Numerical Methods for Variational Problems

Edition 2024.0

David A. Ham, Colin J. Cotter and Jack S. Hale

Sep 06, 2024

CONTENTS

I Numerical analysis **1**

1 Introduction **3**
- 1.1 Poisson's equation in the unit square . . . 3
- 1.2 Triangulations . . . 3
- 1.3 Our first finite element space . . . 4
- 1.4 Integral formulations and L_2 . . . 4
- 1.5 Finite element derivative . . . 5
- 1.6 Towards the finite element discretisation . . . 5
- 1.7 Practical implementation . . . 8

2 Finite element spaces: local to global **11**
- 2.1 Ciarlet's finite element . . . 11
- 2.2 Vandermonde matrix and unisolvence . . . 12
- 2.3 2D and 3D finite elements . . . 14
- 2.4 Some more exotic elements . . . 16
- 2.5 Global continuity . . . 17

3 Interpolation operators **21**
- 3.1 Local and global interpolation operators . . . 21
- 3.2 Measuring interpolation errors . . . 22
- 3.3 Approximation by averaged Taylor polynomials . . . 23
- 3.4 Local and global interpolation errors . . . 25

4 Finite element problems: solvability and stability **29**
- 4.1 Finite element spaces and other Hilbert spaces . . . 29
- 4.2 Linear forms on Hilbert spaces . . . 31
- 4.3 Variational problems on Hilbert spaces . . . 33
- 4.4 Solvability and stability of some finite element discretisations . . . 34

5 Convergence of finite element approximations **41**
- 5.1 Weak derivatives . . . 41
- 5.2 Sobolev spaces . . . 42
- 5.3 Variational formulations of PDEs . . . 43
- 5.4 Galerkin approximations of linear variational problems . . . 45
- 5.5 Interpolation error in H^k spaces . . . 46
- 5.6 Convergence of the finite element approximation to the Helmholtz problem . . . 48
- 5.7 Epilogue . . . 49

6 Stokes equation **51**
- 6.1 Strong form of the equations . . . 51
- 6.2 Variational form of the equations . . . 51
- 6.3 The inf-sup condition . . . 52
- 6.4 Solveability of mixed problems . . . 54
- 6.5 Solveability of Stokes equation . . . 56
- 6.6 Discretisation of Stokes equations . . . 57

6.7 The MINI element . . . 59

II Implementation Exercise **63**

0 The implementation exercise **65**
0.1 Formalities and marking scheme . . . 65
0.2 Obtaining the skeleton code . . . 66
0.3 Skeleton code documentation . . . 67
0.4 How to do the implementation exercises . . . 67
0.5 Testing your work . . . 67
0.6 Coding style and commenting . . . 68
0.7 Getting help . . . 68
0.8 Tips and tricks for the implementation exercise . . . 69

1 Numerical quadrature **71**
1.1 Exact and incomplete quadrature . . . 71
1.2 Examples in one dimension . . . 72
1.3 Reference cells . . . 72
1.4 Quadrature rules on reference elements . . . 73
1.5 Legendre-Gauß quadrature in one dimension . . . 74
1.6 Extending Legendre-Gauß quadrature to two dimensions . . . 74
1.7 Implementing quadrature rules in Python . . . 75

2 Constructing finite elements **77**
2.1 A worked example . . . 77
2.2 Types of node . . . 78
2.3 The Lagrange element nodes . . . 78
2.4 Solving for basis functions . . . 79
2.5 Implementing finite elements in Python . . . 80
2.6 Implementing the Lagrange Elements . . . 80
2.7 Tabulating basis functions . . . 80
2.8 Gradients of basis functions . . . 81
2.9 Interpolating functions to the finite element nodes . . . 82

3 Meshes **83**
3.1 Mesh entities . . . 83
3.2 Adjacency . . . 84
3.3 Mesh geometry . . . 84
3.4 A mesh implementation in Python . . . 84

4 Function spaces: associating data with meshes **87**
4.1 Local numbering and continuity . . . 87
4.2 Implementing local numbering . . . 88
4.3 Global numbering . . . 88
4.4 The cell-node map . . . 89
4.5 Implementing function spaces in Python . . . 89

5 Functions in finite element spaces **91**
5.1 A python implementation of functions in finite element spaces . . . 91
5.2 Interpolating values into finite element spaces . . . 91
5.3 Integration . . . 93

6 Assembling and solving finite element problems **97**
6.1 Assembling the right hand side . . . 98
6.2 Assembling the left hand side matrix . . . 98
6.3 The method of manufactured solutions . . . 100
6.4 Errors and convergence . . . 101
6.5 Implementing finite element problems . . . 102

7 Dirichlet boundary conditions **103**
7.1 An algorithm for homogeneous Dirichlet conditions 104
7.2 Implementing boundary conditions 104
7.3 Inhomogeneous Dirichlet conditions 104

8 Nonlinear problems **105**
8.1 A model problem 105
8.2 Residual form 105
8.3 Linearisation and Gâteaux Derivatives 106
8.4 A Taylor expansion and Newton's method 107
8.5 Implementing a nonlinear problem 109

9 Mixed problems **111**
9.1 Vector-valued finite elements 112
9.2 Vector-valued function spaces 113
9.3 Functions in vector-valued spaces 114
9.4 Mixed function spaces 116
9.5 Manufacturing a solution to the Stokes equations 117
9.6 Implementing the Stokes problem 118

10 fe_utils package **119**
10.1 Subpackages 119
10.2 Submodules 121
10.3 fe_utils.finite_elements module 121
10.4 fe_utils.function_spaces module 122
10.5 fe_utils.mesh module 124
10.6 fe_utils.quadrature module 125
10.7 fe_utils.reference_elements module 126
10.8 fe_utils.utils module 126
10.9 Module contents 126

Bibliography **127**

Python Module Index **129**

Index **131**

Part I

Numerical analysis

CHAPTER ONE

INTRODUCTION

A video recording of the following material is available here.

In this section we provide an introduction that establishes some initial ideas about how the finite element method works and what it is about.

The finite element method is an approach to solving partial differential equations (PDEs) on complicated domains. It has the flexibility to build discretisations that can increase the order of accuracy, and match the numerical discretisation to the physical problem being modelled. It has an elegant mathematical formulation that lends itself both to mathematical analysis and to flexible code implementation. In this course we blend these two directions together.

1.1 Poisson's equation in the unit square

A video recording of the following material is available here.

In this introduction we concentrate on the specific model problem of Poisson's equation in the unit square.

Definition 1.1 (Poisson's equation in the unit square) *Let $\Omega = [0,1] \times [0,1]$. For a given function f, we seek u such that*

$$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u := -\nabla^2 u = f, \quad u(0,y) = u(1,y) = 0, \ \frac{\partial u}{\partial y}(x,0) = \frac{\partial u}{\partial y}(x,1) = 0. \tag{1.1}$$

In this problem, the idea is that we are given a specific known function f (for example, $f = sin(2\pi x)sin(2\pi y)$), and we have to find the corresponding unknown function u that satisfies the equation (including the boundary conditions). Here we have combined a mixture of Dirichlet boundary conditions specifying the value of the function u, and Neumann boundary conditions specifying the value of the normal derivative $\partial u/\partial n := n \cdot \nabla u$. This is because these two types of boundary conditions are treated differently in the finite element method, and we would like to expose both treatments in the same example. The treatment of boundary conditions is one of the strengths of the finite element method.

1.2 Triangulations

A video recording of the following material is available here.

The description of our finite element method starts by considering a triangulation.

Definition 1.2 (Triangulation) *Let Ω be a polygonal subdomain of $\mathbb{R}^2$. A triangulation $\mathcal{T}$ of Ω is a set of triangles $\{K_i\}_{i=1}^N$, such that:*

1. $\operatorname{int} K_i \cap \operatorname{int} K_j = \emptyset, \quad i \neq j$, *where* int *denotes the interior of a set (no overlaps).*
2. $\cup K_i = \bar{\Omega}$, *the closure of Ω (triangulation covers Ω).*

3. *No vertex of any triangle lies in the interior of an edge of another triangle (triangle vertices only meet other triangle vertices).*

1.3 Our first finite element space

A video recording of the following material is available here.

The idea is that we will approximate functions which are polynomial (at some chosen degree) when restricted to each triangle, with some chosen continuity conditions between triangles. We shall call the space of possible functions under these choices a finite element space. In this introduction, we will just consider the following space.

Definition 1.3 (The (P1) finite element space) *Let $\mathcal{T}$ be a triangulation of Ω. Then the P1 finite element space is a space V_h containing all functions v such that*

1. $v \in C^0(\Omega)$ *the space of continuous functions at every point in* Ω,
2. $v|_{K_i}$ *is a linear function for each* $K_i \in \mathcal{T}$.

We also define the following subspace,

$$\mathring{V}_h = \{v \in V_h : v(0,y) = v(1,y) = 0\}. \tag{1.2}$$

This is the subspace of the P1 finite element space V_h of functions that satisfy the Dirichlet boundary conditions. We will search only amongst $\mathring{V}_h$ for our approximate solution to the Poisson equation. This is referred to as strong boundary conditions. Note that we do not consider any subspaces related to the Neumann conditions. These will emerge later.

1.4 Integral formulations and L_2

A video recording of the following material is available here.

The finite element method is based upon integral formulations of partial differential equations. Rather than checking if two functions are equal by checking their value at every point, we will just check that they are equal in an integral sense. We do this by introducing the L^2 norm, which is a way of measuring the "magnitude" of a function.

Definition 1.4 *For a real-valued function f on a domain Ω, with Lebesgue integral*

$$\int_\Omega f(x)\, dx,$$

we define the L^2 norm of f,

$$\|f\|_{L^2(\Omega)} = \left(\int_\Omega |f(x)|^2\, dx\right)^{1/2}.$$

This motivates us to say that two functions are equal if the L^2 norm of their difference is zero. It only makes sense to do that if the functions individually have finite L^2 norm, which then also motivates the L^2 function space.

Definition 1.5 *We define $L^2(\Omega)$ as the set of functions*

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

and identify two functions f and g if $\|f-g\|_{L^2(\Omega)} = 0$, in which case we write $f \equiv g$ in L^2.

Example 1.6 *Consider the two functions f and g defined on $\Omega = [0,1] \times [0,1]$ with*

$$f(x,y) = \begin{cases} 1 & x \geq 0.5, \\ 0 & x < 0.5, \end{cases} \quad g(x,y) = \begin{cases} 1 & x > 0.5, \\ 0 & x \leq 0.5. \end{cases}$$

Since f and g only differ on the line $x = 0.5$ which has zero area, then $\|f-g\|_{L^2(\Omega)} = 0$, and so $f \equiv g$ in L^2.

1.5 Finite element derivative

A video recording of the following material is available here.

Functions in V_h do not have derivatives everywhere. This means that we have to work with a more general definition (and later we shall learn when it does and does not work).

Definition 1.7 (Finite element partial derivative) *The finite element partial derivative* $\frac{\partial^{FE}}{\partial x_i}u$ *of* u *is defined in* $L^2(\Omega)$ *such that restricted to* K_i*, we have*

$$\frac{\partial^{FE} u}{\partial x_i}|_{K_i} = \frac{\partial u}{\partial x_i}.$$

Here we see why we needed to introduce L^2: we have a definition that does not have a unique value on the edge between two adjacent triangles. This is verified in the following exercises.

Exercise 1.8 *Let* V_h *be a P1 finite element space for a triangulation* $\mathcal{T}$ *of* Ω*. For all* $u \in V_h$*, show that the definition above uniquely defines* $\frac{\partial^{FE} u}{\partial x_i}$ *in* $L^2(\Omega)$*.*

Exercise 1.9 *Let* $u \in C^1(\Omega)$ *(the space of functions with continuous partial derivatives at every point in* Ω*). Show that the finite element partial derivative and the usual derivative are equal in* $L^2(\Omega)$*.*

In view of this second exercise, in this section we will consider all derivatives to be finite element derivatives. In later sections we shall consider an even more general definition of the derivative which contains both of these definitions.

1.6 Towards the finite element discretisation

A video recording of the following material is available here.

We will now use the finite element derivative to develop the finite element discretisation. We assume that we have a solution u to Equation (1.1) that is sufficiently smooth (e.g. $u \in C^1$ in this case). (Later, we will consider more general types of solutions to this equation, but this assumption just motivates things for the time being.)

We take $v \in \mathring{V}_h$, multiply by Equation (1.1), and integrate over the domain.

We then use the following integration by parts result.

Theorem 1.10 (Integration by parts) *For a suitably smooth function* u *and with* n *an outward normal to the boundary* $\partial\Omega$ *then for a suitable smooth function* v

$$\int_\Omega (-\Delta u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} v \, (n \cdot \nabla u) \, dS.$$

Proof 1.11

See e.g. Brenner and Scott Section 5.1 including weaker assumptions.

Using this integration by parts in each triangle K_i then gives

$$\sum_i \left(\int_{K_i} \nabla v \cdot \nabla u \, dx - \int_{\partial K_i} v n \cdot \nabla u \, dS \right) = \int_\Omega v f \, dx,$$

where n is the unit outward pointing normal to K_i.

Next, we consider each interior edge f in the triangulation, formed as the intersection between two neighbouring triangles $K_i \cap K_j$. If $i > j$, then we label the K_i side of f with a $+$, and the K_j side with a $-$. Then, denoting Γ as the union of all such interior edges, we can rewrite our equation as

$$\int_\Omega \nabla v \cdot \nabla u \, dx - \int_\Gamma v n^+ \cdot \nabla u + v n^- \cdot \nabla u \, dS - \int_{\partial\Omega} v n \cdot \nabla u \, dS = \int_\Omega v f \, dx,$$

where $n^\pm$ is the unit normal to f pointing from the $\pm$ side into the $\mp$ side. Since $n^- = -n^+$, the interior edge integrals vanish.

Further, on the boundary, either v vanishes (at $x = 0$ and $x = 1$) or $n \cdot \nabla u$ vanishes (at $y = 0$ and $y = 1$), and we obtain

$$\int_\Omega \nabla v \cdot \nabla u \, dx = \int_\Omega v f \, dx.$$

The finite element approximation is then defined by requiring that this equation holds for all $v \in \mathring{V}_h$ and when we approximate u by $u_h \in \mathring{V}_h$.

A video recording of the following material is available here.

Definition 1.12 *The finite element approximation $u_h \in \mathring{V}_h$ to the solution u of Poisson's equation is defined by*

$$\int_\Omega \nabla v \cdot \nabla u_h \, dx = \int_\Omega v f \, dx, \quad \forall v \in \mathring{V}_h. \tag{1.3}$$

A video recording of the following material is available here.

We now present some numerical results for the case $f = 2\pi^2 \sin(\pi x) \sin(2\pi y)$.

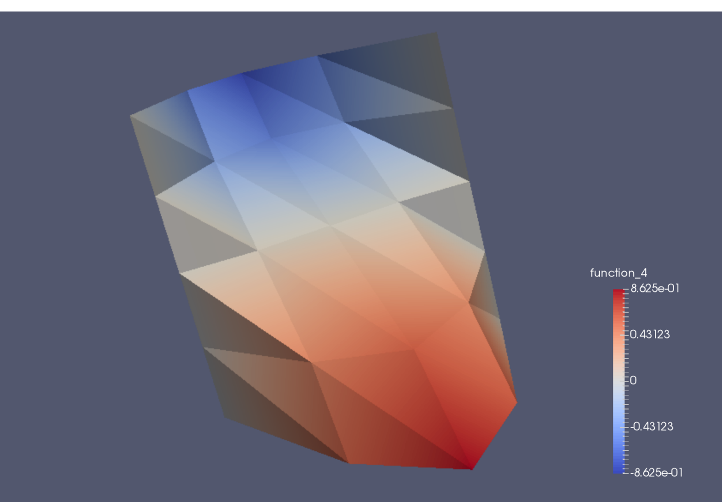

Fig. 1.1: Numerical solution on a 4×4 mesh.

We see that for this example, the error is decreasing as we increase the number of triangles, for the meshes considered.

A video recording of the following material is available here.

In general, our formulation raises a number of questions.

1. Is u_h unique?
2. What is the size of the error $u - u_h$?
3. Does this error go to zero as the mesh is refined?
4. For what types of functions f can these questions be answered?
5. What other kinds of finite element spaces are there?
6. How do we extend this approach to other PDEs?
7. How can we calculate u_h using a computer?

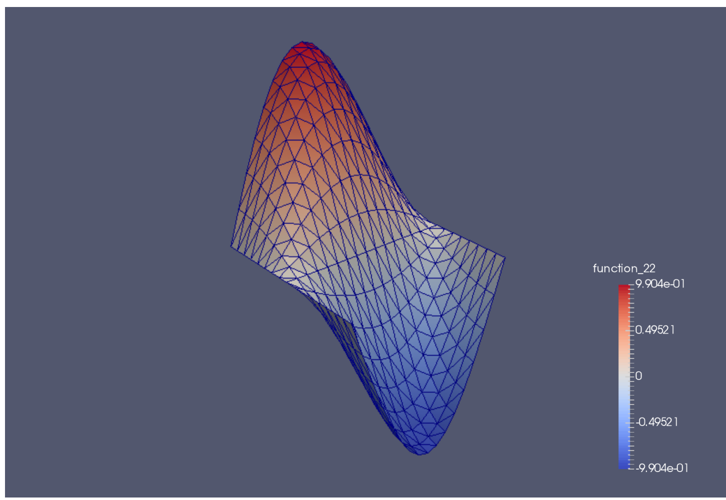

Fig. 1.2: Numerical solution on a 16×16 mesh.

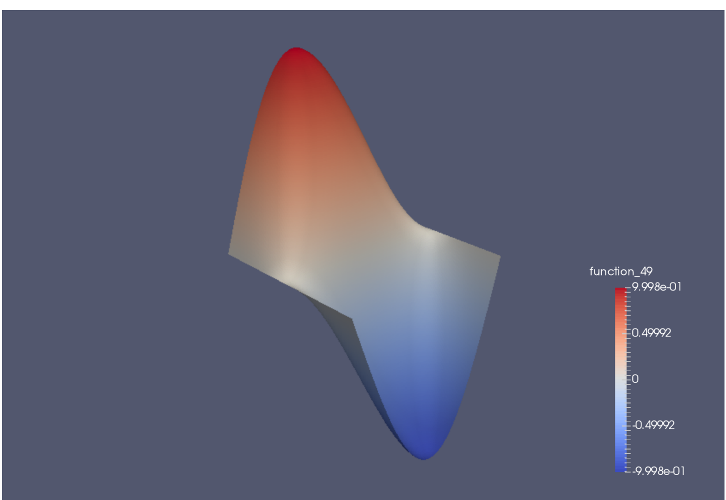

Fig. 1.3: Numerical solution on a 128×128 mesh.

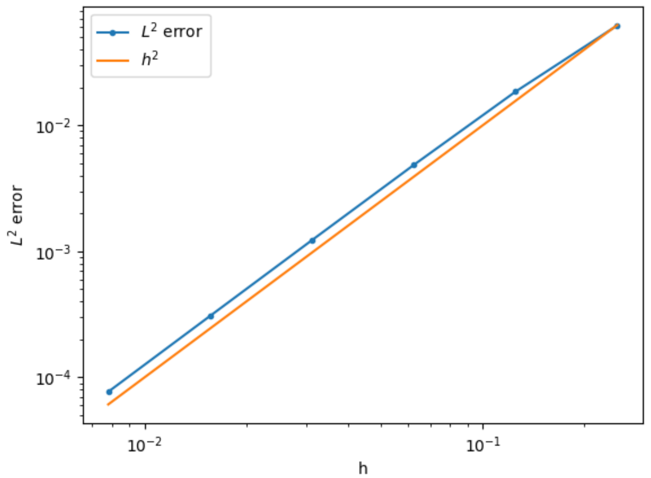

Fig. 1.4: Plot showing error versus mesh resolution. We observe the error decreases proportionally to h^2, where h is the maximum triangle edge size in the triangulation.

We shall aim to address these questions, at least partially, through the rest of this course. For now, we concentrate on the final question, in general terms.

In this course we shall mostly concentrate on finite element methods for elliptic PDEs, of which Poisson's equation is an example, using continuous finite element spaces, of which $P1$ is an example. The design, analysis and implementation of finite methods for PDEs is a huge field of current research, and includes parabolic and elliptic PDEs and other PDEs from elasticity, fluid dynamics, electromagnetism, mathematical biology, mathematical finance, astrophysics and cosmology, etc. This course is intended as a starting point to introduce the general concepts that can be applied in all of these areas.

Exercise 1.13 *Derive a finite element approximation for the following problem.*

Find u such that

$$-\nabla \cdot ((2+\sin(2\pi x))\nabla u) = \exp(\cos(2\pi x)),$$

with boundary conditions $u = 0$ on the entire boundary.

Exercise 1.14 *Derive a finite element approximation for the following problem.*

Find u such that

$$-\nabla^2 u = \exp(xy),$$

in the 1×1 square region as above, with boundary conditions $u = x(1-x)$ on the entire boundary.

Exercise 1.15 *Derive a finite element approximation for the following problem.*

Find u such that

$$-\nabla^2 u = \frac{1}{1+x^2+y^2},$$

in the 1×1 square region, with boundary conditions $u + \frac{\partial u}{\partial n} = x(1-x)$ on the entire boundary.

1.7 Practical implementation

A video recording of the following material is available here.

The finite element approximation above is only useful if we can actually compute it. To do this, we need to construct an efficient basis for $P1$, which we call the nodal basis.

Definition 1.16 (P1 nodal basis) *Let $\{z_i\}_{i=1}^M$ indicate the vertices in the triangulation $\mathcal{T}$. For each vertex z_i, we define a basis function $\phi_i \in V_h$ by*

$$\phi_i(z_j) = \delta_{ij} := \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases}$$

We can define a similar basis for $\mathring{V}_h$ by removing the basis functions ϕ_i corresponding to vertices z_i on the Dirichlet boundaries $x = 0$ and $x = 1$; the dimension of the resulting basis is $\bar{M}$.

If we expand u_h and v in the basis for $\mathring{V}_h$,

$$u_h(x) = \sum_{i=1}^{\bar{M}} u_i \phi_i(x), \quad v(x) = \sum_{i=1}^{\bar{M}} v_i \phi_i(x),$$

into Equation (1.3), then we obtain

$$\sum_{i=1}^{\bar{M}} v_i \left(\sum_{j=1}^{\bar{M}} \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, dx u_j - \int_\Omega \phi_i f \, dx \right) = 0.$$

Since this equation must hold for all $v \in \mathring{V}_h$, then it must hold for all basis coefficients v_i, and we obtain the matrix-vector system

$$K\mathrm{u} = \mathrm{f},$$

where

$$K_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, dx,$$
$$\mathrm{u} = (u_1, u_2, \ldots, u_{\bar{M}})^T,$$
$$\mathrm{f} = (f_1, f_2, \ldots, f_{\bar{M}})^T, \quad f_i = \int_\Omega \phi_i f \, dx.$$

Once we have solved for u, we can use these basis coefficients to reconstruct the solution u_h. The system is square, but we do not currently know that K is invertible. This is equivalent to the finite element approximation having a unique solution u_h, which we shall establish in later sections. This motivates why we care that u_h exists and is unique.

A video recording of the following material is available here.

Putting solvability aside for the moment, the goal of the implementation sections of this course is to explain how to efficiently form K and f, and solve this system. For now we note a few following aspects that suggest that this might be possible. First, the matrix K and vector f can be written as sums over elements,

$$K_{ij} = \sum_{K\in\mathcal{T}} \int_K \nabla\phi_i \cdot \nabla\phi_j \, dx,$$
$$\text{where } f_i = \sum_{K\in\mathcal{T}} \int_K \phi_i f \, dx.$$

For each entry in the sum for K_{ij}, the integrand is composed entirely of polynomials (actually constants in this particular case, but we shall shortly consider finite element spaces using polynomials of higher degree). This motivates our starting point in exposing the computer implementation, namely the integration of polynomials over triangles using quadrature rules. This will also motivate an efficient way to construct derivatives of polynomials evaluated at quadrature points. Further, we shall shortly develop an interpolation operator $\mathcal{I}$ such that $\mathcal{I}_f \in V_h$. If we replace f by $\mathcal{I}_f$ in the approximations above, then the evaluation of f_i can also be performed via quadrature rules.

Even further, the matrix K is very sparse, since in most triangles, both ϕ_i and ϕ_j are zero. Any efficient implementation must make use of this and avoid computing integrals that return zero. This motivates the concept of global assembly, the process of looping over elements, computing only the contributions to K that are non-zero from that element. Finally, the sparsity of K means that the system should be solved using numerical linear algebra algorithms that can exploit this sparsity.

Having set out the main challenges of the computational implementation, we now move on to define and discuss a broader range of possible finite element spaces.

CHAPTER TWO

FINITE ELEMENT SPACES: LOCAL TO GLOBAL

In this section, we discuss the construction of general finite element spaces. Given a triangulation $\mathcal{T}$ of a domain Ω, finite element spaces are defined according to

1. the form the functions take (usually polynomial) when restricted to each cell (a triangle, in the case considered so far),
2. the continuity of the functions between cells.

We also need a mechanism to explicitly build a basis for the finite element space. We first do this by looking at a single cell, which we call the local perspective. Later we will take the global perspective, seeing how function continuity is enforced between cells.

2.1 Ciarlet's finite element

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

The first part of the definition is formalised by Ciarlet's definition of a finite element.

Definition 2.1 (Ciarlet's finite element) *Let*

1. *the element domain $K \subset \mathbb{R}^n$ be some bounded closed set with piecewise smooth boundary,*
2. *the space of shape functions $\mathcal{P}$ be a finite dimensional space of functions on K, and*
3. *the set of nodal variables $\mathcal{N} = (N_0, \ldots, N_k)$ be a basis for the dual space P'.*

Then $(K, \mathcal{P}, \mathcal{N})$ is called a finite element.

For the cases considered in this course, K will be a polygon such as a triangle, square, tetrahedron or cube, and P will be a space of polynomials. Here, P' is the dual space to P, defined as the space of linear functions from P to $\mathbb{R}$. Examples of dual functions to P include:

1. The evaluation of $p \in P$ at a point $x \in K$.
2. The integral of $p \in P$ over a line $l \in K$.
3. The integral of $p \in P$ over K.
4. The evaluation of a component of the derivative of $p \in P$ at a point $x \in K$.

Exercise 2.2 *Show that the four examples above are all linear functions from P to $\mathbb{R}$.*

Exercise 2.3 *For a domain K and shape space P, is the following functional a nodal variable? Explain your answer.*

$$N_0(p) = \int_K p^2 \, dx.$$

Ciarlet's finite element provides us with a standard way to define a basis for the P, called the nodal basis.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Definition 2.4 ((local) nodal basis) *Let $(K, \mathcal{P}, \mathcal{N})$ be a finite element. The nodal basis is the basis $\{\phi_0, \phi_2, \ldots, \phi_k\}$ of $\mathcal{P}$ that is dual to $\mathcal{N}$, i.e.*

$$N_i(\phi_j) = \delta_{ij}, \quad 0 \leq i, j \leq k.$$

We now introduce our first example of a Ciarlet element.

Definition 2.5 (The 1-dimensional Lagrange element) *The 1-dimensional Lagrange element $(K, \mathcal{P}, \mathcal{N})$ of degree k is defined by*

1. *K is the interval $[a, b]$ for $-\infty < a < b < \infty$.*
2. *$\mathcal{P}$ is the $(k+1)$-dimensional space of degree k polynomials on K,*
3. *$\mathcal{N} = \{N_0, \ldots, N_k\}$ with*

$$N_i(v) = v(x_i),\ x_i = a + (b-a)i/k, \quad \forall v \in \mathcal{P},\ i = 0, \ldots, k.$$

Exercise 2.6 *Show that the nodal basis for $\mathcal{P}$ is given by*

$$\phi_i(x) = \frac{\prod_{j=0, j\neq i}^{k}(x - x_j)}{\prod_{j=0, j\neq i}^{k}(x_i - x_j)}, \quad i = 0, \ldots, k.$$

2.2 Vandermonde matrix and unisolvence

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

More generally, It is useful computationally to write the nodal basis in terms of another arbitrary basis $\{\psi_i\}_{i=0}^k$. This transformation is represented by the Vandermonde matrix.

Definition 2.7 (Vandermonde matrix) *Given a dual basis $\mathcal{N}$ and a basis $\{\psi_i\}_{i=0}^k$, the Vandermonde matrix is the matrix V with coefficients*

$$V_{ij} = N_j(\psi_i).$$

This relationship is made clear by the following lemma.

Lemma 2.8 *The expansion of the nodal basis $\{\phi_i\}_{i=0}^k$ in terms of another basis $\{\psi_i\}_{i=0}^k$ for $\mathcal{P}$,*

$$\phi_i(x) = \sum_{j=0}^{k} \mu_{ij}\psi_j(x),$$

has coefficients μ_{ij}, $0 \leq i, j \leq k$ given by

$$\mu = V^{-1},$$

where μ is the corresponding matrix.

Proof 2.9 *The nodal basis definition becomes*

$$\delta_{ij} = N_j(\phi_i) = \sum_{l=0}^{k} \mu_{il} N_j(\psi_l) = \sum_{l=0}^{k} \mu_{il} V_{lj} = (\mu V)_{ij},$$

where μ is the matrix with coefficients μ_{ij}, and V is the matrix with coefficients $N_j(\psi_i)$.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Exercise 2.10 *Consider the following finite element.*

- *K is the interval $[0, 1]$.*
- *P is the quadratic polynomials on K.*
- *The nodal variables are:*

$$N_0[p] = p(0),\ N_1[p] = p(1),\ N_2 = \int_0^1 p(x)\,dx.$$

Find the corresponding nodal basis.

Given a triple $(K, \mathcal{P}, \mathcal{N})$, it is necessary to verify that $\mathcal{N}$ is indeed a basis for $\mathcal{P}'$, i.e. that the Ciarlet element is well-defined. Then the nodal basis is indeed a basis for $\mathcal{P}$ by construction. The following lemma provides a useful tool for checking this.

Lemma 2.11 (dual condition) *Let $K, \mathcal{P}$ be as defined above, and let $\{N_0, N_1, \ldots, N_k\} \in \mathcal{P}'$. Let $\{\psi_0, \psi_1, \ldots, \psi_k\}$ be a basis for $\mathcal{P}$.*

Then the following three statements are equivalent.

1. *$\{N_0, N_1, \ldots, N_k\}$ is a basis for $\mathcal{P}'$.*
2. *The Vandermonde matrix with coefficients*

$$V_{ij} = N_j(\psi_i),\ 0 \leq i, j \leq k,$$

   is invertible.

3. *If $v \in \mathcal{P}$ satisfies $N_i(v) = 0$ for $i = 0, \ldots, k$, then $v \equiv 0$.*

Proof 2.12 *Let $\{N_0, N_1, \ldots, N_k\}$ be a basis for $\mathcal{P}'$. This is equivalent to saying that given element E of $\mathcal{P}'$, we can find basis coefficients $\{e_i\}_{i=0}^k \in \mathbb{R}$ such that*

$$E = \sum_{i=0}^{k} e_i N_i.$$

This in turn is equivalent to being able to find a vector $e = (e_0, e_1, \ldots, e_k)^T$ such that

$$b_i = E(\psi_i) = \sum_{j=0}^{k} e_j N_j(\psi_i) = \sum_{j=0}^{k} e_j V_{ij},$$

i.e. the equation $Ve = b$ is solvable. This means that (1) is equivalent to (2).

On the other hand, we may expand any $v \in \mathcal{P}$ according to

$$v(x) = \sum_{i=0}^{k} f_i \psi_i(x).$$

Then

$$N_i(v) = 0 \iff \sum_{j=0}^{k} f_j N_i(\psi_j) = 0, \quad i = 0, 1, \ldots, k,$$

by linearity of N_i. So (3) is equivalent to

$$\sum_{j=0}^{k} f_j N_i(\psi_j) = 0, \quad i = 0, 1, \ldots, k \implies f_j = 0,\ j = 0, 1, \ldots, k,$$

which is equivalent to V^T being invertible, which is equivalent to V being invertible, and so (3) is equivalent to (2).

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

This result leads us to introducing the following terminology.

Definition 2.13 (Unisolvence.) *We say that $\mathcal{N}$ determines $\mathcal{P}$ if it satisfies condition 3 of Lemma 2.11. If this is the case, we say that $(K, \mathcal{P}, \mathcal{N})$ is unisolvent.*

We can now go and directly apply this lemma to the 1D Lagrange elements.

Corollary 2.14 *The 1D degree k Lagrange element is a finite element.*

Proof 2.15 *Let $(K, \mathcal{P}, \mathcal{N})$ be the degree k Lagrange element. We need to check that $\mathcal{N}$ determines $\mathcal{P}$. Let $v \in \mathcal{P}$ with $N_i(v) = 0$ for all $N_i \in \mathcal{N}$. This means that*

$$v(a + (b - a)i/k) = 0,\ i =, 0, 1, \ldots, k,$$

which means that v vanishes at $k + 1$ points in K. Since v is a degree k polynomial, it must be zero by the fundamental theorem of algebra.

Exercise 2.16 *Consider the following proposed finite element.*

- *K is the interval $[0, 1]$.*
- *P is the linear polynomials on K.*
- *The nodal variables are:*

$$N_0[p] = p(0.5), N_1 = \int_0^1 p(x)\, dx.$$

Is this finite element unisolvent? Explain your answer.

2.3 2D and 3D finite elements

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We would like to construct some finite elements with 2D and 3D domains K. The fundamental theorem of algebra does not directly help us there, but the following lemma is useful when checking that $\mathcal{N}$ determines $\mathcal{P}$ in those cases.

Lemma 2.17 *Let $p(x) : \mathbb{R}^d \to \mathbb{R}$ be a polynomial of degree $k \geq 1$ that vanishes on a hyperplane Π_L defined by*

$$\Pi_L = \{x : L(x) = 0\},$$

for a non-degenerate affine function $L(x) : \mathbb{R}^d \to \mathbb{R}$. Then $p(x) = L(x)q(x)$ where $q(x)$ is a polynomial of degree $k - 1$.

Proof 2.18 *Choose coordinates (by shifting the origin and applying a linear transformation) such that $x = (x_1, \ldots, x_d)$ with $L(x) = x_d$, so Π_L is defined by $x_d = 0$. Then the general form for a polynomial is*

$$P(x_1, \ldots, x_d) = \sum_{i_d=0}^{k} \left(\sum_{|i_1+\ldots+i_{d-1}| \leq k-i_d} c_{i_1,\ldots,i_{d-1},i_d} x_d^{i_d} \prod_{l=1}^{d-1} x_l^{i_l} \right),$$

Then, $p(x_1, \ldots, x_{d-1}, 0) = 0$ for all $(x_1, \ldots, x_{d-1})$, so

$$0 = \left(\sum_{|i_1+\ldots+i_{d-1}| \leq k} c_{i_1,\ldots,i_{d-1},0} \prod_{l=1}^{d-1} x_l^{i_l} \right)$$

which means that

$$c_{i_1,\ldots,i_{d-1},0} = 0, \quad \forall |i_1 + \ldots + i_{d-1}| \leq k.$$

This means we may rewrite

$$P(x) = L(x) \underbrace{\left(\sum_{i_d=1}^{k} \sum_{|i_1+\ldots+i_{d-1}| \leq k-i_d} c_{i_1,\ldots,i_{d-1},i_d} x_d^{i_d-1} \prod_{l=1}^{d-1} x_l^{i_l} \right)},$$

$$P(x) = \underbrace{x_d}_{L(x)} \underbrace{\left(\sum_{i_d=0}^{k-1} \sum_{|i_1+\ldots+i_{d-1}| \leq k-i_d} c_{i_1,\ldots,i_{d-1},i_d} x_d^{i_d-1} \prod_{l=1}^{d-1} x_l^{i_l} \right)}_{Q(x)},$$

with $\deg(Q) = k - 1$.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Exercise 2.19 *The following polynomial vanishes on the line $y = 1 - x$. Show that it satisfies the result of the previous theorem.*

$$x^5 + 5x^4 y - x^4 + 6x^3 y^2 - 4x^3 y - 2x^2 y^3 - 2x^2 y^2 - 3xy^4 + 4xy^3 + y^5 - y^4$$

Equipped with this tool we can consider some finite elements in two dimensions.

