVec);
26
27              delete gaussVec;
28              delete weightVec;
29          } break;
30
31      case 1:
32          {
33              assert(gaussPoints.GetNumberOfRows() == 2);
34
35              n = sqrt(n_q);
36
37              Vector* gaussVec = new Vector(n);
38              Vector* weightVec = new Vector(n);
39              ComputeGQPoints(*gaussVec);
```

```
40              ComputeGQWeights(*weightVec, *gaussVec);
41
42              for (int i=1; i<=n; i++)
43              {
44                  for (int j=1; j<=n; j++)
45                  {
46                      gaussPoints(1,((i-1)*n)+j) = (*gaussVec)(i);
47                      gaussPoints(2,((i-1)*n)+j) = (*gaussVec)(j);
48                      weights((i-1)*n+j) = (*weightVec)(i)*(*weightVec)(j);
49                  }
50              }
51
52              delete gaussVec;
53              delete weightVec;
54
55              Matrix* quadrilateralPoints = new Matrix(gaussPoints);
56              MapReferenceQuadrilateralToTriangle(*quadrilateralPoints, gaussPoints);
57              delete quadrilateralPoints;
58
59              for (int j=1; j<=weights.GetSize(); j++)
60              {
61                  weights(j) = weights(j)*((1-gaussPoints(2,j))*0.5);
62              }
63          } break;
64
65      case 2:
66          {
67              assert(gaussPoints.GetNumberOfRows() == 2);
68
69              n = sqrt(n_q);
70
71              Vector* gaussVec = new Vector(n);
72              Vector* weightVec = new Vector(n);
73              ComputeGQPoints(*gaussVec);
74              ComputeGQWeights(*weightVec, *gaussVec);
75
76              for (int i=1; i<=n; i++)
77              {
78                  for (int j=1; j<=n; j++)
79                  {
80                      gaussPoints(1,((i-1)*n)+j) = (*gaussVec)(i);
81                      gaussPoints(2,((i-1)*n)+j) = (*gaussVec)(j);
82                      weights((i-1)*n+j) = (*weightVec)(i)*(*weightVec)(j);
83                  }
84              }
85
```

```
            delete gaussVec;
            delete weightVec;
        } break;
    }
}
```

A.6 SparseMatrix

```
SparseMatrix::SparseMatrix(FE_Solution& FE, int numElements)
{
    mNumRows = FE.GetNumberOfDofs();
    mNumCols = FE.GetNumberOfDofs();

    Vector* NNZ = new Vector (FE.GetNumberOfDofs());

    for (int k=1; k<=numElements; k++)
    {
        for (int i=1; i<=FE.GetElementPolynomialSpace(k)->GetNumElementDofs(); i++)
        {
            for (int j=1; j<=FE.GetElementPolynomialSpace(k)->GetNumElementDofs(); j++)
            {
                (*NNZ)((FE.GetElementDofs(k))(i)) ++;
            }
        }
    }

    mNumNonZeros = 0;

    for (int i=0; i<NNZ->GetSize(); i++)
    {
        mNumNonZeros += (*NNZ)[i];
    }

    mRow_ptr = new Vector (mNumRows+1);
    (*mRow_ptr) = 1;

    for (int i=1; i<mRow_ptr->GetSize(); i++)
    {
        (*mRow_ptr)[i] = (*mRow_ptr)[i-1] + (*NNZ)[i-1];
    }

    delete NNZ;

    mData = new Vector (mNumNonZeros);
    mCol_index = new Vector (mNumNonZeros);
```

```
38
39      bool ColIndexExists;
40
41      for (int k=1; k<=numElements; k++)
42      {
43          for (int i=1; i<=FE.GetElementPolynomialSpace(k)->GetNumElementDofs(); i++)
44          {
45              for (int j=1; j<=FE.GetElementPolynomialSpace(k)->GetNumElementDofs(); j++)
46              {
47                  ColIndexExists = false;
48                  for (int l=0; l<(*mRow_ptr)((FE.GetElementDofs(k))(i)+1) - (*mRow_ptr)((
                        FE.GetElementDofs(k))(i)); l++)
49                  {
50                      if ((*mCol_index)((*mRow_ptr)((FE.GetElementDofs(k))(i)) + l) == (FE.
                            GetElementDofs(k))(j))
51                      {
52                          ColIndexExists = true;
53                      }
54                  }
55                  if (!ColIndexExists)
56                  {
57                      for (int l=0; l<(*mRow_ptr)((FE.GetElementDofs(k))(i)+1) - (*mRow_ptr
                            )((FE.GetElementDofs(k))(i)); l++)
58                      {
59                          if ((*mCol_index)((*mRow_ptr)((FE.GetElementDofs(k))(i)) + l) ==
                                0)
60                          {
61                              (*mCol_index)((*mRow_ptr)((FE.GetElementDofs(k))(i)) + l) = (
                                    FE.GetElementDofs(k))(j);
62                              break;
63                          }
64                      }
65                  }
66              }
67          }
68      }
69  }
```

A.7 Example population of local element stiffness matrices and load vector

```
1       for (int q=1; q<=n_q; q++)
2       {
3           Vector* basisValues = new Vector(numElementDofs);
```