Definition 2.20 (Lagrange elements on triangles) *The triangular Lagrange element of degree k $(K, \mathcal{P}, \mathcal{N})$, denoted Pk, is defined as follows.*

1. *K is a (non-degenerate) triangle with vertices z_1, z_2, z_3.*
2. *$\mathcal{P}$ is the space of degree k polynomials on K.*
3. *$\mathcal{N} = \{N_{i,j} : 0 \leq i \leq k,\ 0 \leq j \leq i\}$ defined by $N_{i,j}(v) = v(x_{i,j})$ where*

$$x_{i,j} = z_1 + (z_2 - z_1)\frac{i}{k} + (z_3 - z_1)\frac{j}{k}.$$

We illustrate this for the cases $k = 1, 2$.

Example 2.21 (P1 elements on triangles) *The nodal basis for P1 elements is point evaluation at the three vertices.*

Example 2.22 (P2 elements on triangles) *The nodal basis for P2 elements is point evaluation at the three vertices, plus point evaluation at the three edge centres.*

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We now need to check that that the degree k Lagrange element is a finite element, i.e. that $\mathcal{N}$ determines $\mathcal{P}$. We will first do this for $P1$.

Lemma 2.23 *The degree 1 Lagrange element on a triangle K is a finite element.*

Proof 2.24 *Let Π_1, Π_2, Π_3 be the three lines containing the vertices z_2 and z_3, z_1 and z_3, and z_1 and z_3 respectively, and defined by $L_1 = 0$, $L_2 = 0$, and $L_3 = 0$ respectively. Consider a linear polynomial p vanishing at z_1, z_2, and z_3. The restriction $p|_{\Pi_1}$ of p to Π_1 is a linear function vanishing at two points, and therefore $p = 0$ on Π_1, and so $p = L_1(x)Q(x)$, where $Q(x)$ is a degree 0 polynomial, i.e. a constant c. We also have*

$$0 = p(z_1) = cL_1(z_1) \implies c = 0,$$

since $L_1(z_1) \neq 0$, and hence $p(x) \equiv 0$. This means that $\mathcal{N}$ determines $\mathcal{P}$.

A video recording of the following material is available here.

Exercise 2.25 *Let K be a rectangle, P be the polynomial space spanned by $\{1, x, y, xy\}$, let $\mathcal{N}$ be the set of dual elements corresponding to point evaluation at each vertex of the rectangle. Show that $\mathcal{N}$ determines the finite element.*

This technique can then be extended to degree 2.

Lemma 2.26 *The degree 2 Lagrange element is a finite element.*

Proof 2.27 *Let p be a degree 2 polynomial with $N_i(p)$ for all of the degree 2 dual basis elements. Let Π_1, Π_2, Π_3, L_1, L_2 and L_3 be defined as for the proof of Lemma . $p|_{\Pi_1}$ is a degree 2 scalar polynomial vanishing at 3 points, and therefore $p = 0$ on Π_1, and so $p(x) = L_1(x)Q_1(x)$ with $\deg(Q_1) = 1$. We also have $0 = p|_{\Pi_2} = L_1 Q_1|_{\Pi_2}$, so $Q_1|_{\Pi_2} = 0$ and we conclude that $p(x) = cL_1(x)L_2(x)$. Finally, p also vanishes at the midpoint of L_3, so we conclude that $c = 0$ as required.*

The technique extends further to degree 3.

Exercise 2.28 *Show that the degree 3 Lagrange element is a finite element.*

Going beyond degree 3, we have more than 1 nodal variable taking point evaluation inside the triangle. To deal with this, we use the nested triangular structure of the Lagrange triangle.

Lemma 2.29 *The degree k Lagrange element is a finite element for $k > 3$.*

Proof 2.30 *We prove by induction. Assume that the degree $k-3$ Lagrange element is a finite element. Let p be a degree k polynomial with $N_i(p)$ for all of the degree k dual basis elements. Let Π_1, Π_2, Π_3, L_1, L_2 and L_3 be defined as for the proof of lemma 2.23. The restriction $p|_{\Pi_1}$ is a degree k polynomial in one variable that vanishes at $k+1$ points, and therefore $p(x) = L_1(x)Q_1(x)$, with $\deg(Q_1) = k-1$. p and therefore Q also vanishes on Π_2, so $Q_1(x) = L_2(x)Q_2(x)$.*

Repeating the argument again means that $p(x) = L_1(x)L_2(x)L_3(x)Q_3(x)$, with $\deg(Q_3) = k-3$. Q_3 must vanish on the remaining points in the interior of K, which are arranged in a smaller triangle K' and correspond to the evaluation points for a degree $k-3$ Lagrange finite element on K'. From the inductive hypothesis, and using the results for $k = 1, 2, 3$, we conclude that $Q_3 \equiv= 0$, and therefore $p \equiv 0$ as required.

2.4 Some more exotic elements

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We now consider some finite elements that involve derivative evaluation. The Hermite elements involve evaluation of first derivatives as well as point evaluations.

Definition 2.31 (Cubic Hermite elements on triangles) *The cubic Hermite element is defined as follows:*

1. *K is a (nondegenerate) triangle,*
2. *$\mathcal{P}$ is the space of cubic polynomials on K,*
3. *$\mathcal{N} = \{N_1, N_2, \ldots, N_{10}\}$ defined as follows:*
   - *$(N_1, \ldots, N_3)$: evaluation of p at vertices,*
   - *$(N_4, \ldots, N_9)$: evaluation of the gradient of p at the 3 triangle vertices.*
   - *N_{10}: evaluation of p at the centre of the triangle.*

It turns out that the Hermite element is insufficient to guarantee functions with continuous derivatives between triangles. This problem is solved by the Argyris element.

Definition 2.32 (Quintic Argyris elements on triangles) *The quintic Argyris element is defined as follows:*

1. *K is a (nondegenerate) triangle,*
2. *$\mathcal{P}$ is the space of quintic polynomials on K,*

3. *$\mathcal{N}$ defined as follows:*
   - *evaluation of p at 3 vertices,*
   - *evaluation of gradient of p at 3 vertices,*
   - *evaluation of Hessian of p at 3 vertices,*
   - *evaluation of the gradient normal to 3 triangle edges.*

2.5 Global continuity

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Next we need to know how to glue finite elements together to form spaces defined over a triangulation (mesh). To do this we need to develop a language for specifying connections between finite element functions between element domains.

Definition 2.33 (Finite element space) *Let $\mathcal{T}$ be a triangulation made of triangles K_i, with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$. A space V of functions on $\mathcal{T}$ is called a finite element space if for each $u \in V$, and for each $K_i \in \mathcal{T}$, $u|_{K_i} \in \mathcal{P}_i$.*

Note that the set of finite elements do not uniquely determine a finite element space, since we also need to specify continuity requirements between triangles, which we will do in this chapter.

Definition 2.34 (Finite element space) *A finite element space V is a C^m finite element space if $u \in C^m$ for all $u \in V$.*

The following lemma guides use in how to inspect the continuity of finite element functions.

Lemma 2.35 (Continuity lemma) *Let $\mathcal{T}$ be a triangulation on Ω, and let V be a finite element space defined on $\mathcal{T}$. The following two statements are equivalent.*

1. *V is a C^m finite element space.*
2. *The following two conditions hold.*

- *For each vertex z in $\mathcal{T}$, let $\{K_i\}_{i=1}^m$ be the set of triangles that contain z. Then $u|_{K_1}(z) = u|_{K_2}(z) = \ldots = u|_{K_m}(z)$, for all functions $u \in V$, and similarly for all of the partial derivatives of degrees up to m.*
- *For each edge e in $\mathcal{T}$, let K_1, K_2 be the two triangles containing e. Then $u|_{K_1}(z) = u|_{K_2}(z)$, for all points z on the interior of e, and similarly for all of the partial derivatives of degrees up to m.*

Proof 2.36 *V is polynomial on each triangle K, so continuity at points on the interior of each triangle K is immediate. We just need to check continuity at points on vertices, and points on the interior of edges, which is equivalent to the two parts of the second condition.*

This means that we just need to guarantee that the polynomial functions and their derivatives agree at vertices and edges (similar ideas extend to higher dimensions). We achieve this by assigning nodal variables (and their associated nodal basis functions) appropriately to vertices, edges etc. of each triangle K. First we need to introduce this terminology.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Definition 2.37 (local and global mesh entities) *Let K be a triangle. The local mesh entities of K are the vertices, the edges, and K itself. The global mesh entities of a triangulation $\mathcal{T}$ are the vertices, edges and triangles comprising $\mathcal{T}$.*

Having made this definition, we can now talk about how nodal variables can be assigned to local mesh entities in a geometric decomposition.

Definition 2.38 (local geometric decomposition) *Let $(K, \mathcal{P}, \mathcal{N})$ be a finite element. We say that the finite element has a (local) geometric decomposition if each dual basis function N_i can be associated with a single mesh entity $w \in W$ such that for any $f \in \mathcal{P}$, $N_i(f)$ can be calculated from f and derivatives of f evaluated on w.*

Exercise 2.39 *Consider the finite element defined by:*

1. *K is the unit interval $[0, 1]$*
2. *P is the space of quadratic polynomials on K,*
3. *The nodal variables are:*

$$N_0[v] = v(0), N_1[v] = v(1), N_2[v] = \int_0^1 v(x)\, dx.$$

Find the corresponding nodal basis for P in terms of the monomial basis $\{1, x, x^2\}$. Provide the C^0 geometric decomposition for the finite element (demonstrating that it is indeed C^0).

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

To discuss C^m continuity, we need to introduce some further vocabulary about the topology of K.

Definition 2.40 (closure of a local mesh entity) *Let w be a local mesh entity for a triangle. The closure of w is the set of local mesh entities contained in w (including w itself).*

This allows us to define the degree of continuity of the local geometric decomposition.

Definition 2.41 ((C^m) geometric decomposition) *Let $(K, \mathcal{P}, \mathcal{N})$ be a finite element with geometric decomposition W. We say that W is a C^0 geometric decomposition if, for each local mesh entity w, there exists $\mathcal{N}_w \subset \mathcal{N}$ such that*

1. *All $N \in \mathcal{N}_w$ are associated to elements in the closure of w in W,*
2. *$(w, \mathcal{P}|_w, \mathcal{N}_w)$ is a finite element, where $\mathcal{P}|_w$ is the restriction of $\mathcal{P}$ to w.*

We additionally say that W is a C^1 geometric decomposition if for each local mesh entity w, there exists $\mathcal{N}_w \subset \mathcal{N}$ such that

1. *All $N \in \mathcal{N}_w$ are associated to elements in the closure of w in W,*
2. *$(w, \nabla\mathcal{P}|_w, \mathcal{N}_w)$ is a finite element,*

where $\nabla\mathcal{P}|_w$ is the restriction of $\nabla\mathcal{P}$ to w, and

$$\nabla\mathcal{P} = \{u : u = \nabla v,\ v \in \mathcal{P}\}.$$

This idea extends to C^m finite elements in an analogous way.

The idea behind this definition is that if two triangles K_1 and K_2 are joined at a vertex \$v\$, with finite elements $(K_1, \mathcal{P}_1, \mathcal{N}_1)$ and $(K_2, \mathcal{P}_2, \mathcal{N}_2)$, then the nodal variables $\mathcal{N}_{1,v}$ and $\mathcal{N}_{1,v}$ can be chosen so that f (and for $m = 1$, the derivatives of f) has the same values at v in both K_1 and K_2.

Similarly, if K_1 and K_2 are joined at an edge e, then if the corresponding $\mathcal{N}_{1,e}$ and $\mathcal{N}_{2,e}$ nodal variables associated with that edge agree when applied to u, then u will be C^m continuous through that edge. We just need to define these correspondences.

We explore this definition through a couple of exercises.

Exercise 2.42 *Show that the Lagrange elements of degree k have C^0 geometric decompositions.*

Exercise 2.43 *Show that the Argyris element has a C^1 geometric decomposition.*

A video recording of the following material is available here.

We now use the geometric decomposition to construct global finite element spaces over the whole triangulation (mesh). We just need to define what it means for elements of the nodal variables from the finite elements of two neighbouring triangles to "correspond".

We start by considering spaces of functions that are discontinuous between triangles, before defining C^m continuous subspaces.

Definition 2.44 (Discontinuous finite element space) *Let $\mathcal{T}$ be a triangulation, with finite elements $(K_i, P_i, \mathcal{N}_i)$ for each triangle K_i. The associated discontinuous finite element space V, is defined as*

$$V = \{u : u|_{K_i} \in P_i,\, \forall K_i \in \mathcal{T}\}\,.$$

This defines families of discontinuous finite element spaces.

Example 2.45 (Discontinuous Lagrange finite element space) *Let $\mathcal{T}$ be a triangulation, with Lagrange elements of degree k, $(K_i, P_i, \mathcal{N}_i)$, for each triangle $K_i \in \mathcal{T}$. The corresponding discontinuous finite element space, denoted Pk DG, is called the discontinuous Lagrange finite element space of degree k.*

Next we need to associate each nodal variable in each element to a vertex, edge or triangle of the triangulation $\mathcal{T}_h$, i.e. the global mesh entitles. The following definition explains how to choose this association.

Definition 2.46 (Global (C^m) geometric decomposition) *Let $\mathcal{T}$ be a triangulation with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, each with a C^m geometric decomposition. Assume that for each global mesh entity w, the n_w triangles containing w have finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$ each with M_w dual basis functions associated with w. Further, each of these basis functions can be enumerated $N^w_{i,j} \in \mathcal{N}_i$, $j = 1, \ldots, M_w$, such that $N^w_{1,j}(u|_{K_1}) = N^w_{2,j}(u|_{K_2}) = \ldots = N^w_{n_w,j}(u|_{K_n}), \quad, j = 1, \ldots, M_w$, for all functions $u \in C^m(\Omega)$.*

This combination of finite elements on $\mathcal{T}$ together with the above enumeration of dual basis functions on global mesh entities is called a global C^m geometric decomposition.

Now we use this global C^m geometric decomposition to build a finite element space on the triangulation.

Definition 2.47 (Finite element space from a global (C^m) geometric decomposition) *Let $\mathcal{T}$ be a triangulation with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, each with a C^m geometric decomposition, and let $\hat{V}$ be the corresponding discontinuous finite element space. Then the global C^m geometric decomposition defines a subspace V of $\hat{V}$ consisting of all functions that u satisfy $N^w_{1,j}(u|_{K_1}) = N^w_{2,j}(u|_{K_2}) = \ldots = N^w_{n_w,j}(u|_{K_{n_w}}), \quad j = 1, \ldots, M_w$ for all mesh entities $w \in \mathcal{T}$.*

The following result shows that the global C^m geometric decomposition is a useful definition.

Lemma 2.48 *Let V be a finite element space defined from a global C^m geometric decomposition. Then V is a C^m finite element space.*

Proof 2.49 *From the local C^m decomposition, functions and derivatives up to degree m on vertices and edges are uniquely determined from dual basis elements associated with those vertices and edges, and from the global C^m decomposition, the agreement of dual basis elements means that functions and derivatives up to degree m agree on vertices and edges, and hence the functions are in C^m from Lemma 2.35.*

We now apply this to a few examples, which can be proved as exercises.

Example 2.50 *The finite element space built from the C^0 global decomposition built from degree k Lagrange element is called the degree k continuous Lagrange finite element space, denoted Pk.*

Example 2.51 *The finite element space built from the C^1 global decomposition built from the quintic Argyris element is called the Argyris finite element space.*

In this section, we have built a theoretical toolbox for the construction of finite element spaces. In the next section, we move on to studying how well we can approximate continuous functions as finite element functions.

CHAPTER

THREE

INTERPOLATION OPERATORS

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

In this section we investigate how continuous functions can be approximated by finite element functions. We start locally, looking at a single finite element, and then move globally to function spaces on a triangulation.

3.1 Local and global interpolation operators

Definition 3.1 (Local interpolator) *Given a finite element $(K, \mathcal{P}, \mathcal{N})$, with corresponding nodal basis $\{\phi_i\}_{i=0}^k$. Let v be a function such that $N_i(v)$ is well-defined for all i. Then the local interpolator $\mathcal{I}_K$ is an operator mapping v to $\mathcal{P}$ such that*

$$(I_K v)(x) = \sum_{i=0}^{k} N_i(v)\phi_i(x).$$

We now discuss some useful properties of the local interpolator.

Lemma 3.2 *The operator I_K is linear.*

Exercise 3.3 *Prove Lemma 3.2.*

Lemma 3.4

$$N_i(I_K(v)) = N_i(v),\ \forall\, 0 \leq i \leq k.$$

Exercise 3.5 *Prove Lemma 3.4.*

Lemma 3.6 *I_K is the identity when restricted to $\mathcal{P}$.*

Exercise 3.7 *Prove Lemma 3.6.*

By combining together the local interpolators in each triangle of the triangulation, we obtain the global interpolator into the finite element space.

Definition 3.8 (Global interpolator) *Let V_h be a finite element space constructed from a triangulation $\mathcal{T}_h$ with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, each with a C^m geometric decomposition. The global interpolator $\mathcal{I}_h$ is defined by $\mathcal{I}_h u \in V_h$ such that*

$$\mathcal{I}_h u|_K = I_K u$$

for each $K \in \mathcal{T}_h$.

3.2 Measuring interpolation errors

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Next we look at how well we can approximate continuous functions using the interpolation operator, i.e. we want to measure the approximation error $\mathcal{I}_h u - u$. We are interested in integral formulations, so we want to use integral quantities to measure errors. We have already seen the L^2 norm. It is also useful to take derivatives into account when measuring the error. To discuss higher order derivatives, we introduce the multi-index.

Definition 3.9 (Multi-index.) *For d-dimensional space, a multi-index $\alpha = (\alpha_1, \ldots, \alpha_d)$ assigns the number of partial derivatives in each Cartesian direction. We write $|\alpha| = \sum_{i=1}^d \alpha_i$.*

This means we can write mixed partial derivatives, for example if $\alpha = (1, 2)$ then

$$D^\alpha u = \frac{\partial^3 u}{\partial x \partial y^2}.$$

Now we can define some norms involving derivatives for measuring errors.

Definition 3.10 ((H^k) seminorm and norm) *The H^k seminorm is defined as*

$$|u|_{H^k}^2 = \sum_{|\alpha|=k} \int_\Omega |D^\alpha u|^2 \, dx,$$

where the sum is taken over all multi-indices of size k i.e. all the derivatives are of degree k.

The H^k norm is defined as

$$\|u\|_{H^k}^2 = \sum_{i=0}^k |u|_{H^i}^2.$$

where we conventionally write $|u|_{H^0} = \|u\|_{L^2}$.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

To help to estimate interpolation errors, we quote the following important result (which we will return to much later).

Theorem 3.11 (Sobolev's inequality (for continuous functions)) *Let Ω be an n-dimensional domain with Lipschitz boundary, and let u be a continuous function with k continuous derivatives, i.e. $u \in C^{k,\infty}(\Omega)$. Let k be an integer with $k > n/2$. Then there exists a constant C (depending only on Ω) such that*

$$\|u\|_{C^\infty(\Omega)} = \max_{x\in\Omega} |u(x)| \leq C\|u\|_{H^k(\Omega)}.$$

Proof 3.12 *See a functional analysis course or textbook.*

This is extremely useful because it means that we can measure the H^k norm by integrating and know that it gives an upper bound on the value of u at each point. We say that u is in $C^\infty(\Omega)$ if $\|u\|_{C^\infty(\Omega)} < \infty$, and Sobolev's inequality tells us that this is the case if $\|u\|_{H^k(\Omega)} < \infty$.

This result can be easily extended to derivatives.

Corollary 3.13 (Sobolev's inequality for derivatives (for continuous functions)) *Let Ω be a n-dimensional domain with Lipschitz boundary, and let $u \in C^{k,\infty}(\Omega)$ Let k be an integer with $k - m > n/2$. Then there exists a constant C (depending only on Ω) such that*

$$\|u\|_{C^{m,\infty}(\Omega)} := \sum_{|\alpha|\leq m} \max_{x\in\Omega} |D^\alpha u(x)| \leq C\|u\|_{H^k(\Omega)}.$$

Proof 3.14 *Just apply Sobolev's inequality to the m derivatives of u.*

3.3 Approximation by averaged Taylor polynomials

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

The basic tool for analysing interpolation error for continuous functions is the Taylor series. Rather than taking the Taylor series about a single point, since we are interested in integral quantities, it makes sense to consider an averaged Taylor series over some region inside each cell. This will become important later when we start thinking about more general types of derivative that only exist in an integral sense.

Definition 3.15 (Averaged Taylor polynomial) *Let $\Omega \subset \mathbb{R}^n$ be a domain with diameter d, that is star-shaped with respect to a ball B contained within Ω. For $f \in C^{k,\infty}$ the averaged Taylor polynomial $Q_{k,B}f \in \mathcal{P}_k$ is defined as*

$$Q_{k,B}f(x) = \frac{1}{|B|}\int_B T^k f(y,x)\,dy,$$

where $T^k f$ is the Taylor polynomial of degree k of f,

$$T^k f(y,x) = \sum_{|\alpha|\leq k} D^\alpha f(y)\frac{(x-y)^\alpha}{\alpha!},$$

$$\alpha! = \prod_{i=1}^n \alpha_i!,$$

$$x^\alpha = \prod_{i=1}^n x_i^{\alpha_i}.$$

Exercise 3.16 *Show that*

$$D^\beta Q_{k,B} f = Q_{k-|\beta|,B} D^\beta f,$$

where Q^l_B is the degree l averaged Taylor polynomial of f, and D^β is the β-th derivative where β is a multi-index.

Now we develop an estimate of the error $T^k f - f$.

A first video recording of the following material is available here.

Imperial students can also watch this video on Panopto

A second video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Theorem 3.17 *Let $\Omega \subset \mathbb{R}^n$ be a domain with diameter d, that is star-shaped with respect to a ball B contained within Ω. Then there exists a constant $C(k,n)$ such that for $0 \leq |\beta| \leq k+1$ and all $f \in C^{k+1,\infty}(\Omega)$,*

$$\|D^\beta(f - Q_{k,B}f)\|_{L^2(\Omega)} \leq C\frac{|\Omega|^{1/2}}{|B|^{1/2}} d^{k+1-|\beta|}|f|_{H^{k+1}(\Omega)}.$$

Proof 3.18 *The Taylor remainder theorem (see a calculus textbook) gives*

$$f(x) - T_k f(y,x) = (k+1)\sum_{|\alpha|=k+1}\frac{(x-y)^\alpha}{\alpha!}\int_0^1 D^\alpha f(ty+(1-t)x)t^k\,dt,$$

when $f \in C^{k+1,\infty}$.

Integration over y in B and dividing by $|B|$ gives

$$f(x) - Q_{k,B}f(x) = \frac{k+1}{|B|}\sum_{|\alpha|=k+1}\int_B \frac{(x-y)^\alpha}{\alpha!}\times\int_0^1 D^\alpha f(ty+(1-t)x)t^k\,dt\,dy.$$

Then

$$\int_\Omega |f(x)-Q_{k,B}f(x)|^2\,dx \le C\frac{d^{2(k+1)}}{|B|^2}\sum_{|\alpha|=k+1}\int_\Omega\left(\int_B\int_0^1 |D^\alpha f(ty+(1-t)x)|t^k\,dt\,dy\right)^2\,dx,$$
$$\le C_0\frac{d^{2(k+1)}}{|B|^2}\sum_{|\alpha|=k+1}\int_\Omega\int_B\int_0^1 |D^\alpha f(ty+(1-t)x)|^2\,dt\,dy\int_B\int_0^1 t^{2k}\,dt\,dy\;dx.$$

Then

$$\int_\Omega |f(x)-Q_{k,B}f(x)|^2\,dx \le C_1\frac{d^{2(k+1)}}{|B|^2}\sum_{|\alpha|=k+1}\int_\Omega\int_B\int_0^1 |D^\alpha f(ty+(1-t)x)|^2\,dt\,dy\,dx.$$

We will get the result by changing variables and exchanging the t, y and x integrals. To avoid a singularity when $t=0$ or $t=1$, for each α term we can split the t integral into $[0,1/2]$ and $[1/2,1]$. Call these terms I and II.

Denote by g_α the extension by zero of $D^\alpha f$ to $\mathbb{R}^n$. Then

$$\begin{aligned} I &= \int_B\int_0^{1/2}\int_{\mathbb{R}^n} |g_\alpha(ty+(1-t)x)|^2\,dx\,dt\,dy,\\ &= \int_B\int_0^{1/2}\int_{\mathbb{R}^n} |g_\alpha((1-t)x)|^2\,dx\,dt\,dy,\\ &= \int_B\int_0^{1/2}\int_{\mathbb{R}^n} |g_\alpha(z)|^2(1-t)^{-n}\,dz\,dt\,dy,\\ &\le 2^{n-1}|B|\int_\Omega |D^\alpha f(z)|^2\,dz. \end{aligned}$$

Similarly, for II,

$$\begin{aligned} II &= \int_B\int_{1/2}^{1}\int_{\mathbb{R}^n} |g_\alpha(ty+(1-t)x)|^2\,dx\,dt\,dy,\\ &= \int_B\int_{1/2}^{1}\int_{\mathbb{R}^n} |g_\alpha(ty)|^2\,dx\,dt\,dy,\\ &= \int_B\int_{1/2}^{1}\int_{\mathbb{R}^n} |g_\alpha(z)|^2 t^{-n}\,dz\,dt\,dy,\\ &\le 2^{n-1}|B|\int_\Omega |D^\alpha f(z)|^2\,dz. \end{aligned}$$

Hence, we obtain the required bounds for $|\beta|=0$. For higher derivatives we use the fact that

$$D^\beta Q_{k,B}f(x) = Q_{k-|\beta|,B}D^\beta f(x),$$

which immediately leads to the estimate for $|\beta|>0$.

Now we develop this into an estimate that depends on the diameter of the triangle we are interpolating to.

Corollary 3.19 *Let K_1 be a triangle with diameter 1. There exists a constant $C(k,n)$ such that*

$$\|f-Q_{k,B}f\|_{H^k(K_1)} \le C|f|_{H^{k+1}(K_1)}.$$

Proof 3.20 *Take the maximum over the constants for the derivative contributions of the left-hand side with $d=1$ and use the previous result.*

3.4 Local and global interpolation errors

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

The following exercises give a specific example of the interpolation error results of this section without directly using the previous estimate (because they specialise to L^2, 1D and linear elements).

Exercise 3.21 *Let $w \in C^2([0,1])$, with $w(0) = w(1) = 0$. Show that*

$$\int_0^1 w(x)^2 \, dx \leq c \int_0^1 (w''(x))^2 \, dx.$$

Hints: use the Schwarz inequality,

$$\left(\int_0^1 f(x) g(x) \, dx \right)^2 \leq \left(\int_0^1 f(x)^2 \, dx \right) \left(\int_0^1 g(x)^2 \, dx \right),$$

(which we shall discuss in more generality in Section 4), together with Rolle's theorem.

Exercise 3.22 *Using the previous exercise, show that for all $u \in C^2([0,1])$, there exists a constant c such that*

$$\int_0^1 (u(x) - \mathcal{I}_{[0,1]} u(x))^2 \, dx \leq c \int_0^1 u''(x)^2 \, dx,$$

where $\mathcal{I}_{[0,1]}$ is the interpolator to the finite element with $K = [0,1]$, P is the linear polynomials on K, and the nodal variables are $N_0[p] = p(0)$ and $N_1[p] = p(1)$.

Exercise 3.23 *Using the previous exercise, show that for all $u \in C^2([a,b])$, there exists a constant c such that*

$$\int_a^b (u(x) - \mathcal{I}_{[a,b]} u(x))^2 \, dx \leq c(b-a)^4 \int_a^b u''(x)^2 \, dx,$$

where $\mathcal{I}_{[a,b]}$ is the interpolator to the finite element with $K = [a,b]$, P is the linear polynomials on K, and the nodal variables are $N_0[p] = p(a)$ and $N_1[p] = p(b)$.

Exercise 3.24 *Using the previous exercise, show that for a P1 finite element space defined on the interval $[a,b]$ with maximum mesh cell width h, then there exists a constant c such that*

$$\int_a^b (u(x) - \mathcal{I}_h u(x))^2 \, dx \leq ch^4 \int_a^b u''(x)^2 \, dx,$$

where $\mathcal{I}_h$ is the global nodal interpolator to the P1 finite element space.

Exercise 3.25 *Under the same assumptions as the previous exercise, prove the following finite element version of Sobolev's inequality,*

$$\|v\|_{C^0}^2 \leq C \int_0^1 (v')^2 \, dx,$$

for all $v \in V$, where V is the subspace of the P1 finite element space defined on a subdivision of the interval $[0,1]$ containing only functions v with $v(0) = 0$.

Now we will use the Taylor polynomial estimates to derive error estimates for the local interpolation operator. We start by looking at a triangle with diameter 1, and then use a scaling argument to obtain error estimates in terms of the diameter h. It begins by getting the following bound.

Lemma 3.26 *Let $(K_1, \mathcal{P}, \mathcal{N})$ be a finite element such that K_1 is a triangle with diameter 1, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\|N_i\|_{C^{l,\infty}(K_1)'} < \infty$,*

$$\|N_i\|_{C^{l,\infty}(K_1)'} = \sup_{\|u\|_{C^{l,\infty}(K_1)} > 0} \frac{|N_i(u)|}{\|u\|_{C^{l,\infty}(K_1)}} \qquad \text{(Dual norm of } N_i\text{)}$$

Let $k - l > n/2$, and $u \in C^{k,\infty}(\Omega)$. Then

$$\|\mathcal{I}_{K_1} u\|_{H^k(K_1)} \leq C \|u\|_{H^k(K_1)}.$$

Proof 3.27 *Let* $\{\phi_i\}_{i=1}^n$ *be the nodal basis for* $\mathcal{P}$*. Then*

$$
\begin{aligned}
\|\mathcal{I}_{K_1} u\|_{H^k(K_1)} &\leq \sum_{i=1}^k \|\phi_i\|_{H^k(K_1)} |N_i(u)| \\
&\leq \underbrace{\sum_{i=1}^k \|\phi_i\|_{H^k(K_1)} \|N_i\|_{C^{l,\infty}(K_1)'}}_{C_0} \|u\|_{C^{l,\infty}(K_1)}, \\
&\leq C\|u\|_{H^k(K_1)},
\end{aligned}
$$

where the Sobolev inequality was used in the last line.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Now we can directly apply this to the interpolation operator error estimate on the triangle with diameter 1. It is the standard trick of adding and subtracting something, in this case the Taylor polynomial.

Lemma 3.28 *Let* $(K_1, \mathcal{P}, \mathcal{N})$ *be a finite element such that* K_1 *has diameter* 1*, and such that the nodal variables in* $\mathcal{N}$ *involve only evaluations of functions or evaluations of derivatives of degree* $\leq l$*, and* $\mathcal{P}$ *contain all polynomials of degree* k *and below, with* $k > l + n/2$*. Let* $u \in C^{k+1,\infty}(K_1)$*. Then for* $i \leq k$*, the local interpolation operator satisfies*

$$
|\mathcal{I}_{K_1} u - u|_{H^i(K_1)} \leq C_1 |u|_{H^{k+1}(K_1)}.
$$

Proof 3.29

$$
\begin{aligned}
|\mathcal{I}_{K_1} u - u|^2_{H^i(K_1)} &\leq \|\mathcal{I}_{K_1} u - u\|^2_{H^k(K_1)} \\
&= \|\mathcal{I}_{K_1} u - Q_{k,B} u + Q_{k,B} u - u\|^2_{H^k(K_1)} \\
&\leq \|Q_{k,B} u - u\|^2_{H^k(K_1)} + \|\mathcal{I}(u - Q_{k,B} u)\|^2_{H^k(K_1)}, \\
&\leq \|Q_{k,B} u - u\|^2_{H^k(K_1)} + C^2 \|Q_{k,B} u - u\|^2_{H^k(K_1)}, \\
&\leq (1 + C^2)|u|^2_{H^{k+1}(K_1)},
\end{aligned}
$$

where we used the fact that $\mathcal{I}_{K_1} Q_{k,B} u = Q_{k,B} u$ *in the second line and the previous lemma in the third line.*

A video recording of the following material is available here.

Now we apply a scaling argument to translate this to triangles with diameter h.

Lemma 3.30 *Let* $(K, \mathcal{P}, \mathcal{N})$ *be a finite element such that* K *has diameter* d*, and such that the nodal variables in* $\mathcal{N}$ *involve only evaluations of functions or evaluations of derivatives of degree* $\leq l$*, and* $\mathcal{P}$ *contains all polynomials of degree* k *and below, with* $k > l + n/2$*. Let* $u \in C^{k+1,\infty}(K)$*. Then for* $i \leq k$*, the local interpolation operator satisfies*

$$
|\mathcal{I}_K u - u|_{H^i(K)} \leq C_K d^{k+1-i} |u|_{H^{k+1}(K)}.
$$

where C_K *is a constant that depends on the shape of* K *but not the diameter.*

Proof 3.31 *Consider the change of variables* $x \to \phi(x) = x/d$*. This map takes* K *to* K_1 *with diameter 1. Then*

$$
\begin{aligned}
\int_K |D^\beta(I_K u - u)|^2 \, dx &= d^{-2|\beta|+n} \int_{K_1} |D^\beta(I_{K_1} u \circ \phi - u \circ \phi)|^2 \, dx, \\
&\leq C_1^2 d^{-2|\beta|+n} \sum_{|\alpha|=k+1} \int_{K_1} |D^\alpha u \circ \phi|^2 \, dx, \\
&\leq C_1^2 d^{-2|\beta|+2(k+1)} \sum_{|\alpha|=k+1} \int_K |D^\alpha u|^2 \, dx, \\
&= C_1^2 d^{2(-|\beta|+k+1)} |u|^2_{H^{k+1}(K)},
\end{aligned}
$$

and taking the square root gives the result.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

So far we have just developed an error estimate for the local interpolator on a single triangle. Now we extend this to finite element spaces defined on the whole triangulation.

Theorem 3.32 *Let $\mathcal{T}$ be a triangulation of Ω with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, such that the minimum aspect ratio γ of the triangles K_i satisfies $\gamma > 0$, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\mathcal{P}$ contains all polynomials of degree k and below, with $k > l + n/2$. Let $u \in C^{k+1,\infty}(K_1)$. Let h be the maximum over all of the triangle diameters, with $0 \leq h < 1$. Then for $i \leq k$, the global interpolation operator satisfies*

$$\|\mathcal{I}_h u - u\|_{H^i(\Omega)} \leq C h^{k+1-i} |u|_{H^{k+1}(\Omega)}.$$

(Recalling that we use the "broken" finite element derivative in norms for $\mathcal{I}_h u$ over Ω.

Proof 3.33

$$\begin{aligned}
\|\mathcal{I}_h u - u\|^2_{H^i(\Omega)} &= \sum_{K \in \mathcal{T}} \|\mathcal{I}_K u - u\|^2_{H^i(K)}, \\
&\leq \sum_{K \in \mathcal{T}} C_K d_K^{2(k+1-i)} |u|^2_{H^{k+1}(K)}, \\
&\leq C_{\max} h^{2(k+1-i)} \sum_{K \in \mathcal{T}} |u|^2_{H^{k+1}(K)}, \\
&= C_{\max} h^{2(k+1-i)} |u|^2_{H^{k+1}(\Omega)},
\end{aligned}$$

where the existence of the $C_{\max} = \max_K C_K < \infty$ is due to the lower bound in the aspect ratio.

In this section, we have built a theoretical toolbox for the interpolation of functions to finite element spaces. In the next section, we move on to studying the solveability of finite element approximations.

CHAPTER

FOUR

FINITE ELEMENT PROBLEMS: SOLVABILITY AND STABILITY

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

In section 1, we saw the example of a finite element approximation for Poisson's equation in the unit square, which we now recall below.

Definition 4.1 *The finite element approximation* $u_h \in \mathring{V}_h$ *to the solution* u_h *of Poisson's equation is defined by*

$$\int_\Omega \nabla v \cdot \nabla u_h \, dx = \int_\Omega v f \, dx, \quad \forall v \in \mathring{V}_h. \tag{4.1}$$

A fundamental question is whether the solution u_h exists and is unique. This question is of practical interest because if these conditions are not satisfied, then the matrix-vector system for the basis coefficients of u_h will not be solvable. To answer this question for this approximation (and others for related equations), we will use some general mathematical machinery about linear problems defined on Hilbert spaces. It will turn out that this machinery will also help us show that the approximation u_h converges to the exact solution u (and in what sense).

4.1 Finite element spaces and other Hilbert spaces

In the previous sections, we introduced the concept of finite element spaces, which contain certain functions defined on a domain Ω. Finite element spaces are examples of vector spaces (hence the use of the word "space").

Definition 4.2 (Vector space) *A vector space over the real numbers* $\mathbb{R}$ *is a set* V*, with an addition operator* $+ : V \times V \to V$*, plus a scalar multiplication operator* $\times : \mathbb{R} \times V \to V$*, such that:*

1. **There exists a unique zero element** $e \in V$ **such that:**
   - $k \times e = e$ *for all* $k \in \mathbb{R}$,
   - $0 \times v = e$ *for all* $v \in V$,
   - $e + v = v$ *for all* $v \in V$.
2. V **is closed under addition and multiplication, i.e.,**
   $a \times u + v \in V$ *for all* $u, v \in V$, $a \in \mathbb{R}$.

Lemma 4.3 *Let* V *be a finite element space. Then* V *is a vector space.*

Proof 4.4 *First, we note that the zero function* $u(x) := 0$ *is in* V*, and satisfies the above properties. Further, let* $u, v \in V$*, and* $a \in \mathbb{R}$*. Then, when restricted to each triangle* K_i*,* $u + av \in P_i$*. Also, for each shared mesh entity, the shared nodal variables agree between triangles. Therefore,* $u + av \in V$*.*

We now introduce bilinear forms on vector spaces. Bilinear forms are important because they will represent the left hand side of finite element approximations of linear PDEs.

Definition 4.5 (Bilinear form) *A bilinear form* $b(\cdot, \cdot)$ *on a vector space* V *is a mapping* $b : V \times V \to \mathbb{R}$*, such that*

1. $v \to b(v, w)$ *is a linear map in* v *for all* w*.*

2. $v \to b(w, v)$ *is a linear map in* v *for all* w.

It is a symmetric bilinear form if in addition, $b(v, w) = b(w, v)$, *for all* $v, w \in V$.

Here are two important examples of bilinear forms on finite element spaces.

Example 4.6 *Let* V_h *be a finite element space. The following are bilinear forms on* V_h,

$$b(u, v) = \int_\Omega uv \, dx,$$
$$b(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx.$$

To turn a vector space into a Hilbert space, we need to select an inner product.

Definition 4.7 (Inner product) *A real inner product, denoted by* $(\cdot, \cdot)$, *is a symmetric bilinear form on a vector space* V *with*

1. $(v, v) \geq 0 \, \forall v \in V$,
2. $(v, v) = 0 \iff v = 0$.

This enables the following definition.

Definition 4.8 (Inner product space) *We call a vector space* $(V, (\cdot, \cdot))$ *equipped with an inner product an inner product space.*

We now introduce two important examples of inner products for finite element spaces.

Definition 4.9 ((L^2) inner product) *Let* f, g *be two functions in* $L^2(\Omega)$. *The* L^2 *inner product between* f *and* g *is defined as*

$$(f, g)_{L^2} = \int_\Omega fg \, dx.$$

The L^2 inner product satisfies condition 2 provided that we understand functions in L^2 as being equivalence classes of functions under the relation $f \equiv g \iff \int_\Omega (f - g)^2 \, dx = 0$.

Definition 4.10 ((H^1) inner product) *Let* f, g *be two* C^0 *finite element functions. The* H^1 *inner product between* f *and* g *is defined as*

$$(f, g)_{H^1} = \int_\Omega fg + \nabla f \cdot \nabla g \, dx.$$

The H^1 inner product satisfies condition 2 since

$$(f, f)_{L^2} \leq (f, f)_{H^1}.$$

The Schwarz inequality is a useful tool for bounding the size of inner products.

Theorem 4.11 (Schwarz inequality) *If* $(V, (\cdot, \cdot))$ *is an inner product space, then*

$$|(u, v)| \leq (u, u)^{1/2} (v, v)^{1/2}.$$

Equality holds if and only if $u = \alpha v$ *for some* $\alpha \in \mathbb{R}$.

Proof 4.12 *See a course on vector spaces.*

Our solvability conditions will make use of norms that measure the size of elements of a vector space (the size of finite element functions, in our case).

Definition 4.13 (Norm) *Given a vector space* V, *a norm* $\|\cdot\|$ *is a function from* V *to* $\mathbb{R}$, *with*

1. $\|v\| \geq 0$, $\forall v \in V$,
2. $\|v\| = 0 \iff v = 0$,
3. $\|cv\| = |c| \|v\| \, \forall c \in \mathbb{R}, v \in V$,

4. $\|v + w\| \leq \|v\| + \|w\|$.

For inner product spaces, there is a natural choice of norm.

Lemma 4.14 *Let $(V, (\cdot,\cdot))$ be an inner product space. Then $\|v\| = \sqrt{(v,v)}$ defines a norm on V.*

Proof 4.15 *From bilinearity we have*

$$\|\alpha v\| = \sqrt{(\alpha v, \alpha v)} = \sqrt{\alpha^2 (v,v)} = |\alpha|\|v\|,$$

hence property 3.

$\|v\| = (v,v)^{1/2} \geq 0$, *hence property 1.*

If $0 = \|v\| = (v,v)^{1/2} \implies (v,v) = 0 \implies v = 0$, *hence property 2.*

We finally check the triangle inequality (property 4).

$$\begin{aligned}
\|u + v\|^2 &= (u + v, u + v) \\
&= (u,u) + 2(u,v) + (v,v) \\
&= \|u\|^2 + 2(u,v) + \|v\|^2 \\
&\leq \|u\|^2 + 2\|u\|\|v\| + \|v\|^2 \quad \textit{[Schwarz]}, \\
&= (\|u\| + \|v\|)^2,
\end{aligned}$$

hence $\|u + v\| \leq \|u\| + \|v\|$.

We introduce the following useful term.

Definition 4.16 (Normed space) *A vector space V with a norm $\|\cdot\|$ is called a normed vector space, written $(V, \|\cdot\|)$.*

To finish our discussion of Hilbert spaces, we need to review the concept of completeness (which you might have encountered in an analysis course). This seems not so important since finite element spaces are finite dimensional, but later we shall consider sequences of finite element spaces with smaller and smaller triangles, where completeness becomes important.

Completeness depends on the notion of a Cauchy sequence.

Definition 4.17 (Cauchy sequence) *A Cauchy sequence on a normed vector space $(V, \|\cdot\|)$ is a sequence $\{v_i\}_{i=1}^{\infty}$ satisfying $\|v_j - v_k\| \to 0$ as $j, k \to \infty$.*

This definition leads to the definition of completeness.

Definition 4.18 (Complete normed vector space) *A normed vector space $(V, \|\cdot\|)$ is complete if all Cauchy sequences have a limit $v \in V$ such that $\|v - v_j\| \to 0$ as $j \to \infty$.*

Finally, we reach the definition of a Hilbert space.

Definition 4.19 (Hilbert space) *An inner product space $(V, (\cdot,\cdot))$ is a Hilbert space if the corresponding normed space $(V, \|\cdot\|)$ is complete.*

All finite dimensional normed vector spaces are complete. Hence, C^0 finite element spaces equipped with L^2 or H^1 inner products are Hilbert spaces. Later we shall understand our finite element spaces as subspaces of infinite dimensional Hilbert spaces.

4.2 Linear forms on Hilbert spaces

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We will now build some structures on Hilbert spaces that allow us to discuss variational problems on them, which includes finite element approximations such as the Poisson example discussed so far.

Linear functionals are important as they will represent the right-hand side of finite element approximations of PDEs.

Definition 4.20 (Continuous linear functional) *Let H be a Hilbert space with norm $|\cdot|_H$.*

1. *A functional L is a map from H to $\mathbb{R}$.*
2. *A functional $L : H \to \mathbb{R}$ is linear if $u, v \in H$, $\alpha \in \mathbb{R} \implies L(u + \alpha v) = L(u) + \alpha L(v)$.*
3. *A functional $L : H \to \mathbb{R}$ is continuous if there exists $C > 0$ such that*

$$|L(u) - L(v)| \le C\|u - v\|_H \quad \forall u, v \in H.$$

It is important that linear functionals are “nice” in the following sense.

Definition 4.21 (Bounded functional) *A functional $L : H \to \mathbb{R}$ is bounded if there exists $C > 0$ such that*

$$|L(u)| \le C\|u\|_H, \quad \forall u \in H.$$

For linear functionals we have the following relationship between boundedness and continuity.

Lemma 4.22 *Let $L : H \to \mathbb{R}$ be a linear functional. Then L is continuous if and only if it is bounded.*

Proof 4.23 *L bounded $\implies L(u) \le C\|u\|_H \implies |L(u) - L(v)| = |L(u - v)| \le C\|u - v\|_H \, \forall u, v \in H$, i.e. L is continuous.*

On the other hand, L continuous $\implies |L(u - v)| \le C\|u - v\|_H \quad \forall u, v \in H$. Pick $v = 0$, then $|L(u)| = |L(u - 0)| \le C\|u - 0\|_H = C\|u\|_H$, i.e. L is bounded.

We can also interpret bounded linear functionals as elements of a vector space.

Definition 4.24 (Dual space) *Let H be a Hilbert space. The dual space H' is the space of continuous (or bounded) linear functionals $L : H \to \mathbb{R}$.*

This dual space can also be equipped with a norm.

Definition 4.25 (Dual norm) *Let L be a continuous linear functional on H, then*

$$\|L\|_{H'} = \sup_{0 \ne v \in H} \frac{L(v)}{\|v\|_H}.$$

There is a simple mapping from H to H'.

Lemma 4.26 *Let $u \in H$. Then the functional $L_u : H \to \mathbb{R}$ defined by*

$$L_u(v) = (u, v), \quad \forall v \in H,$$

is linear and continuous.

Proof 4.27 *For $v, w \in H$, $\alpha \in \mathbb{R}$ we have*

$$L_u(v + \alpha w) = (u, v + \alpha w) = (u, v) + \alpha(u, w) = L_u(v) + \alpha L_u(w).$$

Hence L_u is linear.

We see that L_u is bounded by Schwarz inequality,

$$|L_u(v)| = |(u, v)| \le C\|v\|_H \text{ with } C = \|u\|_H.$$

The following famous theorem states that the converse is also true.

Theorem 4.28 (Riesz representation theorem) *For any continuous linear functional L on H there exists $u \in H$ such that*

$$L(v) = (u, v) \quad \forall v \in H.$$

Further,

$$\|u\|_H = \|L\|_{H'}.$$

Proof 4.29 *See a course or textbook on Hilbert spaces.*

4.3 Variational problems on Hilbert spaces

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We will consider finite element methods that can be formulated in the following way.

Definition 4.30 (Linear variational problem) *Let $b(u, v)$ be a bilinear form on a Hilbert space V, and F be a linear form on V. This defines a linear variational problem: find $u \in V$ such that*

$$b(u, v) = F(v), \quad \forall v \in V.$$

We now discuss some important examples from finite element discretisations of linear PDEs.

Example 4.31 ((Pk) discretisation of (modified) Helmholtz problem with Neumann bcs) *For some known function f,*

$$b(u, v) = \int_\Omega uv + \nabla u \cdot \nabla v \, dx,$$
$$F(v) = \int_\Omega vf \, dx,$$

and V is the Pk continuous finite element space on a triangulation of Ω.

Example 4.32 ((Pk) discretisation of Poisson equation with partial Dirichlet bcs) *For some known function f,*

$$b(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx,$$
$$F(v) = \int_\Omega vf \, dx,$$

and V is the subspace of the Pk continuous finite element space on a triangulation of Ω such that functions vanishes on $\Gamma_0 \subseteq \partial\Omega$.

Example 4.33 ((Pk) discretisation of Poisson equation with pure Neumann bcs) *For some known function f,*

$$b(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx,$$
$$F(v) = \int_\Omega vf \, dx,$$

and V is the subspace of the Pk continuous finite element space on a triangulation of Ω such that functions satisfy

$$\int_\Omega u \, dx = 0.$$

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We now introduce two important properties of bilinear forms that determine whether a linear variational problem is solvable or not. The first is continuity.

Definition 4.34 (Continuous bilinear form) *A bilinear form is continuous on a Hilbert space V if there exists a constant $0 < M < \infty$ such that*

$$|b(u, v)| \leq M\|u\|_V\|v\|_V.$$

The second is coercivity.

Definition 4.35 (Coercive bilinear form) *A bilinear form is coercive on a Hilbert space V if there exists a constant $0 < \gamma < \infty$ such that*

$$|b(u,u)| \geq \gamma \|u\|_V \|u\|_V.$$

These two properties combine in the following theorem providing sufficient conditions for existence and uniqueness for solutions of linear variational problems.

Theorem 4.36 (Lax-Milgram theorem) *Let b be a bilinear form, F be a linear form, and $(V, \|\cdot\|)$ be a Hilbert space. If b is continuous and coercive, and F is continuous, then a unique solution $u \in V$ to the linear variational problem exists, with*

$$\|u\|_V \leq \frac{1}{\gamma}\|F\|_{V'}.$$

Proof 4.37 *See a course or textbook on Hilbert spaces.*

We are going to use this result to show solvability for finite element discretisations. In particular, we also want to know that our finite element discretisation continues to be solvable as the maximum triangle edge diameter h goes to zero. This motivates the following definition.

Definition 4.38 (Stability) *Consider a sequence of triangulations $\mathcal{T}_h$ with corresponding finite element spaces V_h labelled by a maximum triangle diameter h, applied to a variational problem with bilinear form $b(u,v)$ and linear form L. For each V_h we have a corresponding coercivity constant γ_h.*

If $\gamma_h \to \gamma > 0$, and $\|F\|_{V_h'} \to c < \infty$, then we say that the finite element discretisation is stable.

With this in mind it is useful to consider h-independent definitions of $\|\cdot\|_V$ (such as the L^2 and H^1 norms), which is why we introduced them.

4.4 Solvability and stability of some finite element discretisations

In this section we will introduce some tools for showing coercivity and continuity of bilinear forms, illustrated with finite element approximations of some linear PDEs where they may be applied.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We start with the simplest example, for which continuity and coercivity are immediate.

Theorem 4.39 (Solving the (modified) Helmholtz problem) *Let b, L be the forms from the Helmholtz problem, with $\|f\|_{L^2} < \infty$. Let V_h be a Pk continuous finite element space defined on a triangulation $\mathcal{T}$. Then the finite element approximation u_h exists and the discretisation is stable in the H^1 norm.*

Proof 4.40 *First we show continuity of F. We have*

$$F(v) = \int_\Omega f v \, dx \leq \|f\|_{L^2}\|v\|_{L^2} \leq \|f\|_{L^2}\|v\|_{H^1},$$

since $\|v\|_{L^2} \leq \|v\|_{H^1}$.

Next we show continuity of b.

$$|b(u,v)| = |(u,v)_{H^1}| \leq 1 \times \|u\|_{H^1}\|v\|_{H^1},$$

from the Schwarz inequality of the H^1 inner product. Finally we show coercivity of b.

$$b(u,u) = \|u\|_{H^1}^2 \geq 1 \times \|u\|_{H^1}^2,$$

The continuity and coercivity constants are both 1, independent of h, so the discretisation is stable.

Exercise 4.41 *Let V be a C^0 finite element space on $[0,1]$, defined on a one-dimensional mesh with vertices $0 = x_0 < x_1 < x_2 < \ldots < x_{n-1} < x_n = 1$. Show that $u \in V$ satisfies the fundamental theorem of calculus, i.e.*

$$\int_0^1 u' \, dx = u[1] - u[0],$$

where u' is the usual finite element derivative defined in $L^2([0,1])$ by taking the usual derivative when restricting u to any subinterval $[x_k, x_{k+1}]$.

Exercise 4.42 *Let*

$$a(u,v) = \int_0^1 \left(u'v' + u'v + uv\right) \, dx.$$

Let V be a C^0 finite element space on $[0,1]$ and let $\mathring{V}$ be the subspace of functions that vanish at $x=0$ and $x=1$. Using the finite element version of the fundamental theorem of calculus above, prove that

$$a(v,v) = \int_0^1 \left((v')^2 + v^2\right) \, dx := \|v\|_{H^1}^2, \quad \forall v \in \mathring{V}.$$

Hence conclude that the bilinear form is coercive on $\mathring{V}$.

Exercise 4.43 *Consider the variational problem with bilinear form*

$$a(u,v) = \int_0^1 (u'v' + u'v + uv) \, dx,$$

corresponding to the differential equation

$$-u'' + u' + u = f.$$

Prove that $a(\cdot,\cdot)$ is continuous and coercive on a C^0 finite element space V defined on $[0,1]$, with respect to the H^1 inner product.

Hints: for continuity, just use the triangle inequality and the relationship between L^2 and H^1 norms. For coercivity, try completing the square for the integrand in a.

For the Helmholtz problem, we have

$$b(u,v) = \int_\Omega uv + \nabla u \cdot \nabla v \, dx = (u,v)_{H^1},$$

i.e. $b(u,v)$ is the H^1 inner product of u and v, which makes the continuity and coercivity immediate.

For the Poisson problem, we have

$$b(u,u) = \int_\Omega |\nabla u|^2 \, dx = |u|_{H^1}^2 \neq \|u\|_{H^1}^2,$$

where we recall the H^1 seminorm from the interpolation section. Some additional results are required to show coercivity, as $b(u,u)$ is not the H^1 norm squared any more. A seminorm has all the properties of a norm except $|u| = 0 \nRightarrow u = 0$, which is precisely what is needed in the Lax-Milgram theorem.

Exercise 4.44 *Let $\mathcal{T}_h$ be a triangulation on the 1×1 unit square domain Ω, and let V be a C^0 Lagrange finite element space of degree k defined on $\mathcal{T}_h$. A finite element discretisation for the Poisson equation with Neumann boundary conditions is given by: find $u \in V$ such that*

$$\int_\Omega \nabla v \cdot \nabla u \, dx = \int_\Omega vf \, dx, \quad \forall v \in V,$$

for some known function f. Show that the bilinear form for this problem is not coercive in V.

For the Poisson problem, coercivity comes instead from the following mean estimate.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Lemma 4.45 (Mean estimate for finite element spaces) *Let u be a member of a C^0 finite element space, and define*

$$\bar{u} = \frac{\int_\Omega u \, dx}{\int_\Omega \, dx}.$$

Then there exists a positive constant C, independent of the triangulation but dependent on (convex) Ω, such that

$$\|u - \bar{u}\|_{L^2} \leq C|u|_{H^1}.$$

A video recording of the first part of the following material is available here.

Imperial students can also watch this video on Panopto

A video recording of the second part of section is available here.

Imperial students can also watch this video on Panopto

Proof 4.46 *(Very similar to the proof of the estimate for averaged Taylor polynomials.)*

Let x and y be two points in Ω. We note that $f(s) = u(y + s(x - y))$ is a C^0, piecewise polynomial function of s. Let $s_0 = 0 < s_1 < s_2 < \ldots < s_{k-1} < s_k = 1$ denote the points where $y + s(x - y)$ intersects a triangle edge or vertex. Then f is a continuous function when restricted to each interval $[s_i, s_{i+1}]$, $i = 0, \ldots, k-1$. This means that

$$\begin{aligned} f(s_{i+1}) - f(s_i) &= \int_{s_i}^{s_{i+1}} f'(s) \, ds \\ &= \int_{s_i}^{s_{i+1}} (x - y) \cdot \nabla u(y + s(x - y)) \, ds, \end{aligned}$$

where ∇u is the finite element derivative of u. Summing this up from $i = 0$ to $i = k - 1$, we obtain

$$u(x) = u(y) + \int_0^1 (x - y) \cdot \nabla u(y + s(x - y)) \, ds.$$

Then

$$\begin{aligned} u(x) - \bar{u} &= \frac{1}{|\Omega|} \int_\Omega u(x) - u(y) \, dy \\ &= \frac{1}{|\Omega|} \int_\Omega (x - y) \cdot \int_{s=0}^1 \nabla u(y + s(x - y)) \, ds \, dy, \end{aligned}$$

Therefore

$$\begin{aligned} \|u - \bar{u}\|_{L^2(\Omega)}^2 &= \frac{1}{|\Omega|^2} \int_\Omega \left(\int_\Omega (x - y) \cdot \int_{s=0}^1 \nabla u(y + s(x - y)) \, ds \, dy \right)^2 \, dx, \\ &\leq \frac{1}{|\Omega|^2} \int_\Omega \int_\Omega |x - y|^2 \, dy \int_\Omega \int_{s=0}^1 |\nabla u(y + s(x - y))|^2 \, ds \, dy \, dx, \\ &\leq C \int_\Omega \int_\Omega \int_{s=0}^1 |\nabla u(y + s(x - y))|^2 \, ds \, dy \, dx. \end{aligned}$$

We split this final quantity into two parts (to avoid singularities),

$$\|u - \bar{u}\|_{L^2(\Omega)}^2 \leq C(I + II),$$

where

$$
\begin{aligned}
I &= \int_\Omega \int_{s=0}^{1/2} \int_\Omega |\nabla u(y + s(x-y))|^2 \, dy \, ds \, dx, \\
II &= \int_\Omega \int_{s=1/2}^{2} \int_\Omega |\nabla u(y + s(x-y))|^2 \, dx \, ds \, dy,
\end{aligned}
$$

which we will now estimate separately.

To evaluate I, change variables $y \to y' = y + s(x-y)$, defining $\Omega'_s \subset \Omega$ as the image of Ω under this transformation. Then,

$$
\begin{aligned}
I &= \int_\Omega \int_{s=0}^{1/2} \frac{1}{(1-s)^2} \int_{\Omega'_s} |\nabla u(y')|^2 \, dy' \, ds \, dx, \\
&\leq \int_\Omega \int_{s=0}^{1/2} \frac{1}{(1-s)^2} \int_\Omega |\nabla u(y')|^2 \, dy' \, ds \, dx, \\
&= \frac{|\Omega|}{2} |\nabla u|^2_{H^1(\Omega)}.
\end{aligned}
$$

To evaluate II, change variables $x \to x' = y + s(x-y)$, defining $\Omega'_s \subset \Omega$ as the image of Ω under this transformation. Then,

$$
\begin{aligned}
II &= \int_\Omega \int_{s=1/2}^{2} \frac{1}{s^2} \int_{\Omega'_s} |\nabla u(x')|^2 \, dx' \, ds \, dy, \\
&\leq \int_\Omega \int_{s=0}^{1/2} \frac{1}{s^2} \int_\Omega |\nabla u(x')|^2 \, dx' \, ds \, dy, \\
&= |\Omega| |\nabla u|^2_{H^1(\Omega)}.
\end{aligned}
$$

Combining,

$$
\|u - \bar{u}\|^2_{L^2(\Omega)} \leq C(I + II) = \frac{3C|\Omega|}{2} |u|^2_{H^1(\Omega)},
$$

which has the required form.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

The mean estimate can now be used to show solvability for the Poisson problem with pure Neumann conditions.

Theorem 4.47 (Solving the Poisson problem with pure Neumann conditions) *Let b, L, V, be the forms for the pure Neumann Poisson problem, with $\|f\|_{L^2} < \infty$. Let V_h be a Pk continuous finite element space defined on a triangulation $\mathcal{T}$, and define*

$$
\bar{V}_h = \{u \in V_h : \bar{u} = 0\}.
$$

Then for $\bar{V}_h$, the finite element approximation u_h exists and the discretisation is stable in the H^1 norm.

Proof 4.48 *Using the mean estimate, for $u \in \bar{V}_h$, we have*

$$
\|u\|^2_{L^2} = \|u - \underbrace{\bar{u}}_{=0}\|^2_{L^2} \leq C^2 |u|^2_{H^1}.
$$

Hence we obtain the coercivity result,

$$
\|u\|^2_{H^1} = \|u\|^2_{L^2} + |u|^2_{H^1} \leq (1 + C^2)|u|^2_{H^1} = (1 + C^2) b(u, u).
$$

Continuity follows from Schwarz inequality,

$$
|b(u, v)| \leq |u|_{H^1} |v|_{H^1} \leq \|u\|_{H^1} \|v\|_{H^1}.
$$

The coercivity constant is independent of h, so the approximation is stable.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Proving the coercivity for the Poisson problem with Dirichlet or partial Dirichlet boundary conditions requires some additional results. We start by showing that the divergence theorem also applies to finite element derivatives of C^0 finite element functions.

Lemma 4.49 (Finite element divergence theorem) *Let ϕ be a C^1 vector-valued function. and $u \in V$ be a member of a C^0 finite element space. Then*

$$\int_\Omega \nabla \cdot (\phi u)\, dx = \int_{\partial\Omega} \phi \cdot n u \, dS,$$

where n is the outward pointing normal to $\partial\Omega$.

Proof 4.50

$$\begin{aligned}\int_\Omega \nabla \cdot (\phi u)\, dx &= \sum_{K\in\mathcal{T}} \int_K \nabla \cdot (\phi u)\, dx,\\ &= \sum_{K\in\mathcal{T}} \int_{\partial K} \phi \cdot n_K u \, dS,\\ &= \int_{\partial\Omega} \phi \cdot n u\, dS + \underbrace{\int_\Gamma \phi \cdot (n^+ + n^-) u \, dS}_{=0}.\end{aligned}$$

This allows us to prove the finite element trace theorem, which relates the H^1 norm of a C^0 finite element function to the L^2 norm of the function restricted to the boundary.

Theorem 4.51 (Trace theorem for continuous finite elements) *Let V_h be a continuous finite element space, defined on a triangulation $\mathcal{T}$, on a polygonal domain Ω. Then*

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

where C is a constant that depends only on the geometry of Ω.

Proof 4.52 *The first step is to construct a C^1 function ξ satisfying $\xi \cdot n = 1$ on Ω. We do this by finding a triangulation $\mathcal{T}_0$ (unrelated to $\mathcal{T}$), and defining an C^1 Argyris finite element space V_0 on it. We then choose ξ so that both Cartesian components are in V_0, satisfying the boundary condition.*

Then,

$$\begin{aligned}\|u\|^2_{L^2(\partial\Omega)} &= \int_{\partial\Omega} u^2\, dS = \int_{\partial\Omega} \xi \cdot n u^2 \, dS,\\ &= \int_\Omega \nabla \cdot (\xi u^2)\, dx,\\ &= \int_\Omega u^2 \nabla \cdot \xi + 2u\xi \cdot \nabla u \, dx,\\ &\leq \|u\|^2_{L^2} \|\nabla \cdot \xi\|_\infty + 2|\xi|_\infty \|u\|_{L^2} |u|_{H^1},\end{aligned}$$

So,

$$\begin{aligned}\|u\|^2_{L^2(\partial\Omega)} &\leq \|u\|^2_{L^2} \|\nabla \cdot \xi\|_\infty + |\xi|_\infty \left(\|u\|^2_{L^2} + |u|^2_{H^1}\right),\\ &\leq C\|u\|^2_{H^1},\end{aligned}$$

where we have used the geometric-arithmetic mean inequality $2ab \leq a^2 + b^2$.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We can now use the trace inequality to estabilish solvability for the Poisson problem with (full or partial) Dirichlet conditions.

Theorem 4.53 (Solving the Poisson problem with partial Dirichlet conditions) *Let b, L, V, be the forms for the (partial) Dirichlet Poisson problem, with $\|f\|_{L^2} < \infty$. Let V_h be a Pk continuous finite element space defined on a triangulation $\mathcal{T}$, and define*

$$\mathring{V}_h = \{u \in V_h : u|_{\Gamma_0}\}.$$

Then for $\mathring{V}_h$, the finite element approximation u_h exists and the discretisation is stable in the H^1 norm.

Proof 4.54 *[Proof taken from Brenner and Scott]. We have*

$$\begin{aligned}\|v\|_{L^2(\Omega)} &\leq \|v-\bar{v}\|_{L^2(\Omega)} + \|\bar{v}\|_{L^2(\Omega)}, \\ &\leq C|v|_{H^1(\Omega)} + \frac{|\Omega|^{1/2}}{|\Gamma_0|}\left|\int_{\Gamma_0} \bar{v}\, dS\right|, \\ &= C|v|_{H^1(\Omega)} + \frac{|\Omega|^{1/2}}{|\Gamma_0|}\left(\left|\int_{\Gamma_0} \bar{v} - \underbrace{v}_{=0}\, dS\right|\right).\end{aligned}$$

We have

$$\begin{aligned}\left|\int_{\Gamma_0}(v-\bar{v})\, ds\right| = \left|\int_{\Gamma_0} 1\times(v-\bar{v})\, ds\right| &\leq |\Gamma_0|^{1/2}\|v-\bar{v}\|_{L^2(\partial\Omega)}, \\ &\leq |\Gamma_0|^{1/2}C|v|_{H^1(\Omega)}.\end{aligned}$$

Combining, we get

$$\|v\|_{L^2(\Omega)} \leq C_1|v|_{H^1(\Omega)},$$

and hence coercivity,

$$\|v\|^2_{H^1(\Omega)} \leq (1+C_1^2)b(v,v).$$

The coercivity constant is independent of h, so the approximation is stable.

Exercise 4.55 *For $f \in L^2(\Omega)$, $\sigma \in C^1(\Omega)$, find a finite element formulation of the problem*

$$-\sum_{i=1}^{n}\frac{\partial}{\partial x_i}\left(\sigma(x)\frac{\partial u}{\partial x_i}\right) = f, \quad \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega.$$

If there exist $0 < a < b$ such that $a < \sigma(x) < b$ for all $x \in \Omega$, show continuity and coercive for your formulation with respect to the H^1 norm.

Exercise 4.56 *Find a C^0 finite element formulation for the Poisson equation*

$$-\nabla^2 u = f, \quad u = g \text{ on } \partial\Omega,$$

for a function g which is C^2 and whose restriction to $\partial\Omega$ is in $L^2(\partial\Omega)$. Derive conditions under the discretisation has a unique solution.

In this section, We have developed some techniques for showing that variational problems arising from finite element discretisations for Helmholtz and Poisson problems have unique solutions, that are stable in the H^1-norm. This means that we can be confident that we can solve the problems on a computer and the solution won't become singular as the mesh is refined. Now we would like to go further and ask what is happening to the numerical solutions as the mesh is refined. What are they converging to?

We will address these questions in the next section.

CHAPTER

FIVE

CONVERGENCE OF FINITE ELEMENT APPROXIMATIONS

In this section we develop tools to prove convergence of finite element approximations to the exact solutions of PDEs.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

5.1 Weak derivatives

Consider a triangulation $\mathcal{T}$ with recursively refined triangulations $\mathcal{T}_h$ and corresponding finite element spaces V_h. Given stable finite element variational problems, we have a sequence of solutions u_h as $h \to 0$, satisfying the h-independent bound

$$\|u_h\|_{H^1(\Omega)} \leq C.$$

What are these solutions converging to? We need to find a Hilbert space that contains all V_h as $h \to 0$, that extends the H^1 norm to the $h \to 0$ limit of finite element functions.

Our first task is to define a derivative that works for all finite element functions, without reference to a mesh. This requires some preliminary definitions, starting by considering some very smooth functions that vanish on the boundaries together with their derivatives (so that we can integrate by parts as much as we like).

Definition 5.1 (Compact support on (Omega)) *A function u has compact support on Ω if there exists $\epsilon > 0$ such that $u(x) = 0$ when $\min_{y\in\partial\Omega}|x-y| < \epsilon$.*

Definition 5.2 ((C^infty_0(Omega))) *We denote by $C_0^\infty(\Omega)$ the subset of $C^\infty(\Omega)$ corresponding to functions that have compact support on Ω.*

Next we will define a space containing the generalised derivative.

Definition 5.3 ((L^1_{loc})) *For triangles $K \subset \operatorname{int}(\Omega)$, we define*

$$\|u\|_{L^1(K)} = \int_K |u|\,dx,$$

and

$$L^1_K = \left\{u : \|u\|_{L^1(K)} < \infty\right\}.$$

Then

$$L^1_{loc} = \left\{f : f \in L^1(K) \quad \forall K \subset \operatorname{int}(\Omega)\right\}.$$

Finally we are in a position to introduce the generalisation of the derivative itself.

Definition 5.4 (Weak derivative) *The weak derivative $D^\alpha_w f \in L^1_{loc}(\Omega)$ of a function $f \in L^1_{loc}(\Omega)$ is defined by*

$$\int_\Omega \phi D^\alpha_w f \, dx = (-1)^{|\alpha|} \int_\Omega D^\alpha \phi f \, dx, \quad \forall \phi \in C^\infty_0(\Omega).$$

Not that we do not see any boundary terms since ϕ vanishes at the boundary along with all derivatives.

Now we check that the derivative agrees with our finite element derivative definition.

Lemma 5.5 *Let V be a C^0 finite element space. Then, for $u \in V$, the finite element derivative of u is equal to the weak derivative of u.*

Proof 5.6 *Taking any $\phi \in C^\infty_0(\Omega)$, we have*

$$\begin{aligned}\int_\Omega \phi \frac{\partial}{\partial x_i}|_{FE} u \, dx = \sum_K \int_K \phi \frac{\partial u}{\partial x_i} \, dx, \\ = \sum_K \left(-\int_K \frac{\partial \phi}{\partial x_i} u \, dx + \int_{\partial K} \phi n_i u \, dS \right), \\ = -\sum_K \int_K \frac{\partial \phi}{\partial x_i} u \, dx = -\int_\Omega \frac{\partial \phi}{\partial x_i} u \, dx,\end{aligned}$$

as required.

Exercise 5.7 *Let V be a C^1 finite element space. For $u \in V$, show that the finite second derivatives of u is equal to the weak second derivative of u.*

Exercise 5.8 *Let V be a discontinuous finite element space. For $u \in V$, show that the weak derivative does not coincide with the finite element derivative in general (find a counter-example).*

Lemma 5.9 *For $u \in C^{|\alpha|}(\Omega)$, the usual "strong" derivative D^α of u is equal to the weak derivative D^α_w of u.*

Exercise 5.10 *Prove this lemma.*

Due to these equivalences, we do not need to distinguish between strong, weak and finite element first derivatives for C^0 finite element functions. All derivatives are assumed to be weak from now on.

5.2 Sobolev spaces

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We are now in a position to define a space that contains all C^0 finite element spaces. This means that we can consider the limit of finite element approximations as $h \to 0$.

Definition 5.11 (The Sobolev space (H^1)) *$H^1(\Omega)$ is the function space defined by*

$$H^1(\Omega) = \left\{ u \in L^1_{loc} : \|u\|_{H^1(\Omega)} < \infty \right\}.$$

Going further, the Sobolev space H^k is the space of all functions with finite H^k norm.

Definition 5.12 (The Sobolev space (H^k)) *$H^k(\Omega)$ is the function space defined by*

$$H^k(\Omega) = \left\{ u \in L^1_{loc} : \|u\|_{H^k(\Omega)} < \infty \right\}$$

Since $\|u\|_{H^k(\Omega)} \leq \|u\|_{H^l(\Omega)}$ for $k < l$, we have $H^l \subset H^k$ for $k < l$.

If we are to consider limits of finite element functions in these Sobolev spaces, then it is important that they are closed, i.e. limits remain in the spaces.

Lemma 5.13 ((H^k) spaces are Hilbert spaces) *The space $H^k(\Omega)$ is closed.*

Let $\{u_i\}$ be a Cauchy sequence in H^k. Then $\{D^\alpha u_i\}$ is a Cauchy sequence in $L^2(\Omega)$ (which is closed), so $\exists v^\alpha \in L^2(\Omega)$ such that $D^\alpha u_i \to v^\alpha$ for $|\alpha| \leq k$. If $w_j \to w$ in $L^2(\Omega)$, then for $\phi \in C_0^\infty(\Omega)$,

$$\int_\Omega (w_j - w)\phi \, dx \leq \|w_j - w\|_{L^2(\Omega)} \|\phi\|_{L^\infty} \to 0.$$

We use this equation to get

$$\begin{aligned} \int_\Omega v^\alpha \phi \, dx &= \lim_{i\to\infty} \int_\Omega \phi D^\alpha u_i \, dx, \\ &= \lim_{i\to\infty} (-1)^{|\alpha|} \int_\Omega u_i D^\alpha \phi \, dx, \\ &= (-1)^{|\alpha|} \int_\Omega v D^\alpha \phi \, dx, \end{aligned}$$

i.e. v^α is the weak derivative of u as required.

We quote the following much deeper results without proof.

Theorem 5.14 ((H=W)) *Let Ω be any open set. Then $H^k(\Omega) \cap C^\infty(\Omega)$ is dense in $H^k(\Omega)$.*

The interpretation is that for any function $u \in H^k(\Omega)$, we can find a sequence of C^∞ functions u_i converging to u. This is very useful as we can compute many things using C^∞ functions and take the limit.