```
4             Matrix* basisGrad = new Matrix(myMesh->GetDimension(),numElementDofs);
5             Matrix* jacobian = new Matrix (myMesh->GetDimension(), myMesh->GetDimension()
                  );
6             Vector* pointToEval = new Vector(localQuadraturePoints->GetColumnAsVector(q))
                  ;
7
8             myMesh->GetElement(k)->ComputeMappingJacobian(*pointToEval, *jacobian);
9             delete pointToEval;
10
11            FE->ComputeBasis(k, (*localQuadraturePoints)(1,q), *basisValues);
12            FE->ComputeGradBasis(k, (*localQuadraturePoints)(1,q), *basisGrad);
13
14            for (int i=1; i<=A_loc->GetNumberOfRows(); i++)
15            {
16                for (int j=1; j<=A_loc->GetNumberOfColumns(); j++)
17                {
18                    (*A_loc)(i,j) += (*basisGrad)(1,i)*(*basisGrad)(1,j)*(*
                          quadratureWeights)(q);
19                }
20            }
21            for (int i=1; i<=f_loc->GetSize(); i++)
22            {
23                (*f_loc)(i) += (*basisValues)(i)*(*quadratureWeights)(q) * (-2.0*pow(Pi
                      ,2.0)*cos(2.0*Pi*(*globalQuadraturePoints)(1,q)));
24            }
25
26            delete basisValues;
27            delete basisGrad;
28            delete jacobian;
29        }
```

References

[1] R. A. Adams and J. J. F. Fournier. *Sobolev Spaces*. Vol. 140. Academic press, 2003.

[2] M. Ainsworth and J. T. Oden. "A Posteriori Error Estimation in Finite Element Analysis". In: *Computer Methods in Applied Mechanics and Engineering* 142.1-2 (1997), pp. 1–88.

[3] P. Antonietti et al. *AptoFEM. Users Manual*. Version 1.9.082. 2011.

[4] W. Bangerth, R. Hartmann, and G. Kanschat. "deal.II A General-Purpose Object-Oriented Finite Element Library". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007), p. 24.

[5] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 2001.

[6] S. Brenner and R. Scott. *The Mathematical Theory of Finite Element Methods*. Vol. 15. Springer Science & Business Media, 2007.

[7] A. Bressan. *Lecture Notes on Sobolev Spaces*. The Pennsylvania State University, 2012.

[8] H. Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Springer Science & Business Media, 2010.

[9] E. Cheney and D. Kincaid. *Numerical Mathematics and Computing*. Nelson Education, 2012.

[10] P. G. Ciarlet. *The Finite Element Method For Elliptic Problems*. SIAM, 2002.

[11] S. Deng. *Quadrature Formulas in Two Dimensions*. Math5172 - Finite Element Method. University of North Carolina at Charlotte, 2010.

[12] J. A. T. Dow. *A Unified Approach to the Finite Element Method and Error Analysis Procedures*. Academic Press, 1998.

[13] P. Drábek and G. Holubová. *Elements of Partial Differential Equations*. 2nd ed. De Gruyter, 2014.

[14] M. G. Duffy. “Quadrature Over a Pyramid or Cube of Integrands with a Singularity at a Vertex”. In: *SIAM journal on Numerical Analysis* 19.6 (1982), pp. 1260–1262.

[15] J. F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, 2013.

[16] D. Gilbarg and N. S. Trudinger. *Elliptic Partial Differential Equations of Second Order*. Springer, 2015.

[17] P. R. Halmos. *Finite-Dimensional Vector Spaces*. Springer Science & Business Media, 1974.

[18] P. Houston. *Finite Element Methods for Elliptic Problems*. G14FEM: Introduction to Finite Element Methods. University of Nottingham, 2016.

[19] P. Houston. *Weak Solutions to Elliptic Partial Differential Equations*. G14FEM: Introduction to Finite Element Methods. University of Nottingham, 2016.

[20] C. Ketelsen. *Sparse Data Structures*. APPM 6640: Multigrid Methods. University of Colorado, Boulder, 2015.

[21] A. Logg, K. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Vol. 84. Springer Science & Business Media, 2012.

[22] M. Mandelker. “Supports of Continuous Functions”. In: *Transactions of the American Mathematical Society* 156 (1971), pp. 73–83.

[23] S. E. Mousavi and N. Sukumar. “Generalized Duffy Transformation for Integrating Vertex Singularities”. In: *Computational Mechanics* 45.2 (2010), pp. 127–140.

[24] L. R. Nackman et al. *Incremental Compilation of C++ Programs*. US Patent 6,182,281. 2001.

[25] J. Pitt-Francis and J. Whiteley. *Guide to Scientific Computing in C++*. Springer Science & Business Media, 2012.

[26] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Vol. 37. Springer Science & Business Media, 2010.

[27] X. Saint Raymond. *Elementary Introduction to the Theory of Pseudodifferential Operators*. Vol. 3. CRC Press, 1991.

[28] P. Šolín. *Partial Differential Equations and the Finite Element Method*. Wiley Online Library, 2006.

[29] P. Šolín, K. Segeth, and I. Doležel. *Higher-Order Finite Element Methods*. CRC Press, 2003.

[30] Timothy Warburton. "Spectral/hp Methods on Polymorphic Multidomains: Algorithms and Applications". PhD. 1999.

[31] W. Zheng and H. Qi. "On Friedrichs–Poincaré-Type Inequalities". In: *Journal of mathematical analysis and applications* 304.2 (2005), pp. 542–551.