Theorem 5.15 (Sobolev's inequality) *Let Ω be an n-dimensional domain with Lipschitz boundary, let k be an integer with $k > n/2$. Then there exists a constant C such that*

$$\|u\|_{L^\infty(\Omega)} = \operatorname*{ess\,sup}_{x\in\Omega} |u(x)| \leq C\|u\|_{H^k(\Omega)}.$$

Further, there is a C^0 continuous function in the $L^\infty(\Omega)$ equivalence class of u.

Previously we saw this result for continuous functions. Here it is presented for H^k functions, with an extra statement about the existence of a C^0 function in the equivalence class. The interpretation is that if $u \in H^k$ then there is a continuous function u_0 such that the set of points where $u \neq u_0$ has zero area/volume.

Corollary 5.16 (Sobolev's inequality for derivatives) *Let Ω be a n-dimensional domain with Lipschitz boundary, let k be an integer with $k - m > n/2$. Then there exists a constant C such that*

$$\|u\|_{W_\infty^m(\Omega)} := \sum_{|\alpha|\leq m} \|D^\alpha u\|_{L^\infty(\Omega)} \leq C\|u\|_{H^k(\Omega)}.$$

Further, there is a C^m continuous function in the $L^\infty(\Omega)$ equivalence class of u.

Proof 5.17 *Just apply Sobolev's inequality to the m derivatives of u.*

5.3 Variational formulations of PDEs

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

We can now consider linear variational problems defined on H^k spaces, by taking a bilinear form $b(u, v)$ and linear form $F(v)$, seeking $u \in H^k$ (for chosen H^k) such that

$$b(u, v) = F(v), \quad \forall v \in H^k.$$

Since H^k is a Hilbert space, the Lax-Milgram theorem can be used to analyse, the existence of a unique solution to an H^k linear variational problem.

For example, the Helmholtz problem solvability is immediate.

Theorem 5.18 (Well-posedness for (modified) Helmholtz)) *The Helmholtz variational problem on H^1 satisfies the conditions of the Lax-Milgram theorem.*

Proof 5.19 *The proof for C^0 finite element spaces extends immediately to H^1.*

Next, we develop the relationship between solutions of the Helmholtz variational problem and the strong-form Helmholtz equation,

$$u - \nabla^2 u = f, \quad \frac{\partial u}{\partial n} = 0, \text{ on } \partial\Omega.$$

The basic idea is to check that when you take a solution of the Helmholtz variational problem and integrate by parts (provided that this makes sense) then you reveal that the solution solves the strong form equation. Functions in H^k make boundary values hard to interpret since they are not guaranteed to have defined values on the boundary. We make the following definition.

Definition 5.20 (Trace of (H^1) functions) *Let $u \in H^1(\Omega)$ and choose $u_i \in C^\infty(\Omega)$ such that $u_i \to u$. We define the trace $u|_{\partial\Omega}$ on $\partial\Omega$ as the limit of the restriction of u_i to $\partial\Omega$. This definition is unique from the uniqueness of limits.*

We can extend our trace inequality for finite element functions directly to H^1 functions.

Lemma 5.21 (Trace theorem for (H^1) functions) *Let $u \in H^1(\Omega)$ for a polygonal domain Ω. Then the trace $u|_{\partial\Omega}$ satisfies*

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

The interpretation of this result is that if $u \in H^1(\Omega)$ then $u|_{\partial\Omega} \in L^2(\partial\Omega)$.

Proof 5.22 *Adapt the proof for C^0 finite element functions, choosing $u \in C^\infty(\Omega)$, and pass to the limit in $H^1(\Omega)$.*

This tells us when the integration by parts formula makes sense.

Lemma 5.23 *Let $u \in H^2(\Omega)$, $v \in H^1(\Omega)$. Then*

$$\int_\Omega (-\nabla^2 u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, dS.$$

Proof 5.24 *First note that $u \in H^2(\Omega) \implies \nabla u \in (H^1(\Omega))^d$. Then*

Then, take $v_i \in C^\infty(\Omega)$ and $u_i \in C^\infty(\Omega)$ converging to v and u, respectively, and $v_i \nabla u_i \in C^\infty(\Omega)$ converges to $v\nabla u$. These satisfy the equation; we obtain the result by passing to the limit.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Now we have everything we need to show that solutions of the strong form equation also solve the variational problem. It is just a matter of substituting into the formula and applying integration by parts.

Lemma 5.25 *For $f \in L^2$, let $u \in H^2(\Omega)$ solve*

$$u - \nabla^2 u = f, \quad \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega,$$

in the L^2 sense, i.e. $\|u - \nabla^2 u - f\|_{L^2} = 0$. Then u solves the variational form of the Helmholtz equation.

Proof 5.26 $u \in H^2 \implies \|u\|_{H^2} < \infty \implies \|u\|_{H^1} < \infty \implies u \in H^1$. *Multiplying by test function $v \in H^1$, and using the previous proposition gives*

$$\int_\Omega uv + \nabla u \cdot \nabla v \, dx = \int_\Omega fv \, dx, \quad \forall v \in H^1(\Omega),$$

as required.

Now we go the other way, showing that solutions of the variational problem also solve the strong form equation. To do this, we need to assume a bit more smoothness of the solution, that it is in H^2 instead of just H^1.

Theorem 5.27 *Let $f \in L^2(\Omega)$ and suppose that $u \in H^2(\Omega)$ solves the variational Helmholtz equation on a polygonal domain Ω. Then u solves the strong form Helmholtz equation with zero Neumann boundary conditions.*

Proof 5.28 *Using integration by parts for $u \in H^2$, $v \in C_0^\infty(\Omega) \in H^1$, we have*

$$\int_\Omega (u - \nabla^2 u - f) v \, dx = \int_\Omega uv + \nabla u \cdot \nabla v - vf \, dx = 0.$$

It is a standard result that $C_0^\infty(\Omega)$ is dense in $L^2(\Omega)$ (i.e., every L^2 function can be approximated arbitrarily closely by a C_0^∞ function), and therefore we can choose a sequence of v converging to $u - \nabla^2 u - f$ and we obtain $\|u - \nabla^2 u - f\|_{L^2(\Omega)} = 0$.

Now we focus on showing the boundary condition is satisfied. We have

$$\begin{aligned} 0 = \int_\Omega uv + \nabla u \cdot \nabla v - fv \, dx \\ &= \int_\Omega uv + \nabla u \cdot \nabla v - (u - \nabla^2 u) v \, dx \\ &= \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, dS. \end{aligned}$$

We can find arbitrary $v \in L_2(\partial\Omega)$, hence $\|\frac{\partial u}{\partial n}\|_{L^2(\partial\Omega)} = 0$.

5.4 Galerkin approximations of linear variational problems

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Going a bit more general again, assume that we have a well-posed linear variational problem on H^k, connected to a strong form PDE. Now we would like to approximate it. This is done in general using the Galerkin approximation.

Definition 5.29 (Galerkin approximation) *Consider a linear variational problem of the form:*

find $u \in H^k$ such that

$$b(u, v) = F(v), \quad \forall v \in H^k.$$

For a finite element space $V_h \subset V = H^k(\Omega)$, the Galerkin approximation of this H^k variational problem seeks to find $u_h \in V_h$ such that

$$b(u_h, v) = F(v), \quad \forall v \in V_h.$$

We just restrict the trial function u and the test function v to the finite element space. C^0 finite element spaces are subspaces of H^1, C^1 finite element spaces are subspaces of H^2 and so on.

If $b(u, v)$ is continuous and coercive on H^k, then it is also continuous and coercive on V_h by the subspace property. Hence, we know that the Galerkin approximation exists, is unique and is stable. This means that it will be possible to solve the matrix-vector equation.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Moving on, if we can solve the equation, we would like to know if it is useful. What is the size of the error $u - u_h$? For Galerkin approximations this question is addressed by Céa's lemma.

Theorem 5.30 (Céa's lemma.) *Let $V_h \subset V$, and let u solve a linear variational problem on V, whilst u_h solves the equivalent Galerkin approximation on V_h. Then*

$$\|u - u_h\|_V \leq \frac{M}{\gamma} \min_{v \in V_h} \|u - v\|_V,$$

where M and γ are the continuity and coercivity constants of $b(u, v)$, respectively.

Proof 5.31 *We have*

$$b(u, v) = F(v) \quad \forall v \in V, b(u_h, v) = F(v) \quad \forall v \in V_h.$$

Choosing $v \in V_h \subset V$ means we can use it in both equations, and subtraction and linearity lead to the "Galerkin orthogonality" condition

$$b(u - u_h, v) = 0, \quad \forall v \in V_h.$$

Then, for all $v \in V_h$,

$$\begin{aligned} \gamma\|u - u_h\|_V^2 &\leq b(u - u_h, u - u_h), \\ &= b(u - u_h, u - v) + \underbrace{b(u - u_h, v - u_h)}_{=0}, \\ &\leq M\|u - u_h\|_V \|u - v\|_V. \end{aligned}$$

So,

$$\gamma\|u - u_h\|_V \leq M\|u - v\|_V.$$

Minimising over all v completes the proof.

5.5 Interpolation error in H^k spaces

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

The interpretation of Céa's lemma is that the error is proportional to the minimal error in approximating u in V_h. To do this, we can simply choose $v = \mathcal{I}_h u$ in Céa's lemma, to get

$$\|u - u_h\|_V \leq \frac{M}{\gamma} \min_{v \in V_h} \|u - v\|_V \leq \frac{M}{\gamma}\|u - \mathcal{I}_h u\|_V.$$

Hence, Céa's lemma reduces the problem of estimating the error in the numerical solution to estimating error in the interpolation of the exact solution. We have already examined this in the section on interpolation operators, but in the context of continuous functions. The problem is that we do not know that the solution u is continuous, only that it is in H^k for some k.

We now quickly revisit the results of the interpolation section to extend them to H^k spaces. The proofs are mostly identical, so we just give the updated result statements and state how to modify the proofs.

Firstly we recall the averaged Taylor polynomial. Since it involves only integrals of the derivatives, we can immediately use weak derivatives here.

Definition 5.32 (Averaged Taylor polynomial with weak derivatives) *Let $\Omega \subset \mathbb{R}^n$ be a domain with diameter d, that is star-shaped with respect to a ball B with radius ϵ, contained within Ω. For $f \in H^{k+1}(\Omega)$ the averaged Taylor polynomial $Q_{k,B} f \in \mathcal{P}_k$ is defined as*

$$Q_{k,B} f(x) = \frac{1}{|B|} \int_B T^k f(y, x) \, dy,$$

where $T^k f$ is the Taylor polynomial of degree k of f,

$$T^k f(y, x) = \sum_{|\alpha| \leq k} D^\alpha f(y) \frac{(x - y)^\alpha}{\alpha!},$$

evaluated using weak derivatives.

This definition makes sense since the Taylor polynomial coefficients are in $L^1_{loc}(\Omega)$ and thus their integrals over B are defined.

The next step was to examine the error in the Taylor polynomial.

Theorem 5.33 *Let $\Omega \subset \mathbb{R}^n$ be a domain with diameter d, that is star-shaped with respect to a ball B with radius ϵ, contained within Ω. There exists a constant $C(k,n)$ such that for $0 \leq |\beta| \leq k+1$ and all $f \in H^{k+1}(\Omega)$,*

$$\|D^\beta(f - Q_{k,B}f)\|_{L^2} \leq C\frac{|\Omega|^{1/2}}{|B|^{1/2}} d^{k+1-|\beta|}\|\nabla^{k+1}f\|_{L^2(\Omega)}.$$

Proof 5.34 *To show this, we assume that $f \in C^\infty(\Omega)$, in which case the result of Theorem 3.17 applies. Then we obtain the present result by approximating f by a sequence of $C^\infty(\Omega)$ functions and passing to the limit.*

We then repeat the following corollary.

Corollary 5.35 *Let K_1 be a triangle with diameter 1. There exists a constant $C(k,n)$ such that*

$$\|f - Q_{k,B}f\|_{H^k(K_1)} \leq C|\nabla^{k+1}f|_{H^{k+1}(K_1)}.$$

Proof 5.36 *Same as Lemma 3.19.*

The next step was the bound on the interpolation operator. Now we just have to replace $C^{l,\infty}$ with W^l_∞ as derivatives may not exist at every point.

Lemma 5.37 *Let $(K_1, \mathcal{P}, \mathcal{N})$ be a finite element such that K_1 is a triangle with diameter 1, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\|N_i\|_{W^l_\infty(K_1)'} < \infty$,*

$$\|N_i\|_{W^l_\infty(K_1)'} = \sup_{\|u\|_{W^l_\infty(K_1)}>0} \frac{|N_i(u)|}{\|u\|_{W^l_\infty(K_1)}}.$$

Let $u \in H^k(K_1)$ with $k > l + n/2$. Then

$$\|\mathcal{I}_{K_1}u\|_{H^k(K_1)} \leq C\|u\|_{H^k(K_1)}.$$

Proof 5.38 *Same as Lemma 3.26. replacing $C^{l,\infty}$ with W^l_∞, and using the full version of the Sobolev inequality in Lemma 5.15.*

The next steps then just follow through.

Lemma 5.39 *Let $(K_1, \mathcal{P}, \mathcal{N})$ be a finite element such that K_1 has diameter 1, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\mathcal{P}$ contain all polynomials of degree k and below, with $k > l + n/2$. Let $u \in H^{k+1}(K_1)$. Then for $i \leq k$, the local interpolation operator satisfies*

$$|\mathcal{I}_{K_1}u - u|_{H^i(K_1)} \leq C_1|u|_{H^{k+1}(K_1)}.$$

Proof 5.40 *Same as Lemma 3.28.*

Lemma 5.41 *Let $(K, \mathcal{P}, \mathcal{N})$ be a finite element such that K has diameter d, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\mathcal{P}$ contains all polynomials of degree k and below, with $k > l + n/2$. Let $u \in H^{k+1}(K)$. Then for $i \leq k$, the local interpolation operator satisfies*

$$|\mathcal{I}_K u - u|_{H^i(K)} \leq C_K d^{k+1-i}|u|_{H^{k+1}(K)}.$$

where C_K is a constant that depends on the shape of K but not the diameter.

Proof 5.42 *Repeat the scaling argument of Lemma 3.30.*

Theorem 5.43 *Let $\mathcal{T}$ be a triangulation with finite elements $(K_i, \mathcal{P}_i, \mathcal{N}_i)$, such that the minimum aspect ratio r of the triangles K_i satisfies $r > 0$, and such that the nodal variables in $\mathcal{N}$ involve only evaluations of functions or evaluations of derivatives of degree $\leq l$, and $\mathcal{P}$ contains all polynomials of degree k and below, with $k > l + n/2$. Let $u \in H^{k+1}(\Omega)$. Let h be the maximum over all of the triangle diameters, with $0 \leq h < 1$. Let V be the corresponding C^r finite element space. Then for $i \leq k$ and $i \leq r+1$, the global interpolation operator satisfies*

$$\|\mathcal{I}_h u - u\|_{H^i(\Omega)} \leq Ch^{k+1-i}|u|_{H^{k+1}(\Omega)}.$$

Proof 5.44 *Identical to Theorem 3.32.*

5.6 Convergence of the finite element approximation to the Helmholtz problem

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

Now that we have the required interpolation operator results, we can return to applying Céa's lemma to the convergence of the finite element approximation to the Helmholtz problem.

Corollary 5.45 *The degree k Lagrange finite element approximation u_h to the solution u of the variational Helmholtz problem satisfies*

$$\|u_h - u\|_{H^1(\Omega)} \leq Ch^k \|u\|_{H^{k+1}(\Omega)}.$$

Proof 5.46 *We combine Céa's lemma with the previous estimate, since*

$$\min_{v \in V_h} \|u - v\|_{H^1(\Omega)} \leq \|u - \mathcal{I}_h u\|_{H^1(\Omega)} \leq Ch^k \|u\|_{H^{k+1}}(\Omega),$$

having chosen $i = 1$.

Exercise 5.47 *Consider the variational problem of finding $u \in H^1([0,1])$ such that*

$$\int_0^1 vu + v'u' \, dx = \int_0^1 vx \, dx + v(1) - v(0), \quad \forall v \in H^1([0,1]).$$

After dividing the interval $[0,1]$ into N equispaced cells and forming a $P1$ C^0 finite element space V_N, the error $\|u - u_h\|_{H^1} = 0$ for any $N > 0$.

Explain why this is expected.

Exercise 5.48 *Let $\mathring{H}^1([0,1])$ be the subspace of $H^1([0,1])$ such that $u(0) = 0$. Consider the variational problem of finding $u \in \mathring{H}^1([0,1])$ with*

$$\int_0^1 v'u' \, dx = \int_0^{1/2} v \, dx, \quad \forall v \in \mathring{H}([0,1]).$$

The interval $[0,1]$ is divided into $2N+1$ equispaced cells (where N is a positive integer). After forming a $P2$ C^0 finite element space V_N, the error $\|u - u_h\|_{H^1}$ only converges at a linear rate. Explain why this is expected.

Exercise 5.49 Let Ω be a convex polygonal 2D domain. Consider the *following two problems.*

1. *Find $u \in H^2$ such that*

$$\|\nabla^2 u + f\|_{L^2(\Omega)} = 0, \quad \|u\|_{L^2(\partial\Omega)} = 0,$$

which we write in a shorthand as

$$-\nabla^2 u = f, \quad u|_{\partial\Omega} = 0.$$

2. *Find $u \in \mathring{H}^1(\Omega)$ such that*

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega fv \, dx, \quad \forall v \in \mathring{H}^1(\Omega),$$

where $\mathring{H}^1(\Omega)$ is the subspace of $H^1(\Omega)$ consisting of functions whose trace vanishes on the boundary.

Under assumptions on u which you should state, show that a solution to problem (1.) is a solution to problem (2.).

Let h be the maximum triangle diameter of a triangulation T_h of Ω, with V_h the corresponding linear Lagrange finite element space. Construct a finite element approximation to Problem (2.) above. Briefly give the main arguments as to why the $H^1(\Omega)$ norm of the error converges to zero linearly in h as $h \to 0$, giving your assumptions.

Céa's lemma gives us error estimates in the norm of the space where the variational problem is defined, where the continuity and coercivity results hold. In the case of the Helmholtz problem, this is H^1. We would also like estimates of the error in the L^2 norm, and it will turn out that these will have a more rapid convergence rate as $h \to 0$.

A video recording of the following material is available here.

Imperial students can also watch this video on Panopto

To do this we quote the following without proof.

Theorem 5.50 (Elliptic regularity) *Let w solve the equation*

$$w - \nabla^2 w = f, \quad \frac{\partial w}{\partial n} = 0 \text{ on } \partial\Omega,$$

on a convex (results also hold for other types of "nice" domains) domain Ω, with $f \in L^2$. Then there exists constant $C > 0$ such that

$$|w|_{H^2(\Omega)} \leq C\|f\|_{L^2(\Omega)}.$$

Similar results hold for general elliptic operators, such as Poisson's equation with the types of boundary conditions discussed above. Elliptic regularity is great to have, because it says that the solution of the H^1 variational problem is actually in H^2, provided that $f \in L^2$.

We now use this to obtain the following result, using the Aubin-Nitsche trick.

Theorem 5.51 *The degree k Lagrange finite element approximation u_h to the solution u of the variational Helmholtz problem satisfies*

$$\|u_h - u\|_{L^2(\Omega)} \leq Ch^{k+1}\|u\|_{H^{k+1}(\Omega)}.$$

Proof 5.52 *We use the Aubin-Nitsche duality argument. Let w be the solution of*

$$w - \nabla^2 w = u - u_h,$$

with the same Neumann boundary conditions as for u.

Since $u - u_h \in H^1(\Omega) \subset L^2(\Omega)$, we have $w \in H^2(\Omega)$ by elliptic regularity.

Then we have (by multiplying by a test function an integrating by parts),

$$b(w, v) = (u - u_h, v)_{L^2(\Omega)}, \quad \forall v \in H^1(\Omega),$$

and so

$$\begin{aligned}\|u - u_h\|^2_{L^2(\Omega)} &= (u - u_h, u - u_h) = b(w, u - u_h), = b(w - \mathcal{I}_h w, u - u_h) \text{ (orthogonality)} ,\\ &\leq C\|u - u_h\|_{H^1(\Omega)}\|w - \mathcal{I}_h w\|_{H^1(\Omega)},\\ &\leq Ch\|u - u_h\|_{H^1(\Omega)}|w|_{H^2(\Omega)}\\ &\leq C_1 h^{k+1}|u|_{H^{k+1}(\Omega)\|u-u_h\|_{L^2(\Omega)}}\end{aligned}$$

and dividing both sides by $\|u - u_h\|_{L^2(\Omega)}$ gives the result.

Thus we gain one order of convergence rate with h by using the L^2 norm instead of the H^1 norm.

5.7 Epilogue

This completes our analysis of the convergence of the Galerkin finite element approximation to the Helmholtz problem. Similar approaches can be applied to analysis of other elliptic PDEs, using the following programme.

1. Find a variational formulation of the PDE with a bilinear form that is continuous and coercive (and hence well-posed by Lax-Milgram) on H^k for some k.
2. Find a finite element space $V_h \subset H^k$. For H^1, this requires a C^0 finite element space, and for H^2, a C^1 finite element space is required.

3. The Galerkin approximation to the variational formulation is obtained by restricting the solution and test functions to V_h.
4. Continuity and coercivity (and hence well-posedness) for the Galerkin approximation is assured since $V_h \subset H^k$. This means that the Galerkin approximation is solvable and stable.
5. The estimate of the error estimate in terms of h comes from Céa's lemma plus the error estimate for the nodal interpolation operator.

This course only describes the beginning of the subject of finite element methods, for which research continues to grow in both theory and application. There are many methods and approaches that go beyond the basic Galerkin approach described above. These include

- Discontinuous Galerkin methods, which use discontinuous finite element spaces with jump conditions between cells to compensate for not having the required continuity. These problems do not fit into the standard Galerkin framework and new techniques have been developed to derive and analyse them.
- Mixed finite element methods, which consider systems of partial differential equations such as the Poisson equation in first-order form,

$$u - \nabla p = 0, \quad \nabla \cdot u = f.$$

  The variational forms corresponding to these systems are not coercive, but they are well-posed anyway, and additional techniques have been developed.
- Non-conforming methods, which work even though $V_h \not\subset H^k$. For example, the Crouzeix-Raviart element uses linear functions that are only continuous at edge centres, so the functions are not in C^0 and the functions do not have a weak derivative. However, using the finite element derivative in the weak form for H^1 elliptic problems still gives a solvable system that converges at the optimal rate. Additional techniques have been developed to analyse this.
- Interior penalty methods, which work even though $V_h \not\subset H^k$. These methods are used to solve H^k elliptic problems using H^l finite element spaces with $l < k$, using jump conditions to obtain a stable discretisation. Additional techniques have been developed to analyse this.
- Stabilised and multiscale methods for finite element approximation of PDEs whose solutions have a wide range of scales, for example they might have boundary layers, turbulent structures or other phenomena. Resolving this features is often too expensive, so the goal is to find robust methods that behave well when the solution is not well resolved. Additional techniques have been developed to analyse this.
- Hybridisable methods that involve flux functions that are supported only on cell facets.
- Currently there is a lot of activity around discontinuous Petrov-Galerkin methods, which select optimal test functions to maximise the stability of the discrete operator. This means that they can be applied to problems such as wave propagation which are otherwise very challenging to find stable methods for. Also, these methods come with a bespoke error estimator that can allow for adaptive meshing starting from very coarse meshes. Another new and active area is virtual element methods, where the basis functions are not explicitly defined everywhere (perhaps just on the boundary of cells). This facilitates the use of arbitrary polyhedra as cells, leading to very flexible mesh choices.

All of these methods are driven by the requirements of different physical applications.

Other rich areas of finite element research include

- the development of bespoke, efficient iterative solver algorithms on parallel computers for finite element discretisations of PDEs. Here, knowledge of the analysis of the discretisation can lead to solvers that converge in a number of iterations that is independent of the mesh parameter h.
- adaptive mesh algorithms that use analytical techniques to estimate or bound the numerical error after the numerical solution has been computed, in order to guide iterative mesh refinement in particular areas of the domain.

CHAPTER SIX

STOKES EQUATION

Note

This section is the mastery topic for this module. It consists of some extra material that is not covered in lectures which will be covered in the mastery question on the exam.

This section is not a part of the third year version of this module.

6.1 Strong form of the equations

In this section we consider finite element discretisations of the Stokes equation of a viscous fluid, given by

$$-2\mu\nabla\cdot\epsilon(u) + \nabla p = f, \quad \nabla\cdot u = 0, \quad \epsilon(u) = \frac{1}{2}\left(\nabla u + \nabla u^T\right), \tag{6.1}$$

where u is the (vector-valued) fluid velocity, p is the pressure, μ is the viscosity and f is a (vector-valued) external force applied to the fluid. This model gives the motion of a fluid in the high viscosity limit and has applications in industrial, geological and biological flows. For less viscous fluids we use the Navier-Stokes equation which consists of the Stokes equations plus additional nonlinear terms. To understand discretisations of the Navier-Stokes equations it is necessary to first understand discretisations of the Stokes equation. There are several relevant boundary conditions for the Stokes equation, but for now we shall consider the "no slip" boundary condition $u = 0$ on the entire boundary $\partial\Omega$. Note that ∇u is a 2-tensor (i.e. a matrix-valued function), with

$$(\nabla u)_{ij} = \frac{\partial u_i}{\partial x_j}, (\nabla u^T)_{ij} = (\nabla u)_{ji}. \tag{6.2}$$

Note that under the incompressibility constraint $\nabla\cdot u = 0$, we can write $\nabla\cdot\epsilon(u) = \nabla^2 u$. However, this leads to various issues in the finite element discretisation, and makes it harder to apply stress-free boundary conditions.

Under no slip boundary conditions, the pressure p only appears in Stokes equation inside a gradient, hence we can only expect to solve these equations for p up to an additive constant. To fix that constant, with no slip boundary conditions we additionally require

$$\int_\Omega p \, d\, x = 0. \tag{6.3}$$

6.2 Variational form of the equations

To proceed to the finite element discretisation, we need to find an appropriate variational formulation of the Stokes equations. Defining $V = (\mathring{H}^1(\Omega))^n$ (i.e. vector valued functions in physical dimension n with each Cartesian component in $\mathring{H}^1(\Omega)$, which is the subspace of $H^1(\Omega)$ consisting of functions that vanish on the boundary, and $Q = \mathring{L}^2(\Omega)$, with

$$\mathring{L}^2(\Omega) = \left\{p \in L^2(\Omega) : \int_\Omega p = 0\right\}. \tag{6.4}$$

Definition 6.1 *The variational formulation of the Stokes equation seeks* $(u, p) \in V \times Q$ *such that*

$$\begin{aligned} a(u, v) + b(v, p) &= \int_\Omega f \cdot v \, dx, \\ b(u, q) &= 0, \quad \forall (v, q) \in V \times Q, \end{aligned} \tag{6.5}$$

where

$$\begin{aligned} a(u, v) &= 2\mu \int_\Omega \epsilon(u) : \epsilon(v) dx, \\ b(v, q) &= -\int_\Omega q \nabla \cdot v dx. \end{aligned} \tag{6.6}$$

Exercise 6.2 *Show that if* (u, p) *solve the variational formulation of the Stokes equations, and further that* $u \in H^2(\Omega)$*,* $p \in H^1(\Omega)$*, then* (u, p) *solves the strong form of the Stokes equations.*

We call this type of problem a "mixed problem" defined on a "mixed function space" $V \times Q$, since we solve simultaneously for $u \in V$ and $p \in Q$. If we define $X = V \times Q$, and define $U = (u, p) \in X$ (as well as $W = (v, q) \in X$, then we can more abstractly write the problem as finding $U \in X$ such that

$$c(U, W) = F(W), \tag{6.7}$$

where for the case of Stokes equation,

$$c(U, W) = a(u, v) + b(v, p) + b(u, q), \quad F(W) = \int_\Omega f \cdot v \, dx. \tag{6.8}$$

There is a challenge with Stokes equation which is that it is not coercive, i.e. there does not exist a constant $C > 0$ such that

$$\|U\|_X^2 \leq Cc(U, U), \quad \forall U \in X, \tag{6.9}$$

where here we use the product norm

$$\|U\|_X^2 = \|u\|_{H^1(\Omega)}^2 + \|p\|_{L^2(\Omega)}^2. \tag{6.10}$$

This means that we can't use the Lax Milgram Theorem to show existence and uniqueness of solutions for the variational formulation or any finite element discretisations of it, and we can't use Céa's Lemma to estimate numerical errors in the finite element discretisation. Instead we have to use a more general tool, the inf-sup theorem.

Exercise 6.3 *Show that the form* $c(\cdot, \cdot)$ *is not coercive by considering the case* $v = 0$.

6.3 The inf-sup condition

Tip

The key to understanding this section and the following one is to have a good recollection of the definition of dual spaces and dual space norms given in the earlier section on *Linear forms on Hilbert spaces*. It is a good idea to go back and review that section before you carry on.

The critical tool in mixed problems is the inf-sup condition for a bilinear form on $V \times Q$, which says that there exists $\beta > 0$ such that

$$\inf_{0\neq q\in Q}\sup_{0\neq v\in V}\frac{b(v,q)}{\|v\|_V\|q\|_Q}\geq\beta. \tag{6.11}$$

For brevity, we will drop the $\neq 0$ condition in subsequent formulae. To understand this condition, we consider the map $B:V\to Q'$ given by

$$Bv[p]=b(v,p),\,\forall p\in Q, \tag{6.12}$$

and the transpose operator $B^*:Q\to V'$, by

$$B^*p[v]=b(v,p),\quad\forall v\in V. \tag{6.13}$$

Here, Bv is the map B applied to v: Bv is an element of the dual space Q' which itself maps elements of Q to $\mathbb{R}$. B^*p is the image of the map B^* applied to p: B^*p is an element of the dual space V' which itself maps elements of V to $\mathbb{R}$.

The norm of B^*q is

$$\|B^*q\|_{V'}=\sup_{v\in V}\frac{b(v,q)}{\|v\|_V}. \tag{6.14}$$

This allows us to rewrite the inf-sup condition as

$$\inf_{q\in Q}\frac{\|B^*q\|_{V'}}{\|q\|_Q}\geq\beta, \tag{6.15}$$

which is also equivalent to

$$\|B^*q\|_{V'}\geq\beta\|q\|_Q,\,\forall q\in Q. \tag{6.16}$$

This tells us that the map B^* is injective, since if there exist q_1,q_2 such that $B^*q_1=B^*q_2$, then $B^*(q_1-q_2)=0\implies 0=\|B^*(q_1-q_2)\|_V\geq\beta\|q_1-q_2\|_Q$, i.e. $q_1=q_2$.

In finite dimensions (such as for our finite element spaces), injective B^* is equivalent to surjective B (via the rank-nullity theorem). In infinite dimensions, such as the case $\mathring{H}^1\times\mathring{L}^2$ that we are considering for Stokes equation, the situation is more complicated and is governed by the Closed Range Theorem (which we allude to here but do not state or prove), which tells us that for Hilbert spaces and continuous bilinear forms $b(v,q)$, injective B^* is indeed equivalent to surjective B.

The Closed Range Theorem (and the rank-nullity theorem, its finite dimensional version) further characterises these maps using perpendicular spaces.

Definition 6.4 (Perpendicular space) *For a subspace $Z\subset Q$ of a Hilbert space Q, the perpendicular space $Z^\perp$ of Z in Q is*

$$Z^\perp=\left\{q\in Q:\langle q,p\rangle_Q=0,\,\forall p\in Z\right\}. \tag{6.17}$$

In finite dimensions, we have that B^* defines a one-to-one mapping from $(\mathrm{Ker}B^*)^\perp \subset Q$ (the perpendicular space to the kernel $\mathrm{Ker}B^*$ of B^*) to $\mathrm{Im}(B^*)$ (the image space of B^*). This is also true in infinite dimensions under the conditions of the Closed Range Theorem.

This means that for any $F \in \mathrm{Im}(B^*)$, we can find $q \in (\mathrm{Ker}B^*)^\perp$ such that $B^*q = F$. Further, we have

$$\|F\|_{V'} \geq \beta \|q\|_Q, \tag{6.18}$$

via the inf-sup condition.

Finally, it is useful to characterise $\mathrm{Im}(B^*)$. In $\mathbb{R}^n$, we are used to the rank-nullity theorem telling us that $\mathrm{Im}(B^*) = (\mathrm{Ker}B^*)^\perp$. However, here B^* maps to V', not V, so this does not make sense. When considering maps between dual spaces, we have to generalise this idea to polar spaces.

Definition 6.5 (Polar space) *For Z a subspace of a Hilbert space Q, the polar space Z^0 is the subspace of Q' of continuous linear functionals that vanish on Z i.e.*

$$Z^0 = \{F \in Q' : F[q] = 0 \, \forall q \in Z\}. \tag{6.19}$$

Then the dual space version of the rank-nullity theorem (and the Closed Range Theorem for infinity dimensional Hilbert spaces) tells us that

$$\mathrm{Im}(B^*) = (\mathrm{Ker}B)^0. \tag{6.20}$$

Equipped with this tool, we can look at solveability of mixed problems.

6.4 Solveability of mixed problems

For symmetric, mixed problems in two variables, sufficient conditions for existence are given by the following result of Franco Brezzi.

Theorem 6.6 (Brezzi's conditions) *Let $a(u, v)$ be a continuous bilinear form defined on $V \times V$, and $b(v, q)$ be a continuous bilinear form defined on $V \times Q$. Consider the variational problem for $(u, p) \in V \times Q$,*

$$\begin{aligned} a(u, v) + b(v, p) &= F[v], \ \forall v \in V, \\ b(u, q) &= G[q], \ \forall q \in Q, \end{aligned} \tag{6.21}$$

for F and G continuous linear forms on V and Q respectively.

Define the kernel Z by

$$Z = \{u \in V : b(u, q) = 0 \forall q \in Q\}. \tag{6.22}$$

Assume the following conditions:

1. *$a(u, v)$ is coercive on the kernel Z with coercivity constant α.*
2. *There exists $\beta > 0$ such that the inf-sup condition for $b(v, q)$ holds.*

Then there exists a unique solution (u, p) to the variational problem and we have the stability bound

$$\begin{aligned} \|u\|_V &\leq \frac{1}{\alpha}\|F\|_{V'} + \frac{2M}{\alpha\beta}\|G\|_{Q'}, \\ \|p\|_Q &\leq \frac{2M}{\alpha\beta}\|F\|_{V'} + \frac{2M^2}{\alpha\beta^2}\|G\|_{Q'}, \end{aligned} \tag{6.23}$$

where M is the continuity constant of a.

Proof 6.7 *To show existence, we first note that the inf-sup condition implies that B is surjective, so we can always find $u_g \in V$ such that $Bu_g = g$. Now we write $u = u_g + u_Z$, and we have the following mixed problem,*

$$\begin{aligned} a(u_Z, v) + b(v, p) = F[v] - a(u_g, v),\ \forall v \in V, \\ b(u_Z, q) = 0. \end{aligned} \tag{6.24}$$

Thus, $Bu_Z = 0$, i.e. $u_Z \in Z$. Choosing $v \in Z \subset V$, we get

$$a(u_Z, v) = F'[v] = F[v] - a(u_g, v),\ \forall v \in Z, \tag{6.25}$$

for $u_Z \in Z$. Since $a(u,v)$ is coercive on Z, and F' is continuous (from continuity of F and $a(u,v)$), Lax Milgram tells us that $u_Z \in Z$ exists and is unique. We now notice that

$$L[v] = F[v] - a(u_g + u_Z, v) = 0 \forall v \in Z, \tag{6.26}$$

so $L[v] \in Z^0 = (\mathrm{Ker} B)^0 = \mathrm{Im} B^*$. This means that there exists $p \in Q$ such that $B^* p = L$. Hence, we have found (u,p) that solve our mixed variational problem.

To show uniqueness, we need to show that if there exists (u_1, p_1) and (u_2, p_2) that both solve our mixed variational problem, then $(u,p) = (u_1 - u_2, p_1 - p_2) = 0$. To that end, we take the difference of the equations for the two solutions, and get

$$\begin{aligned} a(u, v) + b(v, p) = 0,\ \forall v \in V, \\ b(u, q) = 0, \forall q \in Q. \end{aligned} \tag{6.27}$$

It is our goal to show that $(u,p) = 0$. We have again that $u \in Z$, and taking $v = u$ gives

$$0 = a(u,u) \geq \alpha \|u\|_V^2 \implies u = 0. \tag{6.28}$$

Substituting this into the problem for (u,p) gives

$$b(v, p) = 0,\ \forall v \in V. \tag{6.29}$$

Since b is injective, this means that $p = 0$ as required.

Having shown existence and uniqueness of (u,p), we want to develop the stability bounds. We now assume that (u,p) solves the variational problem. We first use the surjectivity of B to find u_g such that $Bu_g = G$. This means that

$$b(q, u_g) = G[q], \forall q \in Q, \tag{6.30}$$

Then, for all $q \in Q$,

$$\begin{aligned} \|G\|_{Q'} &= \sup_{q \in Q} \frac{b(q, u_g)}{\|q\|_Q} \\ &= \sup_{q \in Q} \frac{b(q, u_g)}{\|q\|_Q \|u_g\|_V} \|u_g\|_V \\ &\geq \beta \|u_g\|, \end{aligned} \tag{6.31}$$

by the inf-sup condition.

From the Lax Milgram theorem applied to (6.25), we get

$$\begin{aligned}\|u_Z\|_V &\leq \frac{1}{\alpha}\left(\|F\|_{V'} + \sup_{v\in V}\frac{a(u_g,\cdot)}{\|v\|_V}\right)\\ &\leq \frac{1}{\alpha}\|F\|_{V'} + \frac{M}{\alpha}\|u_g\|_V,\\ &\leq \frac{1}{\alpha}\|F\|_{V'} + \frac{M}{\alpha\beta}\|G\|_{Q'},\end{aligned} \tag{6.32}$$

where M is the continuity constant of $a(\cdot,\cdot)$.

Then we have

$$\begin{aligned}\|u\|_V = \|u_Z + u_g\|_V &\leq \|u_Z\|_V + \|u_g\|_V,\\ &\leq \frac{1}{\alpha}\|F\|_{V'} + \frac{M}{\alpha\beta}\|G\|_{Q'} + \frac{1}{\beta}\|G\|_{Q'},\\ &\leq \frac{1}{\alpha}\|F\|_{V'} + \frac{2M}{\alpha\beta}\|G\|_{Q'},\end{aligned} \tag{6.33}$$

making use of $M > \alpha$ (we have $\alpha\|u\|^2 \leq a(u,u) \leq M\|u\|^2$ for any $u \in V$). This gives the estimate for $\|u\|_V$.

To estimate $\|p\|_Q$, we rearrange the variational problem to get

$$b(p,v) = F'[v] = F[v] - a(u,v), \quad \forall v \in V. \tag{6.34}$$

As discussed previously, $F' \in Z^0$, hence this equation is solveable for p and we have

$$\|F'\|_{V'} \geq \beta\|p\|_Q, \tag{6.35}$$

Hence,

$$\begin{aligned}\|p\|_Q &\leq \frac{1}{\beta}\|F\|_{V'} + \frac{M}{\beta}\|u\|_V,\\ &\leq \frac{1}{\beta}\|F\|_{V'} + \frac{M}{\beta}\left(\frac{1}{\alpha}\|F\|_{V'} + \frac{2M}{\alpha\beta}\|G\|_{V'}\right),\\ &\leq \frac{2M}{\alpha\beta}\|F\|_{V'} + \frac{2M^2}{\alpha\beta^2}\|G\|_{Q'},\end{aligned} \tag{6.36}$$

as required, having used $M > \alpha$ again.

6.5 Solveability of Stokes equation

Now we return to our variational formulation of Stokes equation and consider the Brezzi conditions for it. In the case of Stokes, the operator B^* is the divergence operator. It can be shown (beyond the scope of this course) that B^* maps from the whole of V onto Q in this case, so the inf-sup condition holds. It can also be shown that a is coercive on the whole of V, i.e. there exists $\alpha > 0$ such that

$$a(v,v) \geq \alpha\|v\|_V^2. \tag{6.37}$$

This result is called Korn's identity (also beyond our scope). Then of course, a is in particular coercive on the divergence-free subspace Z. Then we immediately get solveability of the variational Stokes problem.

6.6 Discretisation of Stokes equations

To discretise the Stokes equations, we need to choose finite element spaces $V_h \subset V$ and $Q_h \subset Q$. Then we apply the Galerkin approximation, restricting the numerical solution (u_h, p_h) to $V_h \times Q_h$ as well as the test functions (v_h, q_h). If the bilinear form $c(X, Y)$ were coercive, we could immediately get existence, uniqueness and stability for the finite element discretisation. However, we don't have it. This means that in particular we may have issues with the uniqueness of p_h. To control these issues, we need to choose V_h and Q_h such that we have the discrete inf-sup condition

$$\inf_{q \in Q_h} \sup_{v \in V_h} \frac{b(v,q)}{\|v\|_V \|q\|_Q} \geq \beta_h, \tag{6.38}$$

with $\beta_h > 0$. Note that $\beta_h \neq \beta$ in general, but it does not matter as long as β_h is independent of the mesh size parameter h.

If the discrete inf-sup condition is satisfied then we just need to also check whether $a(\cdot,\cdot)$ is coercive on the discrete kernel Z_h defined by

$$Z_h = \{u \in V_h : b(u,q) = 0 \, \forall q \in Q_h\}. \tag{6.39}$$

Note that $Z_h \not\subset Z$ in general (unless V_h and Q_h have been specially chosen to allow that). However, the details do not matter since we already noted that $a(\cdot,\cdot)$ is coercive on all of V, so must be coercive on $Z_h \subset V$ in particular. Hence, as long as the discrete inf-sup condition is satisfied, we immediately get existence and uniqueness of solutions of the finite element approximation of Stokes equation from Theorem *Brezzi's conditions*, along with the stability bounds on (u_h, p_h), but with β replaced by β_h.

We are now in a position to estimate errors in the finite element approximation in a manner very similar to Céa's Lemma.

Theorem 6.8 *Let $V_h \subset V$ and $Q_h \subset Q$ be a pair of finite element spaces satisfying the discrete inf-sup condition for some $\beta_h > 0$. Then,*

$$\begin{aligned} \|u_h - u\|_V &\leq \frac{4MM_b}{\alpha\beta_h} E_u + \frac{M_b}{\alpha} E_p, \\ \|p_h - p\|_V &\leq \frac{3M^2 M_b}{\alpha\beta_h^2} E_u + \frac{3MM_b}{\alpha\beta_h} E_p. \end{aligned} \tag{6.40}$$

where M_b is the continuity constant of $b(\cdot,\cdot)$, and where we have the best approximation errors of u and p in V_h and Q_h respectively,

$$\begin{aligned} E_u &= \inf_{u_I \in V_h} \|u - u_I\|_V, \\ E_p &= \inf_{p_I \in Q_h} \|p - p_I\|_Q. \end{aligned} \tag{6.41}$$

Proof 6.9 *Since $V_h \subset V$ and $Q_h \subset Q$, we can choose $(v, q) \in V_h \times Q_h$ in both the original variational problem and the finite element variational problem and subtract one from the other, to obtain*

$$\begin{aligned} a(u_h - u, v) + b(v, p_h - p) &= 0, \quad \forall v \in V_h, \\ b(u_h - u, q) &= 0, \quad \forall q \in Q_h. \end{aligned} \tag{6.42}$$

This is the mixed finite element version of Galerkin orthogonality that we saw earlier in the course. Replacing $u = u - u_I + u_I$ and $p = p - p_I + p_I$ for $(u_I, p_I) \in V_h \times Q_h$ and rearranging, we get

$$\begin{aligned} a(u_h - u_I, v) + b(v, p_h - p_I) = F_{u_I, p_I}[v] := a(u - u_I, v) + b(v, p - p_I), \quad \forall v \in V_h, \\ b(u_h - u_I, q) = G_{u_I}[q] := b(u - u_I, q), \quad \forall q \in Q_h. \end{aligned} \tag{6.43}$$

Hence, from the stability bound,

$$\begin{aligned} \|u_h - u_I\|_V &\leq \frac{1}{\alpha} \|F_{u_I, p_I}\|_{V'} + \frac{2M}{\alpha \beta_h} \|G_{u_I}\|_{Q'}, \\ \|p_h - p_I\|_Q &\leq \frac{2M}{\alpha \beta_h} \|F_{u_I, p_I}\|_{V'} + \frac{2M^2}{\alpha \beta_h^2} \|G_{u_I}\|_{Q'}. \end{aligned} \tag{6.44}$$

Using continuity of $a(\cdot, \cdot)$ and $b(\cdot, \cdot)$, we have

$$\begin{aligned} \|F_{u_I, p_I}\|_{V'} \leq \sup_{v \in V} \frac{a(u - u_I, v)}{\|v\|_V} + \sup_{v \in V} \frac{b(v, p - p_I)}{\|v\|_V} \leq M \|u - u_I\|_V + M_b \|p - p_I\|_Q, \\ \|G_{u_I}\|_{Q'} = \sup_{p \in Q} \frac{b(u - u_I, p)}{\|p\|_Q} \leq M_b \|u - u_I\|_V. \end{aligned} \tag{6.45}$$

Substitution then gives

$$\|u_h - u_I\|_V \leq \frac{1}{\alpha} \left(M \|u - u_I\|_V + M_b \|p - p_I\|_Q \right) + \frac{2M}{\alpha \beta_h} M_b \|u - u_I\|_V. \tag{6.46}$$

We have

$$\beta_h \leq \inf_{q \in Q_h} \sup_{v \in V_h} \frac{b(v, q)}{\|q\|_Q \|v\|_V} \leq M_b,$$

and hence,

$$\|u_h - u_I\|_V \leq \frac{3 M M_b}{\alpha \beta_h} \|u - u_I\|_V + \frac{M_b}{\alpha} \|p - p_I\|_Q, \tag{6.47}$$

and

$$\begin{aligned} \|p_h - p_I\|_Q &\leq \frac{2M}{\alpha \beta_h} \left(M \|u - u_I\|_V + M_b \|p - p_I\|_Q \right) + \frac{2M^2}{\alpha \beta_h^2} M_b \|u - u_I\|_V \\ &\leq \frac{3M^2 M_b}{\alpha \beta_h^2} \|u - u_I\|_V + \frac{2 M M_b}{\alpha \beta_h} \|p - p_I\|_Q. \end{aligned} \tag{6.48}$$

We then use the triangle inequality to write

$$\begin{aligned} \|u - u_h\|_V &\leq \|u - u_I\|_V + \|u_h - u_I\|_V, \\ &\leq \frac{4 M M_b}{\alpha \beta_h} \|u - u_I\|_V + \frac{M_b}{\alpha} \|p - p_I\|_Q, \end{aligned} \tag{6.49}$$

$$\begin{aligned} \|p - p_h\|_Q &\leq \|p - p_I\|_Q + \|p_h - p_I\|_Q, \\ &\leq \frac{3M^2 M_b}{\alpha \beta_h^2} \|u - u_I\|_V + \frac{3 M M_b}{\alpha \beta_h} \|p - p_I\|_Q. \end{aligned} \tag{6.50}$$

Finally, taking the infimum over the all $u_I \in V$ and all $p_I \in Q$ gives the result.

This theorem tells us that if we can approximate the solution (u, p) well in $V_h \times Q_h$, then the finite element approximation error will also be small.

For scalar H^1 elliptic problems like the Poisson equation that we studied earlier in the course, finding a suitable V_h is easy, as any continuous finite element space will do. In contrast, for Stokes equation it is not straightforward to find pairs of finite element spaces $V_h \times Q_h$ that satisfy this discrete inf-sup condition. For example, the simplest idea of trying Q_h to be P1 (linear Lagrange elements on triangles) and V_h to be $(P1)^d$ (linear Lagrange elements for each Cartesian component of velocity from 1 up to the dimension d) does not work in general. We call this combination P1-P1.

Exercise 6.10 *Consider a square domain divided into 4 smaller and equal squares, and then subdivide the squares into right-angled triangles so all the hypotenuses meet in the middle (like the UK flag). Show that there exists $p \in Q_h$ such that $b(v, p) = 0$ for all $v \in V_h$. (Don't forget to include the boundary conditions for V_h and the mean zero condition for p.) Conclude that the inf-sup condition does not hold.*

We now discuss some examples of finite element pairs that do satisfy the inf-sup condition with $\beta_h > 0$ independent of h.

6.7 The MINI element

In general, the choice P1-P1 produces $\beta_h \to 0$ as $h \to 0$: the discretisation is not stable. This means that the image of the divergence applied to V_h does not converge to Q as $h \to 0$. The way to fix this is to enrich the $(P1)^d$ space for velocity, so that the image is larger. For the MINI element, this is done by considering the following finite element, P1+B3.

Definition 6.11 (P1+B3) *The P1+B3 element $(K, P, \mathcal{N})$ is given by:*

1. *K is a triangle.*
2. *The shape functions are linear combinations of linear functions and cubic "bubble" functions that vanish on the boundary of K.*
3. *The nodal variables are point evaluations at the vertices plus point evaluation at the triangle centre.*

We then take V_h as the $(P1+B3)^d$ continuous finite element space (i.e. each Cartesian component of the functions in V_h is from $P1 + B3$. We choose $P1$ for Q_h.

To prove that the MINI element satisfies the inf-sup condition, we use the following result.

Lemma 6.12 (Fortin's trick) *Assume that the inf-sup condition holds for $b(v,q)$ over $Vtimes Q$ with inf-sup constant $beta>0$. If there exists a linear operator $\Pi_h : V \to V_h$ such that*

$$\begin{aligned} b(v - \Pi_h v, q) = 0, \quad \forall v \in V,\, q \in Q_h, \\ \|\Pi_h v\|_V \leq C_\Pi \|v\|_V, \end{aligned} \tag{6.51}$$

then the discrete inf-sup condition holds.

Proof 6.13 *For any $q_h \in Q_h$, we have*

$$\sup_{v_h \in V_h} \frac{b(v_h, q_h)}{\|v_h\|_V} \geq \sup_{v \in V} \frac{b(\Pi_h v, q_h)}{\|\Pi_h v\|_V} = \sup_{v \in V} \frac{b(v, q_h)}{\|\Pi_h v\|_V} \geq \sup_{v \in V} \frac{b(v, q_h)}{C_\Pi \|v\|_V} \geq \frac{\beta}{C_\Pi} \|q_h\|_Q, \tag{6.52}$$

and rearranging and taking the infimum over $q_h \in Q_h$ gives

$$\inf_{q_h \in Q_h} \sup_{v_h \in V_h} \frac{b(v_h, q_h)}{\|q_h\|_Q \|v_h\|_V} = \beta_h := \frac{\beta}{C_\Pi}. \tag{6.53}$$

The following lemma gives a practical way to find Π_h.

Lemma 6.14 *Assume that there exist two maps $\Pi_1, \Pi_2 : V \to V_h$, with*

$$\begin{aligned} \|\Pi_1 v\|_V \leq c_1\|v\|_V,\, \forall v \in V, \\ \|\Pi_2(I-\Pi_1)v\|_V \leq c_2\|v\|_V,\, \forall v \in V, \\ b(v-\Pi_2 v, q_h) = 0,\, \forall v\in V,\, q_h \in Q_h, \end{aligned} \tag{6.54}$$

where the constants c_1 and c_2 are independent of h. Then the operator Π_h, defined by

$$\Pi_h u = \Pi_1 u + \Pi_2(u - \Pi_1 u), \tag{6.55}$$

satisfies the conditions of Fortin's trick.

Proof 6.15 *We have*

$$\begin{aligned} b(\Pi_h w, q_h) = b(\Pi_2(w-\Pi_1)w, q_h) + b(\Pi_1 w, q_h), \\ = b(w - \Pi_1 w, q_h) + b(\Pi_1 w, q_h) \\ = b(w, q_h), \end{aligned} \tag{6.56}$$

which gives the second condition of Fortin's trick, and

$$\|\Pi_h w\|_V \leq \|\Pi_2(w-\Pi_1 w)\|_V + \|\Pi_1 w\|_V \leq (c_1+c_2)\|w\|_V. \tag{6.57}$$

For continuous finite element spaces, the Clement operator (which we shall not describe here) satifies the condition on Π_1. In fact, the Clement operator generally satisfies

$$|v - \Pi_1 v|_{H^m(K)} \leq c\left(\sum_{\bar{K}'\cap\bar{K}\neq 0} h_{K'}^{1-m}\|v\|_{H^1(K)}\right) \tag{6.58}$$

where $\bar{K}$ is the closure of any triangle K, and the sum is taken over all triangles K' that share an edge or a vertex with triangle K.

We now use this technique to prove the discrete inf-sup condition for the MINI element.

Theorem 6.16 *The MINI element satisfies the discrete inf-sup condition.*

Proof 6.17 *We can use the Clement operator for Π_1. $\Pi_2 : V \to (B_3)^2 \subset V_h$ (i.e. the subspace of V_h of functions that vanish on all vertices (and hence all edges) is defined via*

$$0 = b(\Pi_2 v - v, q_h),\, \forall q_h \in Q_h. \tag{6.59}$$

This is well defined since

$$\begin{aligned} b(\Pi_2 v - v, q_h) = \int_\Omega q_h \nabla\cdot(\Pi_2 v - v)\,d\,x \\ = \int_\Omega (v - \Pi_2 v)\nabla q_h\,d\,x, \end{aligned} \tag{6.60}$$

where we were allowed to integrate by parts since $v, \Pi_2 v, q_h$ *are all in* $H^1(\Omega)$*. We see that our definition can be satisfied by picking* $\Pi_2 v$ *to be the function in $(B_3)^2$ such that*

$$\int_K \Phi_2 v d\,xx = \int_K v d\,xx, \tag{6.61}$$

for each triangle K.

It can be shown using an inverse inequality (we will take it as read here) that

$$\|\Pi_2 v\|_{H^r(K)} \leq ch_K^{-r}\|v\|_{L^2(K)},\, \forall v \in V,\, r = 0, 1. \tag{6.62}$$

Combining this with Equation (6.58) *gives Equation* (6.54) *and hence we have shown that* Π_h *has the properties needed for Fortin's trick.*

Part II

Implementation Exercise

CHAPTER ZERO

THE IMPLEMENTATION EXERCISE

The object of the implementation exercise is to gain an understanding of the finite element method by producing a working one and two dimensional finite element solver library. Along the way you will have the opportunity to pick up valuable scientific computing skills in coding, software engineering and rigorous testing.

This part of the module is very practical, and there are never conventional lectures for it, even when everything is taught on campus. Each week you should work through the notes and videos until you come to an exercise. Each exercise will invite you to implement another part of a finite element implementation, so that by the end of the term we will be solving finite element problems.

Along the way, there will be the opportunity to get help and feedback through the module Piazza board, weekly online labs, and through pull requests for feedback in weeks 4 and 7.

0.1 Formalities and marking scheme

The implementation exercise is due at 1300 on Friday 22 March 2024. Submission is via GitHub: the last commit pushed to GitHub and dated before the deadline will be marked.

The marking scheme will be as follows:

Excellent (18-20)
: All parts of the implementation are correct and all tests pass. The code style is always very clear and the implementation of every exercise is transparent and elegant.

Good (14-17)
: The implementation is correct but let down somewhat by poor coding style. Alternatively, submissions which are correct and well written up to and including solving the Helmholtz problem but which do not include a correct solution to boundary conditions will earn an upper second.

Pass (10-13)
: There are significant failings in the implementation resulting in many test failures, and/or the coding style is sufficiently poor that the code is hard to understand.

Fail (0-9)
: The implementation is substantially incomplete. Correct implementations may have been provided for some of the earlier exercises but the more advanced parts of the implementation exercise have not been attempted or do not work.

Code execution performance is not a primary concern of this module, however the code must still be algorithmically correct. This means not just returning the correct answer but also having the correct algorithmic complexity. Occasionally students submit code that uses quadratic algorithms where linear ones would be possible. The result is that examples that should run in seconds and take megabytes of memory instead take gigabytes of memory and many hours to complete. Such submissions are incorrect, and will be marked as such.

0.2 Obtaining the skeleton code

This section assumes you've already done everything to *set up the software tools you need*.

0.2.1 Set up a folder to hold the repository and virtual environment

You can call this folder anything you like, and store it anywhere that suits you, though don't move it once you've created it as this will break the virtual environment. Suppose you would like to keep the new folder in a folder called `docs` in your home directory. We first open a terminal and switch to the folder:

```
$ cd docs
```

Note that *$* is the command prompt (which might be a different character such as *%* or *>* for you). You don't type the prompt. Start with *cd*. Next we create the folder we'll use for this course. Suppose we choose to call it `finite-element`, then we would type:

```
$ mkdir finite-element
```

mkdir stands for "make directory". *Directory* is an alternative term to *folder*. Finally we switch ("change directory") into that folder:

```
$ cd finite-element
```

0.2.2 Setting up your venv

We're going to use a Python venv. This is a private Python environment in which we'll install the packages we need, including our own implementation exercise. This minimises interference between this project and anything else which might be using Python on the system. With your current working folder set to the course folder, run:

```
$ python3 -m venv fe_venv
```

If your Python interpreter has a different name (e.g. *python3.11* or *py*) then you type that instead.

0.2.3 Activating your venv

Every time you want to work on the implementation exercise, you need to activate the venv. On Linux or Mac do this with:

```
$ source fe_venv/bin/activate
```

while on Windows the command is:

```
> source fe_venv/Scripts/activate
```

Obviously if you are typing this in a directory other than the one containing the venv, you need to modify the path accordingly.

0.2.4 Setting up your repository

We're using a tool called GitHub classroom to automate the creation of your copies of the repository. To create your repository, click here.

0.2.5 Cloning a local copy

At the command line on your working machine type:

```
$ git clone <url> finite-element-course
```

Substituting your git repository url for <url>. Your git repository url can be found by clicking on *clone or download* at the top right of your repository page on GitHub.

0.2.6 Installing the course Python package

Your git repository contains a Python package. Installing this will cause the other Python packages on which it depends to be installed into your venv, and will create various visualisation scripts you'll need later in the module. Run:

```
$ python -m pip install -e finite-element-course/
```

0.3 Skeleton code documentation

There is web documentation for the complete *fe_utils package*. There is also an alphabetical index and a search page.

0.4 How to do the implementation exercises

The implementation exercises build up a finite element library from its component parts. Quite a lot of the coding infrastructure you will need is provided already. Your task is to write the crucial mathematical operations at key points. The mathematical operations required are described on this website, interspersed with exercises which require you to implement and test parts of the mathematics.

The code on which you will build is in the `fe_utils` directory of your repository. The code has embedded documentation which is used to build the *fe_utils package* web documentation.

As you do the exercises, **commit your code** to your repository. This will build up your finite element library. You should commit code early and often - small commits are easier to understand and debug than large ones.

0.5 Testing your work

As you complete the exercises, there will often be test scripts which exercise the code you have just written. These are located in the `test` directory and employ the pytest testing framework. You run the tests with:

```
$ py.test test_script.py
```

from the bash command line, replacing `test_script.py` with the appropriate test file name. The `-x` option to `py.test` will cause the test to stop at the first failure it finds, which is often the best place to start fixing a problem. For those familiar with debuggers, the `--pdb` option will drop you into the Python debugger at the first error.

You can also run all the tests by running `py.test` on the tests directory. This works particularly well with the -x option, resulting in the tests being run in course order and stopping at the first failing test:

```
$ py.test -x tests/
```

0.6 Coding style and commenting

Computer code is not just functional, it also conveys information to the reader. It is important to write clear, intelligible code. **The readability and clarity of your code will count for marks**.

The Python community has agreed standards for coding, which are documented in PEP8. There are programs and editor modes which can help you with this. The skeleton implementation follows PEP8 quite closely. You are encouraged, especially if you are a more experienced programmer, to follow PEP8 in your implementation. However nobody is going to lose marks for PEP8 failures.

0.7 Getting help

It's expected that you will find there are tasks in the implementation exercise that you don't know how to do. Your first port of call should be the Ed forum, followed by the weekly live lab sessions.

0.7.1 Using Ed

The key advantage of asking for help on Ed is that you can do this at any point during the week, whenever you are stuck. The whole class can see the forum, but you can choose to publish anonymously so nobody need know who asked the question. You should also watch the other questions as they appear on Ed, because you will find that you learn a lot from what other people ask, as well as the answers they get. Other students might notice issues that didn't even occur to you!

Do please try to answer other students' questions. Doing so is actually a really effective way of understanding the work better, since you will be looking at the tasks from another student's perspective.

0.7.2 Formulating a good question

One of the key skills in getting help with code is to ask the question in a structured way which provides all the information required by the person helping you. Not only does this radically increase the chances of getting a useful response first time, but often the process of thinking through how to ask the question leads you to its solution before you even ask. Please review the information from the second year Principles of Programming instructions on raising an issue.

Note

Please don't post large pieces of code to Piazza. Just post minimal examples if they help. However always commit and push your work, and post the git commit hash in the repository. The instructor can always find your work from the git hash, so long as you've pushed to GitHub.

0.8 Tips and tricks for the implementation exercise

Work from the documentation.
The notes, and particularly the exercise specifications, contain important information about how and what to implement. If you just read the source code then you will miss out on important information.

Read the hints
The pink sections in the notes starting with a lightbulb are hints. Usually they contain suggestions about how to go about writing your answer, or suggest Python functions which you might find useful.

Don't forget the 1D case
Your finite element library needs to work in one and two dimensions.

Return a `numpy.array()`
Many of the functions you have to write return arrays. Make sure you actually return an array and not a list (it's usually fine to build the answer as a list, but convert it to an array before you return it).

CHAPTER

ONE

NUMERICAL QUADRATURE

The core computational operation with which we are concerned in the finite element method is the integration of a function over a known reference element. It's no big surprise, therefore, that this operation will be at the heart of our finite element implementation.

The usual way to efficiently evaluate arbitrary integrals numerically is numerical quadrature. This basic idea will already be familiar to you from undergraduate maths (or maybe even high school calculus) as it's the generalisation of the trapezoidal rule and Simpson's rule for integration.

The core idea of quadrature is that the integral of a function $f(X)$ over an element e can be approximated as a weighted sum of function values evaluated at particular points:

$$\int_e f(X) = \sum_q f(X_q)w_q + O(h^n) \tag{1.1}$$

we term the set $\{X_q\}$ the set of *quadrature points* and the corresponding set $\{w_q\}$ the set of *quadrature weights*. A set of quadrature points and their corresponding quadrature weights together comprise a *quadrature rule* for e. For an arbitrary function f, quadrature is only an approximation to the integral. The global truncation error in this approximation is invariably of the form $O(h^n)$ where h is the diameter of the element.

If f is a polynomial in X with degree p such that $p \leq n-2$ then it is easy to show that integration using a quadrature rule of degree n results in exactly zero error.

Definition 1.1 *The* degree of precision *of a quadrature rule is the largest p such that the quadrature rule integrates all polynomials of degree p without error.*

1.1 Exact and incomplete quadrature

In the finite element method, integrands are very frequently polynomial. If the quadrature rule employed for a particular interval has a sufficiently high degree of precision such that there is no quadrature error in the integration, we refer to the quadrature as *exact* or *complete*. In any other case we refer to the quadrature as *incomplete*.

Typically, higher degree quadrature rules have more quadrature points than lower degree rules. This results in a trade-off between the accuracy of the quadrature rule and the number of function evaluations, and hence the computational cost, of an integration using that rule. Complete quadrature results in lower errors, but if the error due to incomplete quadrature is small compared with other errors in the simulation, particularly compared with the discretisation error, then incomplete quadrature may be advantageous.

1.2 Examples in one dimension

We noted above that a few one dimensional quadrature rules are commonly taught in introductory integration courses. The first of these is the midpoint rule:

$$\int_0^h f(X)\mathrm{d}X = hf(0.5h) + O(h^3) \tag{1.2}$$

In other words, an approximation to the integral of f over an interval can be calculated by multiplying the value of f at the mid-point of the interval by the length of the interval. This amounts to approximating the function over the integral by a constant value.

If we improve our approximation of f to a straight line over the interval, then we arrive at the trapezoidal (or trapezium) rule:

$$\int_0^h f(X)\mathrm{d}X = \frac{h}{2}f(0) + \frac{h}{2}f(h) + O(h^4) \tag{1.3}$$

while if we employ a quadratic function then we arrive at Simpson's rule:

$$\int_0^h f(X)\mathrm{d}X = \frac{h}{6}f(0) + \frac{2h}{3}f\left(\frac{h}{2}\right) + \frac{h}{6}f(h) + O(h^5) \tag{1.4}$$

1.3 Reference cells

As a practical matter, we wish to write down quadrature rules as arrays of numbers, independent of h. In order to achieve this, we will write the quadrature rules for a single, *reference cell*. When we wish to actually integrate a function over cell, we will change coordinates to the reference cell. We will return to the mechanics of this process later, but for now it means that we need only consider quadrature rules on the reference cells we choose.

A commonly employed one dimensional reference cell is the unit interval $[0, 1]$, and that is the one we shall adopt here (the other popular alternative is the interval $[-1, 1]$, which some prefer due to its symmetry about the origin).

In two dimensions, the cells employed most commonly are triangles and quadrilaterals. For simplicity, in this course we will only consider implementing the finite element method on triangles. The choice of a reference interval implies a natural choice of reference triangle. For the unit interval the natural correspondence is with the triangle with vertices $[(0, 0), (1, 0), (0, 1)]$, though different choices of vertex numbering are possible.

1.3.1 Reference cell topology

A cell is composed of *topological entities*, that is to say vertices, edges, faces and so forth. The topology of the cell is given by the connectivity of its entities, for example which vertices make up each edge. It is useful to define some terms to describe the cell topology:

Definition 1.2 *The* dimension *of a cell is the maximal dimension of the topological entities that make up the cell.*

Definition 1.3 *A topological entity of* codimension n *is a topological entity of dimension* $d - n$ *where* d *is the dimension of the cell.*

Armed with these definitions we are able to define names for topological entities of various dimension and codimension:

entity name	dimension	codimension
vertex	0	
edge	1	
face	2	
facet		1
cell		0

The cells can be polygons or polyhedra of any shape, however in this course we will restrict ourselves to intervals and triangles. The only other two-dimensional cells frequently employed are quadrilaterals.

1.3.2 Reference cell entities

The topological entities of each dimension in a cell are distinguished by giving them unique numbers. We will identify topological entities by an index pair (d, i) where i is the index of the entity within the set of d-dimensional entities.

The particular choices of numbering we will use are shown in Fig. 1.1. The numbering is a matter of convention: that adopted here is that edges share the number of the opposite vertex. The orientation of the edges is also shown, this is always from the lower numbered vertex to the higher numbered one.

The `ReferenceCell` class stores the local topology of the reference cell. Read the source and ensure that you understand the way in which this information is encoded.

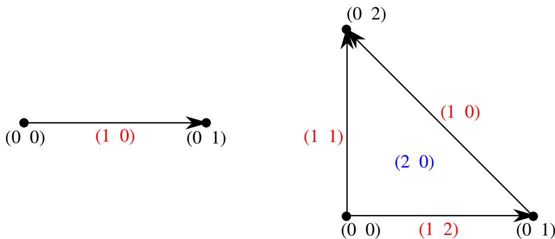

Fig. 1.1: Local numbering and orientation of the reference entities.

1.3.3 Python implementations of reference elements

The `ReferenceCell` class provides Python objects encoding the geometry and topology of the reference cell. At this stage, the relevant information is the dimension of the reference cell and the list of vertices. The topology will become important in the following chapters. The reference cells we will require for this course are the `ReferenceInterval` and `ReferenceTriangle`.

1.4 Quadrature rules on reference elements

Having adopted a convention for the reference element, we can simply express quadrature rules as lists of quadrature points with corresponding quadrature weights. For example Simpson's rule becomes:

$$\begin{aligned} w &= \left[\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\right] \\ X &= \left[(0), (0.5), (1)\right]. \end{aligned} \tag{1.5}$$

We choose to write the quadrature points as 1-tuples for consistency with the n-dimensional case, in which the points will be n-tuples.

The lowest order quadrature rule on the reference triangle is a single point quadrature:

$$\begin{aligned} w &= \left[\frac{1}{2}\right] \\ X &= \left[\left(\frac{1}{3}, \frac{1}{3}\right)\right] \end{aligned} \tag{1.6}$$

This rule has a degree of precision of 1.

Hint

The weights of a quadrature rule always sum to the volume of the reference element. Why is this?

1.5 Legendre-Gauß quadrature in one dimension

The finite element method will result in integrands of different polynomial degrees, so it is convenient if we have access to quadrature rules of arbitrary degree on demand. In one dimension the Legendre-Gauß quadrature rules are a family of rules of arbitrary precision which we can employ for this purpose. Helpfully, numpy provides an implementation which we are able to adopt. The Legendre-Gauß quadrature rules are usually defined for the interval $[-1, 1]$ so we need to change coordinates in order to arrive at a quadrature rule for our reference interval:

$$\begin{aligned} X_q &= \frac{X'_q + 1}{2} \\ w_q &= \frac{w'_q}{2} \end{aligned} \tag{1.7}$$

where $(\{X'_q\}, \{w'_q\})$ is the quadrature rule on the interval $[-1, 1]$ and $(\{X_q\}, \{w_q\})$ is the rule on the unit interval.

Legendre-Gauß quadrature on the interval is optimal in the sense that it uses the minimum possible number of points for each degree of precision.

1.6 Extending Legendre-Gauß quadrature to two dimensions

We can form a unit square by taking the Cartesian product of two unit intervals: $(0, 1) \otimes (0, 1)$. Similarly, we can form a quadrature rule on a unit square by taking the product of two interval quadrature rules:

$$\begin{aligned} X_{\text{sq}} &= \{(x_p, x_q) \mid x_p, x_q \in X\} \\ w_{\text{sq}} &= \{w_p w_q \mid w_p, w_q \in w\} \end{aligned} \tag{1.8}$$

where (X, w) is an interval quadrature rule. Furthermore, the degree of accuracy of $(X_{\text{sq}}, w_{\text{sq}})$ will be the same as that of the one-dimensional rule.

However, we need a quadrature rule for the unit triangle. We can achieve this by treating the triangle as a square with a zero length edge. The Duffy transform maps the unit square to the unit triangle:

$$(x_{\text{tri}},\ y_{\text{tri}}) = (x_{\text{sq}},\ y_{\text{sq}}(1 - x_{\text{sq}})) \tag{1.9}$$

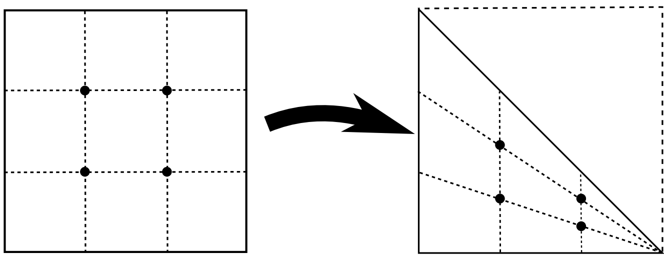

Fig. 1.2: The Duffy transform maps a square to a triangle by collapsing one side.

By composing the Duffy transform with (1.8) we can arrive at a quadrature rule for the triangle:

$$\begin{aligned} X_{\text{tri}} &= \{(x_p, x_q(1 - x_p)) \mid x_p \in X_h, x_q \in X_v\} \\ w_{\text{tri}} &= \{w_p w_q(1 - x_p) \mid w_p \in w_h, w_q \in w_v\} \end{aligned} \tag{1.10}$$

where (X_v, w_v) is a reference interval quadrature rule with degree of precision n and (X_h, w_h) is a reference interval quadrature rule with degree of precision $n+1$. The combined quadrature rule $(X_{\text{tri}}, w_{\text{tri}})$ will then be n. The additional degree of precision required for (X_h, w_h) is because the Duffy transform effectively increases the polynomial degree of the integrand by one.

1.7 Implementing quadrature rules in Python

The *fe_utils.quadrature* module provides the *QuadratureRule* class which records quadrature points and weights for a given *ReferenceCell*. The *gauss_quadrature()* function creates quadrature rules for a prescribed degree of precision and reference cell.

Exercise 1.4 *The* `integrate()` *method is left unimplemented. Using* (1.1)*, implement this method.*

A test script for your method is provided in the `test` *directory as* `test_01_integrate.py`*. Run this script to test your code:*

```
py.test test/test_01_integrate.py
```

from the Bash command line. Make sure you commit your modifications and push them to your fork of the course repository.

Hint

You can implement *integrate()* in one line using a list comprehension and `numpy.dot()`.

Hint

Don't forget to activate your Python venv!

CHAPTER

TWO

CONSTRUCTING FINITE ELEMENTS

At the core of the finite element method is the representation of finite-dimensional function spaces over elements. This concept was formalised by [Cia02]:

Definition 2.1 *A* finite element *is a triple* (K, P, N) *in which* K *is a cell,* P *is a space of functions* $K \to \mathbb{R}^n$ *and* N*, the set of* nodes*, is a basis for* P^**, the dual space to* P*.*

Note that this definition includes a basis for P^*, but not a basis for P. It turns out to be most convenient to specify the set of nodes for an element, and then derive an appropriate basis for P from that. In particular:

Definition 2.2 *Let* $N = \{\phi_j^*\}$ *be a basis for* P^**. A* nodal basis*,* $\{\phi_i\}$ *for* P *is a basis for* P *with the property that* $\phi_j^*(\phi_i) = \delta_{ij}$*.*

2.1 A worked example

To illustrate the construction of a nodal basis, let's consider the linear polynomials on a triangle. We first need to define our reference cell. The obvious choice is the triangle with vertices $\{(0,0),(1,0),(0,1)\}$

Functions in this space have the form $a + bx + cy$. So the function space has three unknown parameters, and its basis (and dual basis) will therefore have three members. In order to ensure the correct continuity between elements, the dual basis we need to use is the evaluation of the function at each of the cell vertices. That is:

$$\begin{aligned} \phi_0^*(f) &= f((0,0)) \\ \phi_1^*(f) &= f((1,0)) \\ \phi_2^*(f) &= f((0,1)) \end{aligned} \tag{2.1}$$

We know that $\phi_i((x,y))$ has the form $a_i + b_i x + c_i y$ so now we can use the definition of the nodal basis to determine the unknown coefficients:

$$\begin{pmatrix} \phi_0^*(\phi_i) \\ \phi_1^*(\phi_i) \\ \phi_2^*(\phi_i) \end{pmatrix} = \begin{pmatrix} \delta_{i,0} \\ \delta_{i,1} \\ \delta_{i,2} \end{pmatrix} \tag{2.2}$$

So for ϕ_0 we have:

$$\begin{pmatrix} \phi_0^*(\phi_0) \\ \phi_1^*(\phi_0) \\ \phi_2^*(\phi_0) \end{pmatrix} = \begin{pmatrix} \phi_0((0,0)) \\ \phi_0((1,0)) \\ \phi_0((0,1)) \end{pmatrix} = \begin{pmatrix} a_0 + b_0(0) + c_0(0) \\ a_0 + b_0(1) + c_0(0) \\ a_0 + b_0(0) + c_0(1) \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.3}$$

Which has solution $\phi_0 = 1 - x - y$. We can write the equations for all the basis functions at once as a single matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

By which we establish that the full basis is given by:

$$\begin{aligned} \phi_0 &= 1 - x - y \\ \phi_1 &= x \\ \phi_2 &= y \end{aligned} \tag{2.5}$$

2.2 Types of node

We have just encountered nodes given by the evaluation of the function at a given point. Other forms of functional are also suitable for use as finite element nodes. Examples include the integral of the function over the cell or some sub-entity and the evaluation of the gradient of the function at some point. For some vector-valued function spaces, the nodes may be given by the evaluation of the components of the function normal or tangent to the boundary of the cell at some point.

In this course we will only consider point evaluation nodes. The implementation of several other forms of node are covered in [Kir04].

2.3 The Lagrange element nodes

The number of coefficients of a degree p polynomial in d dimensions is given by the combination $\binom{p+d}{d}$. The simplest set of nodes which we can employ is simply to place these nodes in a regular grid over the reference cell. Given the classical relationship between binomial coefficients and Pascal's triangle (and between trinomial coefficients and Pascal's pyramid), it is unsurprising that this produces the correct number of nodes.

The set of equally spaced points of degree p on the triangle is:

$$\left\{ \left(\frac{i}{p}, \frac{j}{p} \right) \middle| 0 \leq i + j \leq p \right\} \tag{2.6}$$

The finite elements with this set of nodes are called the *equispaced Lagrange* elements and are the most commonly used elements for relatively low order computations.

While this is the simplest node ordering to construct, when we come to build finite element spaces over a whole computational mesh in Section 4, it will be much more straightforward if the nodes are numbered in topological order. That is to say, the lowest numbered nodes are those associated with the vertices, followed by those associated with the edges and finally, in two dimensions, those associated with the cell. For reasons that will become apparent when we consider the continuity of finite element spaces, the nodes associated with the edges need to be in edge orientation order. That is to say, the node number increases as one moves along the edge in the direction of the arrow. In two dimensions, the ordering of nodes in the cell interior is arbitrary.

Fig. 2.1: The numbering of nodes for the degree 1, 2, and 3 equispaced Lagrange elements on triangles. Black nodes are associated with vertices, red nodes with edges and blue nodes with the cell (face). Note that the numbering of nodes on edges follows the numbering of the edges in Fig. 1.1.

Note

At higher order the equispaced Lagrange basis is poorly conditioned and creates unwanted oscillations in the solutions. However for this course Lagrange elements will be sufficient.

Exercise 2.3 *Implement* `lagrange_points()`. *Make sure your algorithm also works for one-dimensional elements. Some basic tests for your code are to be found in* `test/test_02_lagrange_points.py`. *You can also test your lagrange points on the triangle by running:*

```
plot_lagrange_points degree
```

Where `degree` *is the degree of the points to plot.*

2.4 Solving for basis functions

The matrix in (2.3) is a *generalised Vandermonde*[1] matrix . Given a list of points $(x_i, y_i) \in \mathbb{R}^2, 0 \leq i < m$ the corresponding degree n generalised Vandermonde matrix is given by:

$$\mathrm{V} = \begin{bmatrix} 1 & x_0 & y_0 & x_0^2 & x_0 y_0 & y_0^2 & \ldots & x_0^n & x_0^{n-1} y_0 & \ldots & x_0 y_0^{n-1} & y_0^n \\ 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 & \ldots & x_1^n & x_1^{n-1} y_1 & \ldots & x_1 y_1^{n-1} & y_1^n \\ \vdots & & & & & & & & & & & \\ 1 & x_m & y_m & x_m^2 & x_m y_m & y_m^2 & \ldots & x_m^n & x_m^{n-1} y_m & \ldots & x_m y_m^{n-1} & y_m^n \end{bmatrix} \tag{2.7}$$

If we construct the Vandermonde matrix for the nodes of a finite element, then the equation for the complete set of basis function polynomial coefficients is:

$$\mathrm{VC} = \mathrm{I} \tag{2.8}$$

where the j-th column of C contains the polynomial coefficients of the basis function corresponding to the j-th node. For (2.8) to be well-posed, there must be a number of nodes equal to the number of coefficients of a degree n polynomial. If this is the case, then it follows immediately that:

$$\mathrm{C} = \mathrm{V}^{-1} \tag{2.9}$$

The same process applies to the construction of basis functions for elements in one or three dimensions, except that the Vandermonde matrix must be modified to exclude powers of y (in one dimension) or to include powers of z.

Note

Here we employ a monomial basis to represent polynomial spaces: any polynomial is given as a linear sum of monomials such as x, xy or x^2. This basis becomes increasingly ill-conditioned at higher order, so it may be advantageous to employ a different basis in the construction of the Vandermonde matrix. See [Kir04] for an example.

Exercise 2.4 *Use (2.7) to implement* `vandermonde_matrix()`. *Think carefully about how to loop over each row to construct the correct powers of x and y. For the purposes of this exercise you should ignore the* `grad` *argument.*

Tests for this function are in `test/test_03_vandermonde_matrix.py`

Hint

You can use numpy array operations to construct whole columns of the matrix at once.

[1] A Vandermonde matrix is the one-dimensional case of the generalised Vandermonde matrix.

2.5 Implementing finite elements in Python

The *Ciarlet triple* (K, P, N) also provides a good abstraction for the implementation of software objects corresponding to finite elements. In our case K will be a `ReferenceCell`. In this course we will only implement finite element spaces consisting of complete polynomial spaces so we will specify P by providing the maximum degree of the polynomials in the space. Since we will only deal with point evaluation nodes, we can represent N by a series of points at which the evaluation should occur.

Exercise 2.5 *Implement the rest of the* `FiniteElement` `__init__()` *method. You should construct a Vandermonde matrix for the nodes and invert it to create the basis function coefs. Store these as* `self.basis_coefs`.

Some basic tests of your implementation are in `test/test_04_init_finite_element.py`.

Hint

The `numpy.linalg.inv()` function may be used to invert the matrix.

2.6 Implementing the Lagrange Elements

The `FiniteElement` class implements a general finite element object assuming we have provided the cell, polynomial, degree and nodes. The `LagrangeElement` class is a subclass of `FiniteElement` which will implement the particular case of the equispaced Lagrange elements.

Exercise 2.6 *Implement the* `__init__()` *method of* `LagrangeElement`. *Use* `lagrange_points()` *to obtain the nodes. For the purpose of this exercise, you may ignore the* `entity_nodes` *argument.*

***After** you have implemented* `tabulate()` *in the next exercise, you can use* `plot_lagrange_basis_functions` *to visualise your Lagrange basis functions.*

2.7 Tabulating basis functions

A core operation in the finite element method is integrating expressions involving functions in finite element spaces. This is usually accomplished using *numerical quadrature*. This means that we need to be able to evaluate the basis functions at a set of quadrature points. The operation of evaluating a set of basis functions at a set of points is called *tabulation*.

Recall that the coefficients of the basis functions are defined with respect to the monomial basis in (2.9). To tabulate the basis functions at a particular set of points therefore requires that the monomial basis be evaluated at that set of points. In other words, the Vandermonde matrix needs to be evaluated at the quadrature points. Suppose we have a set of points $\{X_i\}$ and a set of basis functions $\{\phi_j\}$ with coefficents with respect to the monomial basis given by the matrix C. Then the tabulation matrix is given by:

$$T_{ij} = \phi_j(X_i) = \sum_b V(X_i)_b C_{bj} = (V(X_:) \cdot C)_{ij} \tag{2.10}$$

Exercise 2.7 *Implement* `tabulate()`. *You can use a Vandermonde matrix to evaluate the polynomial terms and take the matrix product of this with the basis function coefficients. The method should have at most two executable lines. For the purposes of this exercise, ignore the* `grad` *argument.*

The test file `test/test_05_tabulate.py` *checks that tabulating the nodes of a finite element produces the identity matrix.*

2.8 Gradients of basis functions

A function f defined over a single finite element with basis $\{\phi_i\}$ is represented by a weighted sum of that basis:

$$f = \sum_i f_i \phi_i \tag{2.11}$$

In order to be able to represent and solve PDEs, we will naturally also have terms incorporating derivatives. Since the coefficients f_i are spatially constant, derivative operators pass through to apply to the basis functions:

$$\nabla f = \sum_i f_i \nabla \phi_i \tag{2.12}$$

This means that we will need to be able to evaluate the gradient of the basis functions at quadrature points. Recall once again that the basis functions are evaluated by multiplying the Vandermonde matrix evaluated at the relevant points by the matrix of basis function coefficients. Hence:

$$\nabla \phi(X) = \nabla \left(V(X) \cdot C\right) = \left(\nabla V(X)\right) \cdot C \tag{2.13}$$

The last step follows because C is not a function of X, so it passes through ∇. The effect of this is that evaluating the gradient of a function in a finite element field just requires the evaluation of the gradient of the Vandermonde matrix.

Exercise 2.8 *Extend* `vandermonde_matrix()` *so that setting* `grad` *to* `True` *produces a rank 3 generalised Vandermonde tensor whose indices represent points, monomial basis function, and gradient component respectively. That is:*

$$\nabla V_{ijk} = \frac{\partial V_j(X_i)}{\partial x_k} \tag{2.14}$$

In other words, each entry of V is replaced by a vector of the gradient of that polynomial term. For example, the entry x^2y^3 would be replaced by the vector $[2xy^3, 3x^2y^2]$.

The `test/test_06_vandermonde_matrix_grad.py` *file has tests of this extension. You should also ensure that you still pass* `test/test_03_vandermonde_matrix.py`*.*

Hint

The `transpose()` method of numpy arrays enables generalised transposes swapping any dimensions.

Hint

At least one of the natural ways of implementing this function results in a whole load of `nan` values in the generalised Vandermonde matrix. In this case, you might find `numpy.nan_to_num()` useful.

Exercise 2.9 *Extend* `tabulate()` *to pass the* `grad` *argument through to* `vandermonde_matrix()`*. Then generalise the matrix product in* `tabulate()` *so that the result of this function (when* `grad` *is true) is a rank 3 tensor:*

$$\mathrm{T}_{ijk} = \nabla(\phi_j(X_i)) \cdot \mathbf{e}_k \tag{2.15}$$

where $\mathbf{e}_0 \ldots \mathbf{e}_{\dim -1}$ is the coordinate basis on the reference cell.

The `test/test_07_tabulate_grad.py` *script tests this extension. Once again, make sure you still pass* `test/test_05_tabulate.py`

Hint

The `numpy.einsum()` function implements generalised tensor contractions using Einstein summation notation. For example:

```
A = numpy.einsum("ijk,jl->ilk", T, C)
```

is equivalent to $A_{ilk} = \sum_j T_{ijk}C_{jl}$.

2.9 Interpolating functions to the finite element nodes

Recall once again that a function can be represented on a single finite element as:

$$f = \sum_i f_i \phi_i \tag{2.16}$$

Since $\{\phi_i\}$ is a nodal basis, it follows immediately that:

$$f_i = \phi_i^*(f) \tag{2.17}$$

where ϕ_i^* is the node associated with the basis function ϕ_i. Since we are only interested in nodes which are the point evaluation of their function input, we know that:

$$f_i = f(X_i) \tag{2.18}$$

where X_i is the point associated with the i-th node.

Exercise 2.10 *Implement* `interpolate()`.

Once you have done this, you can use the script provided to plot functions of your choice interpolated onto any of the finite elements you can make:

```
plot_interpolate_lagrange "sin(2*pi*x[0])" 2 5
```

Hint

You can find help on the arguments to this function with:

```
plot_interpolate_lagrange -h
```

CHAPTER

THREE

MESHES

When employing the finite element method, we represent the domain on which we wish to solve our PDE as a mesh. In order to work with meshes, we need to have a somewhat more formal mathematical notion of a mesh. The mesh concepts we will employ here are loosely based on those in [Log09], and are typical of mesh representations for the finite element method.

3.1 Mesh entities

Like a cell, a mesh is composed of *topological entities*, such as vertices, edges, polygons and polyhedra. The distinction is that a mesh is made of potentially many cells, and a commensurate number of lower-dimensional entities.

Definition 3.1 *The* (topological) dimension *of a mesh is the largest dimension among all of the topological entities in a mesh.*

In this course we will not consider meshes of manifolds immersed in higher dimensional spaces (for example the surface of a sphere immersed in $\mathbb{R}^3$) so the topological dimension of the mesh will always match the geometric dimension of space in which we are working, so we will simply refer to the *dimension* of the mesh.

The numbering of mesh entities is similar to that of cell entities, except that the indices range over all of the entities of that dimension in the mesh. For example, entity $(0, 10)$ is vertex number 10, and entity $(1, 10)$ is edge 10. Fig. 3.1 shows an example mesh with the topological entities labelled.

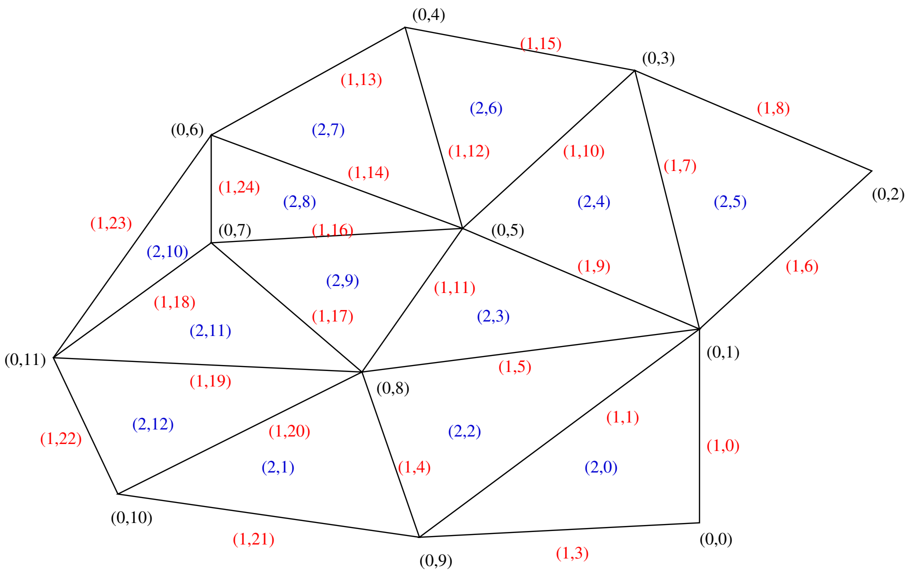

Fig. 3.1: A triangular mesh showing labelled topological entities: vertices (black), edges (red), and cells (blue).

3.2 Adjacency

In order to implement the finite element method, we need to integrate functions over cells, which means knowing which basis functions are nonzero in a given cell. For the function spaces used in the finite element method, these basis functions will be the ones whose nodes lie on the topological entities adjacent to the cell. That is, the vertices, edges and (in 3D) the faces making up the cell, as well as the cell itself. One of the roles of the mesh is therefore to provide a lookup facility for the lower-dimensional mesh entities adjacent to a given cell.

Definition 3.2 *Given a mesh M, then for each $\dim(M) \geq d_1 > d_2 \geq 0$ the* adjacency *function* $\mathrm{Adj}_{d_1,d_2} : \mathbb{N} \to \mathbb{N}^k$ *is the function such that:*

$$\mathrm{Adj}_{d_1,d_2}(i) = (i_0, \ldots i_k)$$

where (d_1, i) is a topological entity and $(d_2, i_0), \ldots, (d_2, i_k)$ are the adjacent d_2-dimensional topological entities numbered in the corresponding reference cell order. If every cell in the mesh has the same topology then k will be fixed for each (d_1, d_2) pair. The correspondence between the orientation of the entity (d_1, i) and the reference cell of dimension d_1 is established by specifying that the vertices are numbered in ascending order[1]*. That is, for any entity (d_1, i):*

$$(i_0, \ldots i_k) = \mathrm{Adj}_{d_1,0}(i) \quad \Longrightarrow \quad i_0 < \ldots < i_k$$

A consequence of this convention is that the global orientation of all the entities making up a cell also matches their local orientation.

Example 3.3 *In the mesh shown in Fig. 3.1 we have:*

$$\mathrm{Adj}_{2,0}(3) = (1, 5, 8).$$

In other words, vertices 1, 5 and 8 are adjacent to cell 3. Similarly:

$$\mathrm{Adj}_{2,1}(3) = (11, 5, 9).$$

Edges 11, 5, and 9 are local edges 0, 1, and 2 of cell 3.

3.3 Mesh geometry

The features of meshes we have so far considered are purely topological: they deal with the adjacency relationships between topological entities, but do not describe the locations of those entities in space. Provided we restrict our attention to meshes in which the element edges are straight (ie not curved), we can represent the geometry of the mesh by simply recording the coordinates of the vertices. The positions of the higher dimensional entities then just interpolate the vertices of which they are composed. We will later observe that this is equivalent to representing the geometry in a vector-valued piecewise linear finite element space.

3.4 A mesh implementation in Python

The `Mesh` class provides an implementation of mesh objects in 1 and 2 dimensions. Given the list of vertices making up each cell, it constructs the rest of the adjacency function. It also records the coordinates of the vertices.

The `UnitSquareMesh` class creates a `Mesh` object corresponding to a regular triangular mesh of a unit square. Similarly, the `UnitIntervalMesh` class performs the corresponding (rather trivial) function for a unit one dimensional mesh.

You can observe the numbering of mesh entities in these meshes using the `plot_mesh` script. Run:

[1] The numbering convention adopted here is very convenient, but only works for meshes composed of simplices (vertices, intervals, triangles and tetrahedra). A more complex convention would be required to support quadrilateral meshes.

```
plot_mesh -h
```

for usage instructions.

CHAPTER

FOUR

FUNCTION SPACES: ASSOCIATING DATA WITH MESHES

A finite element space over a mesh is constructed by associating a finite element with each cell of the mesh. We will refer to the basis functions of this finite element space as *global* basis functions, while those of the finite element itself we will refer to as *local* basis functions. We can establish the relationship between the finite element and each cell of the mesh by associating the nodes (and therefore the local basis functions) of the finite element with the topological entities of the mesh. This is a two stage process. First, we associate the nodes of the finite element with the local topological entities of the reference cell. This is often referred to as *local numbering*. Then we associate the correct number of degrees of freedom (i.e. number of basis functions) with each global mesh entity. This is the *global numbering*.

4.1 Local numbering and continuity

Which nodes should be associated with which topological entities? The answer to this question depends on the degree of continuity required between adjacent cells. The nodes associated with topological entites on the boundaries of cells (the vertices in one dimension, the vertices and edges in two dimensions, and the vertices, edges and faces in three dimensions) are shared between cells. The basis functions associated with nodes on the cell boundary will therefore be continuous between the cells which share that boundary.

For the Lagrange element family, we require global C_0 continuity. This implies that the basis functions are continuous everywhere. This has the following implications for the association of basis functions with local topological entites:

vertices

At the function vertices we can achieve continuity by requiring that there be a node associated with each mesh vertex. The basis function associated with that node will therefore be continuous. Since we have a nodal basis, all the other basis functions will vanish at the vertex so the global space will be continuous at this point.

edges

Where the finite element space has at least 2 dimensions we need to ensure continuity along edges. The restriction of a degree p polynomial over a d-dimensional cell to an edge of that cell will be a one dimensional degree p polynomial. To fully specify this polynomial along an edge requires $p+1$ nodes. However there will already be two nodes associated with the vertices of the edge, so $p-1$ additional nodes will be associated with the edge.

faces

For three-dimensional (tetrahedral) elements, the basis functions must also be continuous across faces. This requires that sufficient nodes lie on the face to fully specify a two dimensional degree p polynomial. However the vertices and edges of the face already have nodes associated with them, so the number of nodes required to be associated with the face itself is actually the number required to represent a degree $p-2$ polynomial in two dimensions given by the combination $\binom{p-1}{2}$.

This pattern holds more generally: for a C_0 function space, the number of nodes which must be associated with a local topological entity of dimension d is $\binom{p-1}{d}$.

Fig. 4.1 illustrates the association of nodes with reference entities for Lagrange elements on triangles.

Fig. 4.1: Association of nodes with reference entities for the degree 1, 2, and 3 equispaced Lagrange elements on triangles. Black nodes are associated with vertices, red nodes with edges and blue nodes with the cell (face). This is the same figure as Fig. 2.1.

4.2 Implementing local numbering

Local numbering can be implemented by adding an additional data structure to the `FiniteElement` class. For each local entity this must record the local nodes associated with that entity. This can be achieved using a dictionary of dictionaries structure. For example employing the local numbering of nodes employed in Fig. 2.1, the `entity_node` dictionary for the degree three equispaced Lagrange element on a triangle is given by:

```
entity_node = {0: {0: [0],
                   1: [1],
                   2: [2]},
               1: {0: [3, 4],
                   1: [5, 6],
                   2: [7, 8]},
               2: {0: [9]}}
```

Note that the order of the nodes in each list is important: it must always consistently reflect the orientation of the relevant entity in order that all the cells which share that entity consistently interpret the nodes. In this case this has been achieved by listing the nodes in order given by the direction of the orientation of each edge.

Exercise 4.1 *Extend the* `__init__()` *method of* `LagrangeElement` *so that it passes the correct* `entity_node` *dictionary to the* `FiniteElement` *it creates.*

The `test/test_08_entity_nodes.py` *script tests this functionality.*

4.3 Global numbering

Given a mesh and a finite element, the global numbering task is to uniquely associate the appropriate number of global node numbers with each global entity. One such numbering[1] is to allocate global numbers in ascending entity dimension order, and within each dimension in order of the index of each global topological entity. The formula for the first global node associated with entity (d, i) is then:

$$G(d,i) = \left(\sum_{\delta<d} N_\delta E_\delta\right) + iN_d \tag{4.1}$$

where N_d is the number of nodes which this finite element associates with a single entity of dimension d, and E_d is the number of dimension d entities in the mesh. The full list of nodes associated with entity (d, i) is therefore:

$$[G(d,i), \ldots, G(d,i) + N_d - 1] \tag{4.2}$$

[1] Many correct global numberings are possible, that presented here is simple and correct, but not optimal from the perspective of the memory layout of the resulting data.

4.4 The cell-node map

The primary use to which we wish to put the finite element spaces we are constructing is, naturally, the solution of finite element problems. The principle operation we will therefore need to support is integration over the mesh of mathematical expressions involving functions in finite element spaces. This will be accomplished by integrating over each cell in turn, and then summing over all cells. This means that a key operation we will need is to find the nodes associated with a given cell.

It is usual in finite element software to explicitly store the map from cells to adjacent nodes as a two-dimensional array with one row corresponding to each cell, and with columns corresponding to the local node numbers. The entries in this map will have the following values:

$$M[c, e(\delta, \epsilon)] = [G(\delta, i), \dots, G(\delta, i) + N_\delta - 1] \qquad \forall 0 \leq \delta \leq \dim(c), \forall 0 \leq \epsilon < \hat{E}_\delta \tag{4.3}$$

where:

$$i = \mathrm{Adj}_{\dim(c),\delta}[c, \epsilon], \tag{4.4}$$

$e(\delta, \epsilon)$ is the local entity-node list for this finite element for the (δ, ϵ) local entity, Adj has the meaning given under *Python implementations of reference elements*, $\hat{E}_\delta$ is the number of dimension δ entities in each cell, and G and N have the meanings given above. This algorithm requires a trivial extension to adjacency:

$$\mathrm{Adj}_{\dim(c),\dim(c)}[c, 0] = c \tag{4.5}$$

Hint

In (4.3), notice that for each value of δ and ϵ, $e(\delta, \epsilon)$ is a vector of indices, so the equation sets the value of zero, one, or more defined entries in row c of M for each δ and ϵ.

4.5 Implementing function spaces in Python

As noted above, a finite element space associates a mesh and a finite element, and contains (in some form) a global numbering of the nodes.

Exercise 4.2 *Implement the* `__init__()` *method of* `fe_utils.function_spaces.FunctionSpace`. *The key operation is to set* `cell_nodes` *using* (4.3).

You can plot the numbering you have created with the `plot_function_space_nodes` *script. As usual, run the script passing the* `-h` *option to discover the required arguments.*

Hint

Many of the terms in (4.3) are implemented in the objects in `fe_utils`. For example:

- $\mathrm{Adj}_{\dim(c),\delta}$ is implemented by the `adjacency()` method of the `Mesh`.
- You have $e(\delta, \epsilon)$ as `entity_nodes`. Note that in this case you need separate square brackets for each index:

```
element.entity_nodes[delta][epsilon]
```

Hint

`cell_nodes` needs to be integer-valued. If you choose to use `numpy.zeros()` to create a matrix which you then populate with values, you need to explicitly specify that you want a matrix of integers. This can be

achieved by passing the `dtype` argument to `numpy.zeros()`. For example `numpy.zeros((nrows, ncols), dtype=int)`.

CHAPTER

FIVE

FUNCTIONS IN FINITE ELEMENT SPACES

Recall that the general form of a function in a finite element space is:

$$f(x) = \sum_i f_i \phi_i(x) \tag{5.1}$$

Where the $\phi_i(x)$ are now the global basis functions achieved by stitching together the local basis functions defined by the *finite element*.

5.1 A python implementation of functions in finite element spaces

The `Function` class provides a simple implementation of function storage. The input is a `FunctionSpace` which defines the mesh and finite element to be employed, to which the `Function` adds an array of degree of freedom values, one for each node in the `FunctionSpace`.

5.2 Interpolating values into finite element spaces

Suppose we have a function $g(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ which we wish to approximate as a function $f(x)$ in some finite element space V. In other words, we want to find the f_i such that:

$$\sum_i f_i \phi_i(x) \approx g(x) \tag{5.2}$$

The simplest way to do this is to *interpolate* $g(x)$ onto V. In other words, we evaluate:

$$f_i = n_i(g(x)) \tag{5.3}$$

where n_i is the node associated with ϕ_i as considered over the entire mesh (*globally*). Since we are only concerned with point evaluation nodes, this is equivalent to:

$$f_i = g(x_i) \tag{5.4}$$

where x_i is the coordinate vector of the point defining the node n_i. This looks straightforward, however the x_i are the *global* node points, and so far we have only defined the node points in *local* coordinates on the reference element.

5.2.1 Changing coordinates between reference and physical space

We'll refer to coordinates on the global mesh as being in *physical space* while those on the reference element are in *local space*. We'll use case to distinguish local and global objects, so local coordinates will be written as X and global coordinates as x. The key observation is that within each cell, the global coordinates are the linear interpolation of the global coordinate values at the cell vertices. In other words, if $\{\Psi_j\}$ is the local basis for the **linear** lagrange elements on the reference cell and $\hat{x}_j$ are the corresponding global vertex locations on a cell c then:

$$x = \sum_j \hat{x}_j \Psi_j(X) \quad \forall x \in c. \tag{5.5}$$

Remember that we know the location of the nodes in local coordinates, and we have the `tabulate()` method to evaluate all the basis functions of an element at a known set of points. So if we write:

$$A_{i,j} = \Psi_j(X_i) \tag{5.6}$$

where $\{X_i\}$ are the local node points of our finite element, then the corresponding global node points $\{x_i\}$ are given by

$$x = A \cdot \hat{x} \tag{5.7}$$

where $\hat{x}$ is the $(\dim +1, \dim)$ array whose rows are the current element vertex coordinates. Specifically, x is the $(\mathrm{nodes}, \dim)$ array whose rows are the global coordinates of the nodes in the current element. We can then apply $g()$ to each row of x in turn and record the result as the `Function` value for that node.

Hint

The observant reader will notice that this algorithm is inefficient because the function values at nodes on the boundaries of elements are evaluated more than once. This can be avoided with a little tedious bookkeeping but we will not concern ourselves with that here.

5.2.2 Looking up cell coordinates and values

In the previous section we used the vertex coordinates of a cell to find the node coordinates, and then we calculated `Function` values at those points. The coordinates are stored in a single long list associated with the `Mesh`, and the `Function` contains a single long list of values. We need to use *indirect addressing* to access these values. This is best illustrated using some Python code.

Suppose `f` is a `Function`. For brevity, we write `fs = f.function_space`, the `FunctionSpace` associated with `f`. Now, we first need a linear element and a corresponding `FunctionSpace`:

```
cg1 = fe_utils.LagrangeElement(fs.mesh.cell, 1)
cg1fs = fe_utils.FunctionSpace(fs.mesh, cg1)
```

Then the vertex indices of cell number `c` in the correct order for the linear Lagrange element are:

```
cg1fs.cell_nodes[c, :]
```

and therefore the set of coordinate vectors for the vertices of element `c` are:

```
fs.mesh.vertex_coords[cg1fs.cell_nodes[c, :], :]
```

That is, the `cg1fs.cell_nodes` array is used to look up the right vertex coordinates. By a similar process we can access the values associated with the nodes of element `c`:

```
f.values[fs.cell_nodes[c, :]]
```

5.2.3 A Python implementation of interpolation

Putting together the change of coordinates with the right indirect addressing, we can provide the `Function` class with a `interpolate()` method which interpolates a user-provided function onto the `Function`.

Exercise 5.1 *Read and understand the* `interpolate()` *method. Use* `plot_sin_function` *to investigate interpolating different functions onto finite element spaces at differering resolutions and polynomial degrees.*

Hint

There is no implementation work associated with this exercise, but the programming constructs used in `interpolate()` will be needed when you implement integration.

5.3 Integration

We now come to one of the fundamental operations in the finite element method: integrating a `Function` over the domain. The full finite element method actually requires the integration of expressions of unknown test and trial functions, but we will start with the more straightforward case of integrating a single, known, `Function` over a domain Ω:

$$\int_\Omega f \mathrm{d}x \quad f \in V \tag{5.8}$$

where $\mathrm{d}x$ should be understood as being the volume measure with the correct dimension for the domain and V is some finite element space over Ω. We can express this integral as a sum of integrals over individual cells:

$$\int_\Omega f \mathrm{d}x = \sum_{c\in\Omega} \int_c f \mathrm{d}x. \tag{5.9}$$

So we have in fact reduced the integration problem to the problem of integrating f over each cell. In *a previous part* of the module we implemented quadrature rules which enable us to integrate over specified reference cells. If we can express the integral over some arbitrary cell c as an integral over a reference cell c_0 then we are done. In fact this simply requires us to employ the change of variables formula for integration:

$$\int_c f(x)\mathrm{d}x = \int_{c_0} f(X)|J|\mathrm{d}X \tag{5.10}$$

where $|J|$ is the absolute value of the determinant of the Jacobian matrix. J is given by:

$$J_{\alpha\beta} = \frac{\partial x_\alpha}{\partial X_\beta}. \tag{5.11}$$

Hint

We will generally adopt the convention of using Greek letters to indicate indices in spatial dimensions, while we will use Roman letters in the sequence $i, j, \ldots$ for basis function indices. We will continue to use q for the index over the quadrature points.

Evaluating (5.11) depends on having an expression for x in terms of X. Fortunately, (5.5) is exactly this expression, and applying the usual rule for differentiating functions in finite element spaces produces:

$$J_{\alpha\beta} = \sum_j (\tilde{x}_j)_\alpha \nabla_\beta \Psi_j(X) \tag{5.12}$$

where $\{\Psi_j\}$ is once again the degree 1 Lagrange basis and $\{\tilde{x}_j\}$ are the coordinates of the corresponding vertices of cell c. The presence of X in (5.12) implies that the Jacobian varies spatially across the reference cell. However

since $\{\Psi_j\}$ is the degree 1 Lagrange basis, the gradients of the basis functions are constant over the cell and so it does not matter at which point in the cell the Jacobian is evaluated. For example we might choose to evaluate the Jacobian at the cell origin $X = 0$.

Hint

When using simplices with curved sides, and on all but the simplest quadrilateral or hexahedral meshes, the change of coordinates will not be affine. In that case, to preserve full accuracy it will be necessary to compute the Jacobian at every quadrature point. However, non-affine coordinate transforms are beyond the scope of this course.

5.3.1 Expressing the function in the finite element basis

Let $\{\Phi_i(X)\}$ be a **local** basis for V on the reference element c_0. Then our integral becomes:

$$\int_c f(x)\mathrm{d}x = \int_{c_0} \sum_i F(M(c,i))\,\Phi_i(X)\,|J|\,\mathrm{d}X \tag{5.13}$$

where F is the vector of global coefficient values of f, and M is *the cell node map*.

5.3.2 Numerical quadrature

The actual evaluation of the integral will employ the quadrature rules we discussed in *a previous section*. Let $\{X_q\}, \{w_q\}$ be a quadrature rule of sufficient degree of precision that the quadrature is exact. Then:

$$\int_c f(x)\mathrm{d}x = \sum_q \sum_i F(M(c,i))\,\Phi_i(X_q)\,|J|\,w_q \tag{5.14}$$

5.3.3 Implementing integration

Exercise 5.2 *Use* (5.12) *to implement the* `jacobian()` *method of* `Mesh`. `test/test_09_jacobian.py` *is available for you to test your results.*

Hint

The $\nabla_\beta \Psi_j(X)$ factor in (5.12) is the same for every cell in the mesh. You could make your implementation more efficient by precalculating this term in the `__init__()` method of `Mesh`.

Exercise 5.3 *Use* (5.9) *and* (5.14) *to implement* `integrate()`. `test/test_10_integrate_function.py` *may be used to test your implementation.*

Hint

Your method will need to:

1. Construct a suitable `QuadratureRule`.
2. `tabulate()` the basis functions at each quadrature point.
3. Visit each cell in turn.
4. Construct the `jacobian()` for that cell and take the absolute value of its determinant (`numpy.absolute` and `numpy.linalg.det()` will be useful here).
5. Sum all of the arrays you have constructed over the correct indices to a contribution to the integral (`numpy.einsum()` may be useful for this).

Hint

You might choose to read ahead before implementing `integrate()`, since the `errornorm()` function is very similar and may provide a useful template for your work.

CHAPTER

SIX

ASSEMBLING AND SOLVING FINITE ELEMENT PROBLEMS

Having constructed functions in finite element spaces and integrated them over the domain, we now have the tools in place to actually assemble and solve a simple finite element problem. To avoid having to explicitly deal with boundary conditions, we choose in the first instance to solve a Helmholtz problem[1], find u in some finite element space V such that:

$$\begin{aligned} -\nabla^2 u + u &= f \\ \nabla u \cdot \mathbf{n} &= 0 \text{ on } \Gamma \end{aligned} \tag{6.1}$$

where Γ is the domain boundary and n is the outward pointing normal to that boundary. f is a known function which, for simplicity, we will assume lies in V. Next, we form the weak form of this equation by multiplying by a test function in V and integrating over the domain. We integrate the Laplacian term by parts. The problem becomes, find $u \in V$ such that:

$$\int_\Omega \nabla v \cdot \nabla u + vu \,\mathrm{d}x - \underbrace{\int_\Gamma v \nabla u \cdot \mathbf{n} \,\mathrm{d}s}_{=0} = \int_\Omega vf \,\mathrm{d}x \qquad \forall v \in V \tag{6.2}$$

If we write $\{\phi_i\}_{i=0}^{n-1}$ for our basis for V, and recall that it is sufficient to ensure that (6.2) is satisfied for each function in the basis then the problem is now, find coefficients u_i such that:

$$\int_\Omega \sum_j \left(\nabla \phi_i \cdot \nabla(u_j \phi_j) + \phi_i u_j \phi_j\right) \,\mathrm{d}x = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \qquad \forall\, 0 \leq i < n \tag{6.3}$$

Note that since (6.2) is linear in $v = \sum_i v_i \phi_i$ we are able to drop the coefficients v_i. Since the left hand side is linear in the scalar coefficients u_j, we can move them out of the integral:

$$\sum_j \left(\int_\Omega \nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j \,\mathrm{d}x \, u_j \right) = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \qquad \forall\, 0 \leq i < n \tag{6.4}$$

We can write this as a matrix equation:

$$\mathrm{A}\mathbf{u} = \mathbf{f} \tag{6.5}$$

where:

$$\mathrm{A}_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j \,\mathrm{d}x \tag{6.6}$$

$$\mathbf{u}_j = u_j \tag{6.7}$$

$$\mathbf{f}_i = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \tag{6.8}$$

[1] Strictly speaking this is the positive definite Helmholtz problem. Changing the sign on u produces the indefinite Helmholtz problem, which is significantly harder to solve.

6.1 Assembling the right hand side

The assembly of these integrals exploits the same decomposition property we exploited previously to integrate functions in finite element spaces. For example, (6.8) can be rewritten as:

$$\mathbf{f}_i = \sum_c \int_c \phi_i \sum_k f_k \phi_k \, \mathrm{d}x \tag{6.9}$$

This has a practical impact once we realise that only a few basis functions are non-zero in each element. This enables us to write an efficient algorithm for right hand side assembly. Assume that at the start of our algorithm:

$$\mathbf{f}_i = 0. \tag{6.10}$$

Now for each cell c, we execute:

$$\mathbf{f}_{M(c,\hat{\imath})} \stackrel{+}{=} \int_c \Phi_{\hat{\imath}} \left(\sum_{\hat{k}} f_{M(c,\hat{k})} \, \Phi_{\hat{k}} \right) |J| \, \mathrm{d}X \qquad \forall 0 \leq \hat{\imath} < N \tag{6.11}$$

Where M is the cell-node map for the finite element space V, N is the number of nodes per element in V, and $\{\Phi_{\hat{\imath}}\}_{\hat{\imath}=0}^{N-1}$ are the local basis functions. In other words, we visit each cell and conduct the integral for each local basis function, and add that integral to the total for the corresponding global basis function.

By choosing a suitable quadrature rule, $\{X_q\}, \{w_q\}$, we can write this as:

$$\mathbf{f}_{M(c,\hat{\imath})} \stackrel{+}{=} \left(\sum_q \Phi(X_q)_{\hat{\imath}} \left(\sum_{\hat{k}} f_{M(c,\hat{k})} \, \Phi(X_q)_{\hat{k}} \right) w_q \right) |J| \qquad \forall 0 \leq \hat{\imath} < N, \, \forall c \tag{6.12}$$

6.2 Assembling the left hand side matrix

The left hand side matrix follows a similar pattern, however there are two new complications. First, we have two unbound indices (i and j), and second, the integral involves derivatives. We will address the question of derivatives first.

6.2.1 Pulling gradients back to the reference element

On element c, there is a straightforward relationship between the local and global bases:

$$\phi_{M(c,i)}(x) = \Phi_i(X) \tag{6.13}$$

We can also, as we showed in *Changing coordinates between reference and physical space*, express the global coordinate x in terms of the local coordinate X.

What about $\nabla \phi$? We can write the gradient operator in component form and apply (6.13):

$$\frac{\partial \phi_{M(c,i)}(x)}{\partial x_\alpha} = \frac{\partial \Phi_i(X)}{\partial x_\alpha} \quad \forall \, 0 \leq \alpha < \dim \tag{6.14}$$

However, the expression on the right involves the gradient of a local basis function with respect to the global coordinate variable x. We employ the chain rule to express this gradient with respect to the local coordinates, X:

$$\frac{\partial \phi_{M(c,i)}(x)}{\partial x_\alpha} = \sum_{\beta=0}^{\dim - 1} \frac{\partial X_\beta}{\partial x_\alpha} \frac{\partial \Phi_i(X)}{\partial X_\beta} \quad \forall \, 0 \leq \alpha < \dim \tag{6.15}$$

Using the *definition of the Jacobian*, and using ∇_x and ∇_X to indicate the global and local gradient operators respectively, we can equivalently write this expression as:

$$\nabla_x \phi_{M(c,i)}(x) = J^{-\mathrm{T}} \nabla_X \Phi_i(X) \tag{6.16}$$

where $J^{-\mathrm{T}} = (J^{-1})^{\mathrm{T}}$ is the transpose of the inverse of the cell Jacobian matrix.

6.2.2 The assembly algorithm

We can start by transforming (6.6) to local coordinates (referred to as *pulling back*) and considering it in the algorithmic form used for the right hand side in (6.9) to (6.12):

$$
\begin{gathered}
\mathrm{A}_{ij} = 0. \\
\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{\pm}{=} \int_c \left(\left(J^{-T}\nabla_X \Phi_{\hat{i}}\right) \cdot \left(J^{-T}\nabla_X \Phi_{\hat{j}}\right) + \Phi_{\hat{i}}\Phi_{\hat{j}} \right) |J| \,\mathrm{d}X \quad \forall 0 \leq \hat{i}, \hat{j} < N, \forall c
\end{gathered} \tag{6.17}
$$

We now employ a suitable quadrature rule, $\{X_q\}, \{w_q\}$, to calculate the integral:

$$
\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{\pm}{=} \sum_q \left(\left(J^{-T}\nabla_X \Phi_{\hat{i}}(X_q)\right) \cdot \left(J^{-T}\nabla_X \Phi_{\hat{j}}(X_q)\right) + \Phi_{\hat{i}}(X_q)\Phi_{\hat{j}}(X_q) \right) |J| \, w_q \quad \forall 0 \leq \hat{i}, \hat{j} < N, \forall c \tag{6.18}
$$

Some readers may find this easier to read using index notation over the geometric dimensions:

$$
\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{\pm}{=} \sum_q \left(\sum_{\alpha\beta\gamma} J^{-1}_{\beta\alpha} \left(\nabla_X \Phi_{\hat{i}}(X_q)\right)_\beta \, J^{-1}_{\gamma\alpha} \left(\nabla_X \Phi_{\hat{j}}(X_q)\right)_\gamma + \Phi_{\hat{i}}(X_q)\Phi_{\hat{j}}(X_q) \right) |J| \, w_q \quad \forall 0 \leq \hat{i}, \hat{j} < N, \forall c \tag{6.19}
$$

6.2.3 A note on matrix insertion

For each cell c, the right hand sides of equations (6.18) and (6.19) have two free indices, $\hat{i}$ and $\hat{j}$. The equation therefore assembles a local $N \times N$ matrix corresponding to one integral for each test function, trial function pair on the current element. This is then added to the global matrix at the row and column pairs given by the cell node map $M(c, \hat{i})$ and $M(c, \hat{j})$.

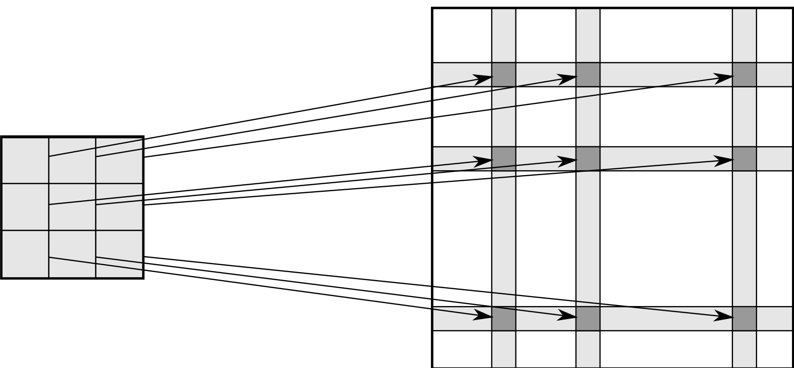

Fig. 6.1: Computing integrals for each local test and trial function produces a local dense (in this case, 3×3) matrix. The entries in this matrix are added to the corresponding global row and column positions in the global matrix.

Hint

One might naïvely expect that if `nodes` is the vector of global node numbers for the current cell, `m` is the matrix of local integral values and `A` is the global matrix, then the Python code might look like:

```
A[nodes, nodes] += m # DON'T DO THIS!
```

Unfortunately, `numpy` interprets this as an instruction to insert a vector into the diagonal of `A`, and will complain that the two-dimensional right hand side does not match the one-dimensional left hand side. Instead, one has to employ the `numpy.ix_()` function:

```
A[np.ix_(nodes, nodes)] += m # DO THIS!
```

No such problem exists for adding values into the global right hand side vector. If `l` is the global right hand side vector and `v` is the vector of local right hand integrals, then the following will work just fine:

```
l[nodes] += v
```

6.2.4 Sparse matrices

Each row of the global matrix corresponds to a single global basis function. The number of non-zeros in this row is equal to the number of other basis functions which are non-zero in the elements where the original basis function is non-zero. The maximum number of non-zeros on a row may vary from a handful for a low degree finite element to a few hundred for a fairly high degree element. The important point is that it is essentially independent of the size of the mesh. This means that as the number of cells in the mesh increases, the proportion of the matrix entries on each row which have the value zero increases.

For example, a degree 4 Lagrange finite element space defined on 64×64 unit square triangular mesh has about 66000 nodes. The full global matrix therefore has more that 4 billion entries and, at 8 bytes per matrix entry, will consume around 35 gigabytes of memory! However, there are actually only around 23 nonzeros per row, so more than 99.9% of the entries in the matrix are zeroes.

Instead of storing the complete matrix, sparse matrix formats store only those entries in the matrix which are nonzero. They also have to store some metadata to describe where in the matrix the non-zero entries are stored. There are various different sparse matrix formats available, which make different trade-offs between memory usage, insertion speed, and the speed of different matrix operations. However, if we make the (conservative) assumption that a sparse matrix takes 16 bytes to store each nonzero value, instead of 8 bytes, then we discover that in the example above, we would use less than 25 megabytes to store the matrix. The time taken to solving the matrix system will also be vastly reduced since operations on zeros are avoided.

Hint

The `scipy.sparse` package provides convenient interfaces which enable Python code to employ a variety of sparse matrix formats using essentially identical operations to the dense matrix case. The skeleton code already contains commands to construct empty sparse matrices and to solve the resulting linear system. You may, if you wish, experiment with choosing other sparse formats from `scipy.sparse`, but it is very strongly suggested that you do **not** switch to a dense numpy array; unless, that is, you particularly enjoy running out of memory on your computer!

6.3 The method of manufactured solutions

When the finite element method is employed to solve Helmholtz problems arising in science and engineering, the value forcing function f will come from the application data. However for the purpose of testing numerical methods and software, it is exceptionally useful to be able to find values of f such that an analytic solution to the partial differential equation is known. It turns out that there is a straightforward algorithm for this process. This algorithm is known as the *method of manufactured solutions*. It has but two steps:

1. Choose a function $\tilde{u}$ which satisfies the boundary conditions of the PDE.
2. Substitute $\tilde{u}$ into the left hand side of (6.1). Set f equal to the result of this calculation, and now $\tilde{u}$ is a solution to (6.1).

To illustrate this algorithm, suppose we wish to construct f such that:

$$\tilde{u} = \cos(4\pi x_0) x_1^2 (1 - x_1)^2 \tag{6.20}$$

is a solution to (6.1) defined on a domain Γ bounded by the unit square (a square with vertices at the points $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$). It is simple to verify that $\tilde{u}$ satisfies the boundary conditions. We then note that:

$$-\nabla^2\tilde{u} + \tilde{u} = \left((16\pi^2+1)(x_1-1)^2x_1^2 - 12x_1^2 + 12x_1 - 2\right)\cos(4\pi x_0) \tag{6.21}$$

If we choose:

$$f = \left((16\pi^2+1)(x_1-1)^2x_1^2 - 12x_1^2 + 12x_1 - 2\right)\cos(4\pi x_0) \tag{6.22}$$

then $\tilde{u}$ is a solution to (6.1).

6.4 Errors and convergence

6.4.1 The L^2 error

When studying finite element methods we are freqently concerned with convergence in the L^2 norm. That is to say, if V and W are finite element spaces defined over the same mesh, and $f \in V, g \in W$ then we need to calculate:

$$\sqrt{\int_\Omega (f-g)^2 \mathrm{d}x} = \sqrt{\sum_c \int_c \left(\left(\sum_i f_{M_V(c,i)}\Phi_i\right) - \left(\sum_j g_{M_W(c,j)}\Psi_j\right)\right)^2 |J|\mathrm{d}X} \tag{6.23}$$

where M_V is the cell-node map for the space V and M_W is the cell-node map for the space W. Likewise $\{\Phi_i\}$ is the local basis for V and $\{\Psi_j\}$ is the local basis for W.

A complete quadrature rule for this integral will, due to the square in the integrand, require a degree of precision equal to twice the greater of the polynomial degrees of V and W.

6.4.2 Numerically estimating convergence rates

Using the approximation results from the theory part of the course, we know that the error term in the finite element solution of the Helmholtz equation is expected to have the form $\mathcal{O}(h^{p+1})$ where h is the mesh spacing and p is the polynomial degree of the finite element space employed. That is to say if $\tilde{u}$ is the exact solution to our PDE and u_h is the solution to our finite element problem, then for sufficiently small h:

$$\|u_h - \tilde{u}\|_{L^2} < ch^{p+1} \tag{6.24}$$

for some $c > 0$ not dependent on h. Indeed, for sufficiently small h, there is a c such that we can write:

$$\|u_h - \tilde{u}\|_{L^2} \approx ch^{p+1} \tag{6.25}$$

Suppose we solve the finite element problem for two different (fine) mesh spacings, h_1 and h_2. Then we have:

$$\begin{aligned} \|u_{h_1} - \tilde{u}\|_{L^2} &\approx ch_1^{p+1} \\ \|u_{h_2} - \tilde{u}\|_{L^2} &\approx ch_2^{p+1} \end{aligned} \tag{6.26}$$

or equivalently:

$$\frac{\|u_{h_1} - \tilde{u}\|_{L^2}}{\|u_{h_2} - \tilde{u}\|_{L^2}} \approx \left(\frac{h_1}{h_2}\right)^{p+1} \tag{6.27}$$

By taking logarithms and rearranging this equation, we can produce a formula which, given the analytic solution and two numerical solutions, produces an estimate of the rate of convergence:

$$q = \frac{\ln\left(\dfrac{\|u_{h_1} - \tilde{u}\|_{L^2}}{\|u_{h_2} - \tilde{u}\|_{L^2}}\right)}{\ln\left(\dfrac{h_1}{h_2}\right)} \tag{6.28}$$

6.5 Implementing finite element problems

Exercise 6.1 `fe_utils/solvers/helmholtz.py` *contains a partial implementation of the finite element method to solve* (6.2) *with* f *chosen as in* (6.22). *Your task is to implement the* `assemble()` *function using* (6.12), *and* (6.18) *or* (6.19). *The comments in the* `assemble()` *function provide some guidance as to the steps involved. You may also wish to consult the* `errornorm()` *function as a guide to the structure of the code required.*

Run:

```
python fe_utils/solvers/helmholtz.py --help
```

for guidance on using the script to view the solution, the analytic solution and the error in your solution. In addition, `test/test_11_helmholtz_convergence.py` *contains tests that the helmholtz solver converges at the correct rate for degree 1, 2 and 3 polynomials.*

Warning

`test/test_12_helmholtz_convergence.py` *may take many seconds or even a couple of minutes to run, as it has to solve on some rather fine meshes in order to check convergence.*

CHAPTER

SEVEN

DIRICHLET BOUNDARY CONDITIONS

The Helmholtz problem we solved in the previous part was chosen to have homogeneous Neumann or *natural* boundary conditions, which can be implemented simply by cancelling the zero surface integral. We can now instead consider the case of Dirichlet, or *essential* boundary conditions. Instead of the Helmholtz problem we solved before, let us now specify a Poisson problem with homogeneous Dirichlet conditions, find u in some finite element space V such that:

$$\begin{aligned} -\nabla^2 u &= f \\ u &= 0 \text{ on } \Gamma \end{aligned} \tag{7.1}$$

In order to implement the Dirichlet conditions, we need to decompose V into two parts:

$$V = V_0 \oplus V_\Gamma \tag{7.2}$$

where V_Γ is the space spanned by those functions in the basis of V which are non-zero on Γ, and V_0 is the space spanned by the remaining basis functions (i.e. those basis functions which vanish on Γ). It is a direct consequence of the nodal nature of the basis that the basis functions for V_Γ are those corresponding to the nodes on Γ while the basis for V_0 is composed of all the other functions.

We now write the weak form of (7.1), find $u = u_0 + u_\Gamma$ with $u_0 \in V_0$ and $u_\Gamma \in V_\Gamma$ such that:

$$\begin{aligned} \int_\Omega \nabla v_0 \cdot \nabla(u_0 + u_\Gamma)\,\mathrm{d}x - \underbrace{\int_\Gamma v_0 \nabla(u_0 + u_\Gamma)\cdot \mathbf{n}\,\mathrm{d}s}_{=0} = \int_\Omega v_0\, f\,\mathrm{d}x \qquad & \forall v_0 \in V_0 \\ u_\Gamma = 0 \qquad & \text{on } \Gamma \end{aligned} \tag{7.3}$$

There are a number of features of this equation which require some explanation:

1. We only test with functions from V_0. This is because it is only necessary that the differential equation is satisfied on the interior of the domain: on the boundary of the domain we need only satisfy the boundary conditions.
2. The surface integral now cancels because v_0 is guaranteed to be zero everywhere on the boundary.
3. The u_Γ definition actually implies that $u_\Gamma = 0$ everywhere, since all of the nodes in V_Γ lie on the boundary.

This means that the weak form is actually:

$$\begin{aligned} \int_\Omega \nabla v_0 \cdot \nabla u\,\mathrm{d}x = \int_\Omega v_0\, f\,\mathrm{d}x \qquad & \forall v_0 \in V_0 \\ u_\Gamma &= 0 \end{aligned} \tag{7.4}$$

7.1 An algorithm for homogeneous Dirichlet conditions

The implementation of homogeneous Dirichlet conditions is actually rather straightforward.

1. The system is assembled completely ignoring the Dirichlet conditions. This results in a global matrix and vector which are correct on the rows corresponding to test functions in V_0, but incorrect on the V_Γ rows.
2. The global vector rows corresponding to boundary nodes are set to 0.
3. The global matrix rows corresponding to boundary nodes are set to 0.
4. The diagonal entry on each matrix row corresponding to a boundary node is set to 1.

This has the effect of replacing the incorrect boundary rows of the system with the equation $u_i = 0$ for all boundary node numbers i.

Hint

This algorithm has the unfortunate side effect of making the global matrix non-symmetric. If a symmetric matrix is required (for example in order to use a symmetric solver), then forward substition can be used to zero the boundary columns in the matrix, but that is beyond the scope of this module.

7.2 Implementing boundary conditions

Let:

$$f = \left(16\pi^2(x_1-1)^2x_1^2 - 2(x_1-1)^2 - 8(x_1-1)x_1 - 2x_1^2\right)\sin(4\pi x_0)$$

With this definition, (7.4) has solution:

$$u = \sin(4\pi x_0)(x_1-1)^2x_1^2$$

Exercise 7.1 *`fe_utils/solvers/poisson.py` contains a partial implementation of this problem. You need to implement the `assemble()` function. You should base your implementation on your `fe_utils/solvers/helmholtz.py` but take into account the difference in the equation, and the boundary conditions. The `fe_utils.solvers.poisson.boundary_nodes()` function in `fe_utils/solvers/poisson.py` is likely to be helpful in implementing the boundary conditions. As before, run:*

```
python fe_utils/solvers/poisson.py --help
```

for instructions (they are the same as for `fe_utils/solvers/helmholtz.py`). Similarly, `test/test_12_poisson_convergence.py` contains convergence tests for this problem.

7.3 Inhomogeneous Dirichlet conditions

The algorithm described here can be extended to inhomogeneous systems by setting the entries in the global vector to the value of the boundary condition at the corresponding boundary node. This additional step is required for the mastery exercise, but will be explained in more detail in the next section.

CHAPTER

EIGHT

NONLINEAR PROBLEMS

The finite element method may also be employed to numerically solve *nonlinear* PDEs. In order to do this, we can apply the classical technique for solving nonlinear systems: we employ an iterative scheme such as Newton's method to create a sequence of linear problems whose solutions converge to the correct solution to the nonlinear problem.

> **Warning**
>
> This chapter formed the content of the mastery material in some years, but does not currently do so. It is presented for reference only.

8.1 A model problem

As a simple case of a non-linear PDE, we can consider a steady non-linear diffusion equation. This is similar to the Poisson problem, except that the diffusion rate now depends on the value of the solution:

$$\begin{aligned} -\nabla\cdot((u+1)\nabla u) &= g\\ u &= b \text{ on } \Gamma \end{aligned} \tag{8.1}$$

where g and b are given functions defined over Ω and Γ respectively.

We can create the weak form of (8.1) by integrating by parts and taking the boundary conditions into account. The problem becomes, find $u \in V$ such that:

$$\begin{aligned} \int_\Omega \nabla v_0\cdot(u+1)\nabla u \,\mathrm{d}x &= \int_\Omega v_0 g\,\mathrm{d}x \qquad \forall v_0\in V_0\\ u_\Gamma &= b. \end{aligned} \tag{8.2}$$

Once more, V_0 is the subspace of V spanned by basis functions which vanish on the boundary, $V = V_0 \oplus V_\Gamma$, and $u = u_0 + u_\Gamma$ with $u_0 \in V_0$ and $u_\Gamma \in V_\Gamma$. This is corresponds directly with the weak form of the Poisson equation we already met. However, (8.2) is still nonlinear in u so we cannot simply substitute $u = u_i\phi_i$ in order to obtain a linear matrix system to solve.

8.2 Residual form

The general weak form of a non-linear problem is, find $u \in V$ such that:

$$f(u;v) = 0 \qquad \forall v \in V \tag{8.3}$$

The use of a semicolon is a common convention to indicate that f is assumed to be linear in the arguments after the semicolon, but might be nonlinear in the arguments before the semicolon. In this case, we observe that f may be nonlinear in u but is (by construction) linear in v.

The function f is called the *residual* of the nonlinear system. In essence, $f(u;v)=0\,\forall v\in V$ if and only if u is a weak solution to the PDE. Since the residual is linear in v, it suffices to define the residual for each ϕ_i in the basis of V. For $\phi_i\in V_0$, the residual is just the weak form of the equation, but what do we do for the boundary? The simple answer is that we need a linear functional which is zero if the boundary condition is satisfied at this test function, and nonzero otherwise. The simplest example of such a functional is:

$$f(u;\phi_i)=\phi_i^*(u)-\phi_i^*(b) \tag{8.4}$$

where ϕ_i^* is the node associated with basis function ϕ_i. For point evaluation nodes, $\phi_i^*(u)$ is the value of the proposed solution at node point i and $\phi_i^*(b)$ is just the boundary condition evaluated at that same point.

So for our model problem, we now have a full statement of the residual in terms of a basis function ϕ_i:

$$f(u;\phi_i)=\begin{cases}\displaystyle\int_\Omega \nabla\phi_i\cdot((u+1)\nabla u)-\phi_i g\,\mathrm{d}x & \phi_i\in V_0\\ \phi_i^*(u)-\phi_i^*(b) & \phi_i\in V_\Gamma\end{cases} \tag{8.5}$$

Hint

Evaluating the residual requires that the boundary condition be evaluated at the boundary nodes. A simple (if slightly inefficient) way to achieve this is to interpolate the boundary condition onto a function $\hat{b}\in V$.

8.3 Linearisation and Gâteaux Derivatives

Having stated our PDE in residual form, we now need to linearise the problem and thereby employ a technique such as Newton's method. In order to linearise the residual, we need to differentiate it with respect to u. Since u is not a scalar real variable, but is instead a function in V, the appropriate form of differentiation is the Gâteaux Derivative, given by:

$$J(u;v,\hat{u})=\lim_{\epsilon\to 0}\frac{f(u+\epsilon\hat{u};v)-f(u;v)}{\epsilon}. \tag{8.6}$$

Here, the new argument $\hat{u}\in V$ indicates the "direction" in which the derivative is to be taken. Let's work through the Gâteaux Derivative for the residual of our model problem. Assume first that $v\in V_0$. Then:

$$\begin{aligned}J(u;v,\hat{u})&=\lim_{\epsilon\to 0}\frac{\displaystyle\int_\Omega \nabla v\cdot((u+\epsilon\hat{u}+1)\nabla(u+\epsilon\hat{u}))-vg\,\mathrm{d}x-\int_\Omega\nabla v\cdot((u+1)\nabla u)-vg\,\mathrm{d}x}{\epsilon}\\ &=\lim_{\epsilon\to 0}\frac{\displaystyle\int_\Omega\nabla v\cdot\left(\epsilon\hat{u}\nabla u+(u+1)\nabla(\epsilon\hat{u})+\epsilon\hat{u}\nabla(\epsilon\hat{u})\right)\,\mathrm{d}x}{\epsilon}\\ &=\int_\Omega\nabla v\cdot\left(\hat{u}\nabla u+(u+1)\nabla\hat{u}\right)\,\mathrm{d}x.\end{aligned} \tag{8.7}$$

Note that, as expected, J is linear in $\hat{u}$.

Next, we can work out the boundary case by assuming $v=\phi_i$, one of the basis functions of V_Γ:

$$\begin{aligned}J(u;\phi_i,\hat{u})&=\lim_{\epsilon\to 0}\frac{\phi_i^*(u+\epsilon\hat{u})-\phi_i^*(b)-(\phi_i^*(u)-\phi_i^*(b))}{\epsilon}\\ &=\phi_i^*(\hat{u})\qquad\text{since }\phi_i^*(\cdot)\text{ is linear.}\end{aligned} \tag{8.8}$$

Once again, we can observe that J is linear in $\hat{u}$. Indeed, if we choose $\hat{u}=\phi_j$ for some ϕ_j in the basis if V then the definition of a nodal basis gives us:

$$J(u;\phi_i,\phi_j)=\delta_{ij} \tag{8.9}$$

8.4 A Taylor expansion and Newton's method

Since we now have the derivative of the residual with respect to a perturbation to the prospective solution u, we can write the first terms of a Taylor series approximation for the value of the residual at a perturbed solution $u+\hat{u}$:

$$f(u+\hat{u};v) = f(u;v) + J(u;v,\hat{u}) + \ldots \qquad \forall v \in V. \tag{8.10}$$

Now, just as in the scalar case, Newton's method consists of approximating the function (the residual) by the first two terms and solving for the update that will set these terms to zero. In other words:

$$u^{n+1} = u^n + \hat{u} \tag{8.11}$$

where $\hat{u} \in V$ is the solution to:

$$J(u^n;v,\hat{u}) = -f(u^n;v) \qquad \forall v \in V. \tag{8.12}$$

In fact, (8.12) is simply a linear finite element problem! To make this explicit, we can expand v and $\hat{u}$ in terms of basis functions $\{\phi_i\}_{i=0}^{n-1} \in V$ such that $v = \sum_i v_i\phi_i$ and $\hat{u} = \sum_j \hat{u}_j\phi_j$. We note, as previously, that we we can drop the coefficients v_i giving:

$$\sum_j J(u^n;\phi_i,\phi_j)\hat{u}_j = -f(u^n;\phi_i) \qquad \forall\, 0 \leq i < n. \tag{8.13}$$

For our nonlinear diffusion problem, the matrix J is given by:

$$J_{ij} = J(u^n;\phi_i,\phi_j) = \begin{cases} \displaystyle\int_\Omega \nabla\phi_i \cdot \left(\phi_j\nabla u^n + (u^n+1)\nabla\phi_j\right)\,\mathrm{d}x & \phi_i \in V_0 \\ \delta_{ij} & \phi_i \in V_\Gamma, \end{cases} \tag{8.14}$$

and the right hand side vector f is given by (8.5). This matrix, J, is termed the *Jacobian matrix* of f.

8.4.1 Stopping criteria for Newton's method

Since Newton's method is an iterative algorithm, it creates a (hopefully convergent) sequence of approximations to the correct solution to the original nonlinear problem. How do we know when to accept the solution and terminate the algorithm?

The answer is that the update, $\hat{u}$ which is calculated at each step of Newton's method is itself an approximation to the error in the solution. It is therefore appropriate to stop Newton's method when this error estimate becomes sufficiently small in the L^2 norm.

The observant reader will observe that $\hat{u}$ is in fact an estimate of the error in the *previous* step. This is indeed true: the Newton step is both an estimate of the previous error and a correction to that error. However, having calculated the error estimate, it is utterly unreasonable to not apply the corresponding correction.

Note

Note!

Another commonly employed stopping mechanism is to consider the size of the residual f. However, the residual is not actually a function in V, but is actually a linear operator in V^*. Common practice would be to identify f with a function in V by simply taking the function whose coefficients match those of f. The L^2 or l^2 norm is then taken of this function and this value is used to determine when convergence has occured.

This approach effectively assumes that the Riesz map on V is the trivial operator which identifies the basis function coefficients. This would be legitimate were the inner product on V the l^2 dot product. However, since the inner product on V is defined by an integral, the mesh resolution is effectively encoded into f. This means that this approach produces convergence rates which depend on the level of mesh refinement.

Avoiding this mesh dependency requires the evaluation of an operator norm or, equivalently, the solution of a linear system in order to find the Riesz representer of f in V. However, since the error-estimator approach given above is both an actual estimate of the error in the solution, and requires no additional linear solves, it should be regarded as a preferable approach. For a full treatment of Newton methods, see [Deu11].

8.4.2 Stopping threshold values

What, then, qualifies as a sufficiently small value of our error estimate? There are two usual approaches:

relative tolerance
Convergence is deemed to occur when the estimate becomes sufficiently small compared with the first error estimate calculated. This is generally the more defensible approach since it takes into account the overall scale of the solution. 10^{-6} would be a reasonably common relative tolerance.

absolute tolerance
Computers employ finite precision arithmetic, so there is a limit to the accuracy which can ever be achieved. This is a difficult value to estimate, since it depends on the number and nature of operations undertaken in the algorithm. A common approach is to set this to a very small value (e.g. 10^{-50}) initially, in order to attempt to ensure that the relative tolerance threshold is hit. Only if it becomes apparent that the problem being solved is in a regime for which machine precision is a problem is a higher absolute tolerance set.

It is important to realise that both of these criteria involve making essentially arbitrary judgements about the scale of error which is tolerable. There is also a clear trade-off between the level of error tolerated and the cost of performing a large number of Newton steps. For realistic problems, it is therefore frequently expedient and/or necessary to tune the convergence criteria to the particular case.

In making these judgements, it is also important to remember that the error in the Newton solver is just one of the many sources of error in a calculation. It is pointless to expend computational effort in an attempt to drive the level of error in this component of the solver to a level which will be swamped by a larger error occurring somewhere else in the process.

8.4.3 Failure modes

Just as with the Newton method for scalar problems, Newton iteration is not guaranteed to converge for all nonlinear problems or for all initial guesses. If Newton's method fails to converge, then the algorithm presented so far constitutes an infinite loop. It is therefore necessary to define some circumstances in which the algorithm should terminate having failed to find a solution. Two such circumstances are commonly employed:

maximum iterations
It is a reasonable heuristic that Newton's method has failed if it takes a very large number of iterations. What constitutes "too many" is once again a somewhat arbitrary judgement, although if the approach takes many tens of iterations this should always be cause for reconsideration!

diverged error estimate
Newton's method is not guaranteed to produce a sequence of iterations which monotonically decrease the error, however if the error estimate has increased to, say, hundreds or thousands of times its initial value, this would once again be grounds for the algorithm to fail.

Note that these failure modes are heuristic: having the algorithm terminate for these reasons is really an instruction to the user to think again about the problem, the solver, and the initial guess.

8.5 Implementing a nonlinear problem

Note

This problem is intentionally stated in more general terms than the previous ones. It is your responsibility to decide on a code structure, to derive a method of manufactured solutions answer, and to create the convergence tests which demonstrate that your solution is correct.

Warning

This problem is not currently an assessable part of the module at Imperial College. It is presented here for reference.

Exercise 8.1 *The p-laplacian is a generalisation of the laplacian from a second derivative to an arbitrary derivative. It is nonlinear for $p \neq 2$.*

Implement `solve_mastery()` *so that it solves the following problem using degree 1 Lagrange elements over the unit square domain:*

$$\begin{aligned} -\nabla \cdot \left(|\nabla u|^{p-2}\nabla u\right) &= g \\ u &= b \text{ on } \Gamma \\ p &= 4 \end{aligned} \tag{8.15}$$

Select the solution $u = e^{xy}$ and compute the required forcing function g so that your solution solves the equations. Make sure your boundary condition function b is consistent with your chosen solution!

For this problem, it is not possible to use the zero function as an initial guess for Newton's method. A much better choice is to treat the 2-laplacian as an approximation to the 4-laplacian, and therefore to solve Poisson's equation first to obtain a good initial guess for the 4-laplacian problem.

Your submitted answer will consist of:

1. *A written component containing your derivation of:*
   a. *The weak form of (8.15); and*
   b. *the Jacobian; and*
   c. *the forcing term implied by the specified manufactured solution; and*
   d. *an explanation of why the zero function cannot be used as an initial guess for the solution.*

   A neatly hand-written or a typed submission are equally acceptable.

2. *The code to implement the solution. This should be in* `fe_utils.solvers.mastery.py` *in your implementation. A convergence test for your code is provided in* `test/test_12_mastery_convergence.py`.

   The submission of your mastery exercise, and indeed the entire implementation exercise will be on Blackboard. You will submit a PDF containing the derivations above, and the git sha1 for the commit you would like to have marked.

Hint

It is an exceptionally useful aid to debugging to have your Newton iteration print out the value of the error norm and the iteration number for each iteration. If you wish to see the printed output while running the test, you can pass the `-s` *option to* `py.test`.

Hint

You could parametrise your code by p. By setting $p = 2$, you reduce your problem to the linear case. You can use the linear case to test your code initially, before setting $p = 4$ for the actual exercise. Note that, in the linear case, Newton's method will converge in exactly one iteration (although your algorithm will have to actually calculate two steps in order to know that convergence has occurred).

CHAPTER

NINE

MIXED PROBLEMS

Note

This section is the mastery exercise for this module. This exercise is explicitly intended to test whether you can bring together what has been learned in the rest of the module in order to go beyond what has been covered in lectures and labs.

This exercise is not a part of the third year version of this module.

As an example of a mixed problem, let's take the Stokes problem presented in Section 6 of the analysis part of the course. The weak form of the Stokes problem presented in Definition 6.1 is find $(u, p) \in V \times Q$ such that:

$$\begin{aligned} a(u, v) + b(v, p) &= \int_\Omega f \cdot v \, \mathrm{d}\, x, \\ b(u, q) &= 0, \quad \forall (v, q) \in V \times Q, \end{aligned} \tag{9.1}$$

where

$$\begin{aligned} a(u, v) &= \mu \int_\Omega \epsilon(u) : \epsilon(v) \, \mathrm{d}\, x, \\ b(v, q) &= \int_\Omega q \nabla \cdot v \, \mathrm{d}\, x, \\ V &= (\mathring{H}^1(\Omega))^d, \\ Q &= \mathring{L}^2(\Omega), \end{aligned} \tag{9.2}$$

and where $(\mathring{H}^1(\Omega))^d$ is the subspace of $H^1(\Omega)^d$ for which all components vanish on the boundary, and $\mathring{L}^2(\Omega)$ is the subspace of $L^2(\Omega)$ for which the integral of the function over the domain vanishes. This last constraint was introduced to remove the null space of constant pressure functions from the system. This constraint introduces a little complexity into the implementation. So instead, we will redefine $\mathring{L}^2(\Omega)$ to be the subspace of $L^2(\Omega)$ constrained to take the value 0 at some arbitrary but specified point. For example, one might choose to require the pressure at the origin to vanish. This is also an effective way to remove the nullspace, but it is simpler to implement. We will implement the two-dimensional case ($d = 2$) and, for simplicity, we will assume $\mu = 1$.

The colon (:) indicates an inner product so:

$$\epsilon(u) : \epsilon(v) = \sum_{\alpha\beta} \epsilon(u)_{\alpha\beta} \epsilon(v)_{\alpha\beta} \tag{9.3}$$

In choosing a finite element subspace of $V \times Q$ we will similarly choose a simpler to implement, yet still stable, space than was chosen in Analysis Section 6. The space we will use is the lowest order Taylor-Hood space [TH73]. This has:

$$\begin{aligned} V^h &= P2(\Omega)^2 \\ Q^h &= P1(\Omega) \end{aligned} \tag{9.4}$$

i.e. quadratic velocity and linear pressure on each cell. We note that $Q^h \in H^1(\Omega)$ but since $H^1(\Omega) \subset L^2(\Omega)$, this is not itself a problem. We will impose further constraints on the spaces in the form of Dirichlet boundary conditions to ensure that the solutions found are in fact in $V \times Q$.

In addition to the finite element functionality we have already implemented, there are two further challenges we need to address. First, the implementation of the vector-valued space $P2(\Omega)^2$‘$mandsecond, theimplementationoffunctionsandmatricesdefinedoverthemixedspace$‘$V^h \times Q^h$.

9.1 Vector-valued finite elements

Consider the local representation of $P2(\Omega)^2$ on one element. The scalar $P2$ element on one triangle has 6 nodes, one at each vertex and one at each edge. If we write $\{\Phi_i\}_{i=0}^5$ for the scalar basis, then a basis $\{\mathbf{\Phi}_i\}_{i=0}^{11}$ for the vector-valued space is given by:

$$\mathbf{\Phi}_i(X) = \Phi_{i\,//\,2}(X)\,\mathbf{e}_{i\,\%\,2} \tag{9.5}$$

where $//$ is the integer division operator, $\%$ is the modulus operator, and $\mathbf{e}_0, \mathbf{e}_1$ is the standard basis for $\mathbb{R}^2$. That is to say, we interleave x and y component basis functions.

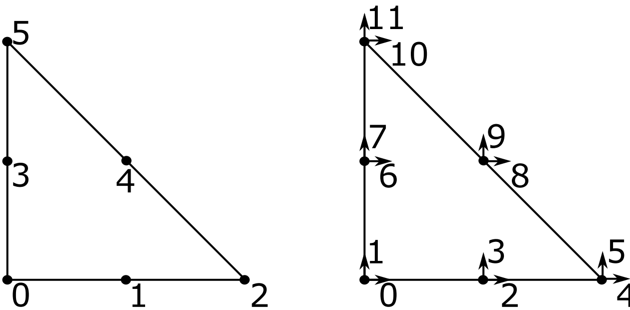

Fig. 9.1: The local numbering of vector degrees of freedom.

We can practically implement vector function spaces by implementing a new class `fe_utils.finite_elements.VectorFiniteElement`. The constructor (`__init__()`) of this new class should take a *`FiniteElement`* and construct the corresponding vector element. For current purposes we can assume that the vector element will always have a vector dimension equal to the element geometric and topological dimension (i.e. 2 if the element is defined on a triangle). We'll refer to this dimension as d.

9.1.1 Implementing `VectorFiniteElement`

`VectorFiniteElement` needs to implement as far as possible the same interface as *`FiniteElement`*. Let's think about how to do that.

`cell`, `degree`
: Same as for the input *`FiniteElement`*.

`entity_nodes`
: There will be twice as many nodes, and the node ordering is such that each node is replaced by a d-tuple. For example the scalar triangle P1 entity-node list is:

```
{
    0 : {0 : [0], 1 : [1], 2 : [2]},
    1 : {0 : [], 1 : [], 2 : []},
    2 : {0 : []}
}
```

The vector version is achieved by looping over the scalar version and returning a mapping with the pair $2n, 2(n+1)$ in place of node n:

```
{
    0 : {0 : [0, 1], 1 : [2, 3], 2 : [4, 5]},
    1 : {0 : [], 1 : [], 2 : []},
    2 : {0 : []}
}
```

nodes_per_entity:
Each entry will be d times that on the input *FiniteElement*.

9.1.2 Tabulation

In order to tabulate the element, we can use (9.5). We first call the tabulate method from the input *FiniteElement*, and we use this and (9.5) to produce the array to return. Notice that the array will both have a basis functions dimension which is d times longer than the input element, and will also have an extra dimension to account for the multiplication by $\mathbf{e}_{i\,\%\,d}$. This means that the tabulation array with `grad=False` will now be rank 3, and that with `grad=True` will be rank 4. Make sure you keep track of which rank is which! The `VectorFiniteElement` will need to keep a reference to the input *FiniteElement* in order to facilitate tabulation.

9.1.3 Nodes

Even though we didn't need the nodes of the `VectorFiniteElement` to construct its basis, we will need them to implement interpolation. In Definition 2.2 we learned that the nodes of a finite element are related to the corresponding nodal basis by:

$$\mathbf{\Phi}_i^*(\mathbf{\Phi}_j) = \delta_{ij} \tag{9.6}$$

From (9.5) and assuming, as we have throughout the course, that the scalar finite element has point evaluation nodes given by:

$$\Phi_i(v) = v(X_i), \tag{9.7}$$

it follows that:

$$\begin{aligned}\mathbf{\Phi}_i^*(v) &= \Phi_{i\,//\,d}^*(\mathbf{e}_{i\,\%\,d} \cdot v)\\ &= \mathbf{e}_{i\,\%\,d} \cdot v(X_{i\,//\,d})\end{aligned} \tag{9.8}$$

Hint

To see that this is the correct nodal basis, choose $v = \mathbf{\Phi}_j$ in (9.8) and substitute (9.5) for $\mathbf{\Phi}_j$.

This means that the correct `VectorFiniteElement.nodes` attribute is the list of nodal points from the input *FiniteElement* but with each point repeated d times. It will also be necessary to add another attribute, perhaps `node_weights` which is a rank 2 array whose i-th row is the correct canonical basis vector to contract with the function value at the i-th node ($\mathbf{e}_{i\,\%\,d}$).

9.2 Vector-valued function spaces

Assuming we correctly implement `VectorFiniteElement`, *FunctionSpace* should work out of the box. In particular, the global numbering algorithm only depends on having a correct local numbering so this should work unaltered.

9.3 Functions in vector-valued spaces

The general form of a function in a vector-valued function space is:

$$f = f_i \mathbf{\Phi}_i(X). \tag{9.9}$$

That is to say, the basis functions are vector valued and their coefficients are still scalar. This means that if the `VectorFiniteElement` had a correct entity-node list then the core functionality of the existing *Function* will automatically be correct. In particular, the array of values will have the correct extent. However, interpolation and plotting of vector valued fields will require some adjustment.

9.3.1 Interpolating into vector-valued spaces

Since the form of the nodes of a `VectorFiniteElement` is different from that of a scalar element, there will be some changes required in the *interpolate()* method. Specifically:

```
self.values[fs.cell_nodes[c, :]] = [fn(x) for x in node_coords]
```

This line will need to take into account the dot product with the canonical basis from (9.8), which you have implemented as `VectorFiniteElement.node_weights`. This change will need to be made conditional on the class of finite element passed in, so that the code doesn't break in the scalar element case.

9.3.2 Plotting functions in vector-valued spaces

The coloured surface plots that we've used thus far for two-dimensional scalar functions don't extend easily to vector quantities. Instead, a frequently used visualisation technique is the quiver plot. This draws a set of arrows representing the function value at a set of points. For our purposes, the nodes of the function space in question are a good choice of evaluation points. Listing 9.1 provides the code you will need. Notice that at line 3 we interpolated the function $f(x) = x$ into the function space in order to obtain a list of the global coordinates of the node locations. At lines 6 and 7 we use what we know about the node ordering to recover vector values from the list of basis function coefficients.

Listing 9.1: Code implementing quiver plots to visualise functions in vector function spaces. This code should be added to *plot()* immediately after the definition of `fs`.

```
if isinstance(fs.element, VectorFiniteElement):
    coords = Function(fs)
    coords.interpolate(lambda x: x)
    fig = plt.figure()
    ax = fig.add_subplot()
    x = coords.values.reshape(-1, 2)
    v = self.values.reshape(-1, 2)
    plt.quiver(x[:, 0], x[:, 1], v[:, 0], v[:, 1])
    plt.show()
    return
```

Once this code has been inserted, then running the code in Listing 9.2 will result in a plot rather like Fig. 9.2.

Listing 9.2: Creation of a vector function space, interpolation of a given function into it, and subsequent plot creation.

```
from fe_utils import *
from math import cos, sin, pi

se = LagrangeElement(ReferenceTriangle, 2)
```

(continues on next page)

(continued from previous page)

```
ve = VectorFiniteElement(se)
m = UnitSquareMesh(10,10)
fs = FunctionSpace(m, ve)
f = Function(fs)
f.interpolate(lambda x: (2*pi*(1 - cos(2*pi*x[0]))*sin(2*pi*x[1]),
                        -2*pi*(1 - cos(2*pi*x[1]))*sin(2*pi*x[0])))
f.plot()
```

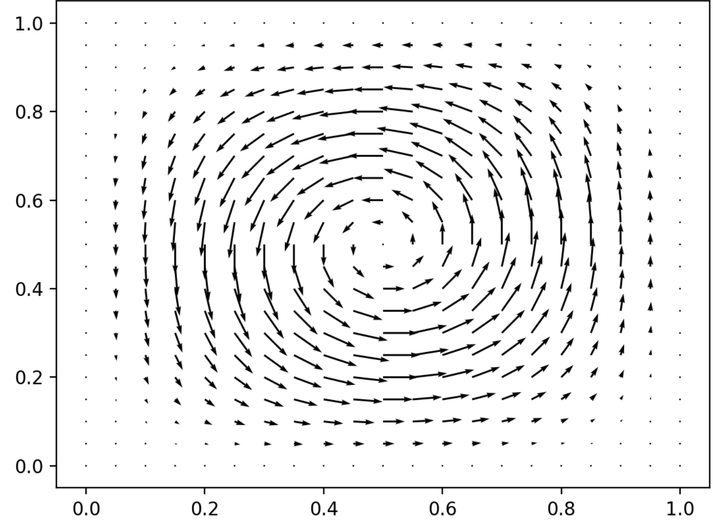

Fig. 9.2: The quiver plot resulting from Listing 9.2.

9.3.3 Solving vector-valued systems

Solving a finite element problem in a vector-valued space is essentially similar to the scalar problems you have already solved. It does, nonetheless, provide a useful check on the correctness of your code before adding in the additional complications of mixed systems. As a very simple example, consider computing vector-valued field which is the gradient of a known function. For some suitable finite element space $V \subset H^1(\Omega)^2$ and $f : \Omega \to \mathbb{R}$, find $u \in V$ such that:

$$\int_\Omega u \cdot v \,\mathrm{d}x = \int_\Omega \nabla f \cdot v \,\mathrm{d}x \quad \forall v \in V. \tag{9.10}$$

If f is chosen such that $\nabla f \in V$ then this projection is exact up to roundoff, and the following calculation should result in a good approximation to zero:

$$e = \int_\Omega (u - \nabla f) \cdot (u - \nabla f) \,\mathrm{d}x \tag{9.11}$$

Note

The computations in this subsection are not required to complete the mastery exercise. They are, nonetheless, strongly recommended as a mechanism for checking your implementation thus far.

9.4 Mixed function spaces

The Stokes equations are defined over the mixed function space $W = V \times Q$. Here "mixed" simply means that there are two solution variables, and therefore two solution spaces. Functions in W are pairs (u, p) where $u \in V$ and $p \in Q$. If $\{\phi_i\}_{i=0}^{m-1}$ is a basis for V and $\{\psi_j\}_{j=0}^{n-1}$ then a basis for W is given by:

$$\{\omega_i\}_{i=0}^{m+n-1} = \{(\phi_i, 0)\}_{i=0}^{m-1} \cup \{(0, \psi_{j-n})\}_{j=m}^{m+n-1}. \tag{9.12}$$

This in turn enables us to write $w \in W$ in the form $w = w_i\omega_i$ as we would expect for a function in a finite element space. The Cartesian product structure of the mixed space W means that the first m coefficients are simply the coefficients of the V basis functions, and the latter n coefficients correspond to the Q basis functions. This means that our full mixed finite element system is simply a linear system of block matrices and block vectors. If we disregard boundary conditions, including the pressure constraint, this system has the following form:

$$\begin{bmatrix} A & B^{\mathrm{T}} \\ B & 0 \end{bmatrix} \begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{9.13}$$

where:

$$\begin{aligned} A_{ij} &= a(\phi_j, \phi_i), \\ B_{ij} &= b(\phi_j, \psi_i), \\ F_i &= \int_\Omega f \cdot v \, \mathrm{d}\, x, \\ U_i &= u_i = w_i, \\ P_i &= p_i = w_{i+m}. \end{aligned} \tag{9.14}$$

This means that the assembly of the mixed problem comes down to the assembly of several finite operators of the form that we have already encountered. These then need to be assembled into the full block matrix and right hand side vector, before the system is solved and the resulting solution vector pulled appart and interpreted as the coefficients of u and p. Observe in (9.14) that the order of the indices i and j is reversed on the right hand side of the equations. This reflects the differing conventions for matrix indices and bilinear form arguments, and is a source of unending confusion in this field.

9.4.1 Assembling block systems

The procedure for assembling the individual blocks of the block matrix and the block vectors is the one you are familiar with, but we will need to do something new to assemble the block structures. What is required differs slightly between the matrix and the vectors.

In the case of the vectors, then it is sufficient to know that a slice into a `numpy.ndarray` returns a view on the same memory as the full vector. This is most easily understood through an example:

```
In [1]: import numpy as np

In [2]: a = np.zeros(10)

In [3]: b = a[:5]

In [4]: b[2] = 1

In [5]: a
Out[5]: array([0., 0., 1., 0., 0., 0., 0., 0., 0., 0.])
```

This means that one can first create a full vector of length $n + m$ and then slice it to create subvectors that can be used for assembly.

Conversely, `scipy.sparse` provides the `bmat()` function which will stitch together a larger sparse matrix from blocks. In order to have the full indexing options you are likely to want for imposing the boundary conditions, you will probably want to specify that the resulting matrix is in `"lil"` format.

9.4.2 Boundary conditions

The imposition of the constraint in $(\mathring{H}^1(\Omega))^2$ that solutions vanish on the boundary is a Dirichlet condition of the type that you have encountered before. Observe that the condition changes the test space, which affects whole rows of the block system, so you will want to impose the boundary condition *after* assembling the block matrix. You will also need to ensure that the constraint is applied to both the x and y components of the space.

The imposition of the constraint in $\mathring{L}^2(\Omega)$ that the solution is zero at some prescribed point can be achieved by selecting an arbitrary basis function and applying a zero Dirichlet condition for that degree of freedom. In this regard we can observe that there is nothing about the implementation of Dirichlet conditions that constrains them to lie on the boundary. Rather, they should be understood as specifying a subspace on which the solution is prescribed rather than solved for. In this particular case, that subspace is one-dimensional.

9.4.3 Solving the matrix system

The block matrix system that you eventually produce will be larger than many of those we have previously encountered, and will have non-zero entries further from the diagonal. This can cause the matrix solver to become expensive in both time and memory. Fortunately, `scipy.sparse.linalg` now incorporates an interface to SuperLU, which is a high-performance direct sparse solver. The recommended solution strategy is therefore:

1. Convert your block matrix to `scipy.sparse.csc_matrix`, which is the format that SuperLU requires.
2. Factorise the matrix using `scipy.sparse.linalg.splu()`.
3. Use the resulting `SuperLU` object to finally solve the system.

9.4.4 Computing the error

We will wish to compute the convergence of our solution in the L^2 norm. For $w \in W$, this is given by:

$$\|w\|_{L^2} = \sqrt{\int_\Omega w \cdot w \,\mathrm{dx}} \tag{9.15}$$

When we expand this in terms of the basis of W, it will be useful to note that basis functions from the different component spaces are orthogonal. That is to say:

$$(\phi, 0) \cdot (0, \psi) = 0 \quad \forall \phi \in V,\, \psi \in Q. \tag{9.16}$$

The direct result of this is that if $w = (u, p)$ then:

$$\|w\|_{L^2}^2 = \|u\|_{L^2}^2 + \|p\|_{L^2}^2. \tag{9.17}$$

9.5 Manufacturing a solution to the Stokes equations

As previously, we will wish to check our code using the method of manufactured solutions. The Stokes equations represent a form of incompressible fluid mechanics, so it is usually preferable to select a target solution for which $\nabla \cdot u = 0$. The straightforward way to do this is to choose a scalar field $\gamma : \Omega \to \mathbb{R}$ to use as a streamfunction. We can then define $u = \nabla^\perp \gamma$ and rely on the vector calculus identity $\nabla \cdot \nabla^\perp \gamma = 0$ to guarantee that the velocity field is divergence-free. We also need to ensure that u satisfies the boundary conditions, which amounts to choosing γ such that its gradient vanishes on the domain boundary. The following function is a suitable choice on a unit square domain:

$$\gamma(x, y) = \big(1 - \cos(2\pi x)\big)\big(1 - \cos(2\pi y)\big) \tag{9.18}$$

9.6 Implementing the Stokes problem

Exercise 9.1 *The goal of this exercise is to implement a solver for the Stokes equations, on a unit square. Implement* `solve_mastery()` *so that it solves* (9.1) *using the forcing function derived from* (9.18).

Your full solution should:

1. *Implement* `VectorFiniteElement`.
2. *Make the consequential changes to* `Function` *to enable values to be interpolated into vector-valued functions, and to create quiver plots.*
3. *Assemble and solve the required mixed system.*
4. *Compute the L^2 error of the mixed solution from the analytic solution.*

A convergence test for your code is provided in `test/test_13_mastery_convergence.py`. *In order to be compatible with this code, your implementation of* `solve_mastery()` *should return its results as a tuple of the form* `(u, p), error`. *This is a slight change from the comment in the code which takes into account that the problem is mixed. The obvious consequential change will be needed at the end of* `fe_utils.solvers.mastery`.

CHAPTER

TEN

FE_UTILS PACKAGE

10.1 Subpackages

10.1.1 fe_utils.scripts package

Submodules

fe_utils.scripts.plot_function_space_nodes module

fe_utils.scripts.plot_function_space_nodes.**plot_function_space_nodes**()

fe_utils.scripts.plot_interpolate_lagrange module

fe_utils.scripts.plot_interpolate_lagrange.**plot_interpolate_lagrange**()

fe_utils.scripts.plot_lagrange_basis_functions module

fe_utils.scripts.plot_lagrange_basis_functions.**plot_lagrange_basis_functions**()

fe_utils.scripts.plot_lagrange_points module

fe_utils.scripts.plot_lagrange_points.**plot_lagrange_points**()

fe_utils.scripts.plot_mesh module

fe_utils.scripts.plot_mesh.**plot_mesh**()

fe_utils.scripts.plot_sin_function module

fe_utils.scripts.plot_sin_function.**plot_sin_function**()

Module contents

10.1.2 fe_utils.solvers package

Submodules

fe_utils.solvers.helmholtz module

Solve a model helmholtz problem using the finite element method. If run as a script, the result is plotted. This file can also be imported as a module and convergence tests run on the solver.

`fe_utils.solvers.helmholtz.`**`assemble`**`(`*fs*, *f*`)`

Assemble the finite element system for the Helmholtz problem given the function space in which to solve and the right hand side function.

`fe_utils.solvers.helmholtz.`**`solve_helmholtz`**`(`*degree*, *resolution*, *analytic=False*, *return_error=False*`)`

Solve a model Helmholtz problem on a unit square mesh with `resolution` elements in each direction, using equispaced Lagrange elements of degree `degree`.

fe_utils.solvers.mastery module

Solve a nonlinear problem using the finite element method. If run as a script, the result is plotted. This file can also be imported as a module and convergence tests run on the solver.

`fe_utils.solvers.mastery.`**`solve_mastery`**`(`*resolution*, *analytic=False*, *return_error=False*`)`

This function should solve the mastery problem with the given resolution. It should return both the solution `Function` and the L^2 error in the solution.

If `analytic` is `True` then it should not solve the equation but instead return the analytic solution. If `return_error` is true then the difference between the analytic solution and the numerical solution should be returned in place of the solution.

fe_utils.solvers.poisson module

Solve a model Poisson problem with Dirichlet boundary conditions.

If run as a script, the result is plotted. This file can also be imported as a module and convergence tests run on the solver.

`fe_utils.solvers.poisson.`**`assemble`**`(`*fs*, *f*`)`

Assemble the finite element system for the Poisson problem given the function space in which to solve and the right hand side function.

`fe_utils.solvers.poisson.`**`boundary_nodes`**`(`*fs*`)`

Find the list of boundary nodes in fs. This is a unit-square-specific solution. A more elegant solution would employ the mesh topology and numbering.

`fe_utils.solvers.poisson.`**`solve_poisson`**`(`*degree*, *resolution*, *analytic=False*, *return_error=False*`)`

Solve a model Poisson problem on a unit square mesh with `resolution` elements in each direction, using equispaced Lagrange elements of degree `degree`.

Module contents

10.2 Submodules

10.3 fe_utils.finite_elements module

class fe_utils.finite_elements.**FiniteElement**(*cell*, *degree*, *nodes*, *entity_nodes=None*)

Bases: `object`

A finite element defined over cell.

Parameters

- **cell** – the *ReferenceCell* over which the element is defined.
- **degree** – the polynomial degree of the element. We assume the element spans the complete polynomial space.
- **nodes** – a list of coordinate tuples corresponding to point evaluation node locations on the element.
- **entity_nodes** – a dictionary of dictionaries such that entity_nodes[d][i] is the list of nodes associated with entity *(d, i)* of dimension *d* and index *i*.

Most of the implementation of this class is left as exercises.

cell

The *ReferenceCell* over which the element is defined.

degree

The polynomial degree of the element. We assume the element spans the complete polynomial space.

entity_nodes

A dictionary of dictionaries such that `entity_nodes[d][i]` is the list of nodes associated with entity *(d, i)*.

interpolate(*fn*)

Interpolate fn onto this finite element by evaluating it at each of the nodes.

Parameters

fn – A function `fn(X)` which takes a coordinate vector and returns a scalar value.

Returns

A vector containing the value of `fn` at each node of this element.

The implementation of this method is left as an *exercise*.

node_count

The number of nodes in this element.

nodes

The list of coordinate tuples corresponding to the nodes of the element.

nodes_per_entity

`nodes_per_entity[d]` is the number of entities associated with an entity of dimension d.

tabulate(*points*, *grad=False*)

Evaluate the basis functions of this finite element at the points provided.

Parameters

- **points** – a list of coordinate tuples at which to tabulate the basis.
- **grad** – whether to return the tabulation of the basis or the tabulation of the gradient of the basis.

Result

an array containing the value of each basis function at each point. If *grad* is *True*, the gradient vector of each basis vector at each point is returned as a rank 3 array. The shape of the array is (points, nodes) if `grad` is `False` and (points, nodes, dim) if `grad` is `True`.

The implementation of this method is left as an *exercise*.

`class fe_utils.finite_elements.`**`LagrangeElement`**`(`*cell*, *degree*`)`

Bases: `FiniteElement`

An equispaced Lagrange finite element.

Parameters

- **`cell`** – the `ReferenceCell` over which the element is defined.
- **`degree`** – the polynomial degree of the element. We assume the element spans the complete polynomial space.

The implementation of this class is left as an *exercise*.

`fe_utils.finite_elements.`**`lagrange_points`**`(`*cell*, *degree*`)`

Construct the locations of the equispaced Lagrange nodes on cell.

Parameters

- **`cell`** – the `ReferenceCell`
- **`degree`** – the degree of polynomials for which to construct nodes.

Returns

a rank 2 `array` whose rows are the coordinates of the nodes.

The implementation of this function is left as an *exercise*.

`fe_utils.finite_elements.`**`vandermonde_matrix`**`(`*cell*, *degree*, *points*, *grad=False*`)`

Construct the generalised Vandermonde matrix for polynomials of the specified degree on the cell provided.

Parameters

- **`cell`** – the `ReferenceCell`
- **`degree`** – the degree of polynomials for which to construct the matrix.
- **`points`** – a list of coordinate tuples corresponding to the points.
- **`grad`** – whether to evaluate the Vandermonde matrix or its gradient.

Returns

the generalised *Vandermonde matrix*

The implementation of this function is left as an *exercise*.

10.4 fe_utils.function_spaces module

`class fe_utils.function_spaces.`**`Function`**`(`*function_space*, *name=None*`)`

Bases: `object`

A function in a finite element space. The main role of this object is to store the basis function coefficients associated with the nodes of the underlying function space.

Parameters

- **`function_space`** – The `FunctionSpace` in which this `Function` lives.
- **`name`** – An optional label for this `Function` which will be used in output and is useful for debugging.

function_space

The *FunctionSpace* in which this *Function* lives.

integrate()

Integrate this *Function* over the domain.

Result
The integral (a scalar).

interpolate(*fn*)

Interpolate a given Python function onto this finite element *Function*.

Parameters
fn – A function `fn(X)` which takes a coordinate vector and returns a scalar value.

name

The (optional) name of this *Function*

plot(*subdivisions=None*)

Plot the value of this *Function*. This is quite a low performance plotting routine so it will perform poorly on larger meshes, but it has the advantage of supporting higher order function spaces than many widely available libraries.

Parameters
subdivisions – The number of points in each direction to use in representing each element. The default is $2d + 1$ where d is the degree of the *FunctionSpace*. Higher values produce prettier plots which render more slowly!

values

The basis function coefficient values for this *Function*

class fe_utils.function_spaces.**FunctionSpace**(*mesh*, *element*)

Bases: `object`

A finite element space.

Parameters

- **mesh** – The *Mesh* on which this space is built.
- **element** – The *FiniteElement* of this space.

Most of the implementation of this class is left as an *exercise*.

cell_nodes

The global cell node list. This is a two-dimensional array in which each row lists the global nodes incident to the corresponding cell. The implementation of this member is left as an *exercise*

element

The *FiniteElement* of this space.

mesh

The *Mesh* on which this space is built.

node_count

The total number of nodes in the function space.

10.5 fe_utils.mesh module

class `fe_utils.mesh.`**`Mesh`**(*vertex_coords*, *cell_vertices*)

Bases: `object`

A one or two dimensional mesh composed of intervals or triangles respectively.

Parameters

- **`vertex_coords`** – a vertex_count x dim array of the coordinates of the vertices in the mesh.
- **`cell_vertices`** – a cell_count x (dim+1) array of the indices of the vertices of which each cell is made up.

`adjacency`(*dim1*, *dim2*)

Return the set of *dim2* entities adjacent to each *dim1* entity. For example if *dim1==2* and *dim2==1* then return the list of edges (1D entities) adjacent to each triangle (2D entity).

The return value is a rank 2 `numpy.array` such that `adjacency(dim1, dim2)[e1, :]` is the list of dim2 entities adjacent to entity `(dim1, e1)`.

This operation is only defined where *self.dim >= dim1 > dim2*.

This method is simply a more systematic way of accessing *`edge_vertices`*, *`cell_edges`* and *`cell_vertices`*.

`cell`

The *`ReferenceCell`* of which this *`Mesh`* is composed.

`cell_edges`

The indices of the edges incident to each cell (only for 2D meshes).

`cell_vertices`

The indices of the vertices incident to cell.

`dim`

The geometric and topological dimension of the mesh. Immersed manifolds are not supported.

`edge_vertices`

The indices of the vertices incident to edge (only for 2D meshes).

`entity_counts`

The number of entities of each dimension in the mesh. So `entity_counts[0]` is the number of vertices in the mesh.

`jacobian`(*c*)

Return the Jacobian matrix for the specified cell.

Parameters

`c` – The number of the cell for which to return the Jacobian.

Result

The Jacobian for cell `c`.

`vertex_coords`

The coordinates of all the vertices in the mesh.

class `fe_utils.mesh.`**`UnitIntervalMesh`**(*nx*)

Bases: *`Mesh`*

A mesh of the unit interval.

Parameters

`nx` – The number of cells.

class `fe_utils.mesh.`**`UnitSquareMesh`**(*nx*, *ny*)

Bases: *`Mesh`*

A triangulated *`Mesh`* of the unit square.

Parameters

- **`nx`** – The number of cells in the x direction.
- **`ny`** – The number of cells in the y direction.

10.6 fe_utils.quadrature module

class `fe_utils.quadrature.`**`QuadratureRule`**(*cell*, *degree*, *points*, *weights*)

Bases: `object`

A quadrature rule implementing integration of the reference cell provided.

Parameters

- **`cell`** – the *`ReferenceCell`* over which this quadrature rule is defined.
- **`degree`** – the *degree of precision* of this quadrature rule.

Points

a list of the position vectors of the quadrature points.

Weights

the corresponding vector of quadrature weights.

`cell`

The *`ReferenceCell`* over which this quadrature rule is defined.

`degree`

The degree of precision of the quadrature rule.

`integrate`(*function*)

Integrate the function provided using this quadrature rule.

Parameters

`function` – A Python function taking a position vector as its single argument and returning a scalar value.

The implementation of this method is left as an *exercise*.

`points`

Two dimensional array, the rows of which form the position vectors of the quadrature points.

`weights`

The corresponding array of quadrature weights.

`fe_utils.quadrature.`**`gauss_quadrature`**(*cell*, *degree*)

Return a Gauss-Legendre *`QuadratureRule`*.

Parameters

- **`cell`** – the *`ReferenceCell`* over which this quadrature rule is defined.
- **`degree`** – the *degree of precision* of this quadrature rule.

10.7 fe_utils.reference_elements module

`class fe_utils.reference_elements.ReferenceCell(vertices, topology, name)`

Bases: `object`

An object storing the geometry and topology of the reference cell.

Parameters

- **vertices** – a list of coordinate vectors corresponding to the coordinates of the vertices of the cell.
- **topology** – a dictionary of dictionaries such that topology[d][i] is the list of vertices incident to the *i*-th entity of dimension *d*.

dim

The geometric and topological dimension of the reference cell.

entity_counts

The number of entities of each dimension.

point_in_entity(*x*, *e*)

Return true if the point x lies on the entity e.

Parameters

- **x** – The coordinate vector of the point.
- **e** – The (d, i) pair describing the entity of dimension *d* and index *i*.

topology

The vertices making up each topological entity of the reference cell.

vertices

The list of coordinate veritices of the reference cell.

`fe_utils.reference_elements.ReferenceInterval = ReferenceInterval`

A *ReferenceCell* storing the geometry and topology of the interval [0, 1].

`fe_utils.reference_elements.ReferenceTriangle = ReferenceTriangle`

A *ReferenceCell* storing the geometry and topology of the triangle with vertices [[0., 0.], [1., 0.], [0., 1.]].

10.8 fe_utils.utils module

`fe_utils.utils.errornorm(f1, f2)`

Calculate the L^2 norm of the difference between f1 and f2.

10.9 Module contents

BIBLIOGRAPHY

[Cia02] Philippe G Ciarlet. *The finite element method for elliptic problems*. Elsevier, 2002. doi:10.1137/1.9780898719208.

[Deu11] Peter Deuflhard. *Newton Methods for Nonlinear Problems*. Springer, 2011. doi:10.1007/978-3-642-23899-4.

[Kir04] R.C. Kirby. Algorithm 839: fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software (TOMS)*, 30(4):502–516, 2004. doi:10.1145/1039813.1039820.

[Log09] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283 – 295, 2009. doi:10.1504/IJCSE.2009.029164.

[TH73] C. Taylor and P. Hood. A numerical solution of the navier-stokes equations using the finite element technique. *Computers & Fluids*, 1(1):73–100, 1973. URL: https://www.sciencedirect.com/science/article/pii/0045793073900273, doi:https://doi.org/10.1016/0045-7930(73)90027-3.

PYTHON MODULE INDEX

f

fe_utils, 126
fe_utils.finite_elements, 121
fe_utils.function_spaces, 122
fe_utils.mesh, 124
fe_utils.quadrature, 125
fe_utils.reference_elements, 126
fe_utils.scripts, 120
fe_utils.scripts.plot_function_space_nodes, 119
fe_utils.scripts.plot_interpolate_lagrange, 119
fe_utils.scripts.plot_lagrange_basis_functions, 119
fe_utils.scripts.plot_lagrange_points, 119
fe_utils.scripts.plot_mesh, 119
fe_utils.scripts.plot_sin_function, 119
fe_utils.solvers, 121
fe_utils.solvers.helmholtz, 120
fe_utils.solvers.mastery, 120
fe_utils.solvers.poisson, 120
fe_utils.utils, 126

INDEX

A

adjacency() (*fe_utils.mesh.Mesh method*), 124
assemble() (*in module fe_utils.solvers.helmholtz*), 120
assemble() (*in module fe_utils.solvers.poisson*), 120

B

boundary_nodes() (*in module fe_utils.solvers.poisson*), 120

C

cell (*fe_utils.finite_elements.FiniteElement attribute*), 121
cell (*fe_utils.mesh.Mesh attribute*), 124
cell (*fe_utils.quadrature.QuadratureRule attribute*), 125
cell_edges (*fe_utils.mesh.Mesh attribute*), 124
cell_nodes (*fe_utils.function_spaces.FunctionSpace attribute*), 123
cell_vertices (*fe_utils.mesh.Mesh attribute*), 124

D

degree (*fe_utils.finite_elements.FiniteElement attribute*), 121
degree (*fe_utils.quadrature.QuadratureRule attribute*), 125
dim (*fe_utils.mesh.Mesh attribute*), 124
dim (*fe_utils.reference_elements.ReferenceCell attribute*), 126

E

edge_vertices (*fe_utils.mesh.Mesh attribute*), 124
element (*fe_utils.function_spaces.FunctionSpace attribute*), 123
entity_counts (*fe_utils.mesh.Mesh attribute*), 124
entity_counts (*fe_utils.reference_elements.ReferenceCell attribute*), 126
entity_nodes (*fe_utils.finite_elements.FiniteElement attribute*), 121
errornorm() (*in module fe_utils.utils*), 126

F

fe_utils
    module, 126
fe_utils.finite_elements
    module, 121
fe_utils.function_spaces
    module, 122
fe_utils.mesh
    module, 124
fe_utils.quadrature
    module, 125
fe_utils.reference_elements
    module, 126
fe_utils.scripts
    module, 120
fe_utils.scripts.plot_function_space_nodes
    module, 119
fe_utils.scripts.plot_interpolate_lagrange
    module, 119
fe_utils.scripts.plot_lagrange_basis_functions
    module, 119
fe_utils.scripts.plot_lagrange_points
    module, 119
fe_utils.scripts.plot_mesh
    module, 119
fe_utils.scripts.plot_sin_function
    module, 119
fe_utils.solvers
    module, 121
fe_utils.solvers.helmholtz
    module, 120
fe_utils.solvers.mastery
    module, 120
fe_utils.solvers.poisson
    module, 120
fe_utils.utils
    module, 126
FiniteElement (*class in fe_utils.finite_elements*), 121
Function (*class in fe_utils.function_spaces*), 122
function_space (*fe_utils.function_spaces.Function attribute*), 122
FunctionSpace (*class in fe_utils.function_spaces*), 123

G

gauss_quadrature() (*in module fe_utils.quadrature*), 125

I

integrate() (*fe_utils.function_spaces.Function method*), 123

integrate() (*fe_utils.quadrature.QuadratureRule method*), 125
interpolate() (*fe_utils.finite_elements.FiniteElement method*), 121
interpolate() (*fe_utils.function_spaces.Function method*), 123

J

jacobian() (*fe_utils.mesh.Mesh method*), 124

L

lagrange_points() (*in module fe_utils.finite_elements*), 122
LagrangeElement (*class in fe_utils.finite_elements*), 122

M

Mesh (*class in fe_utils.mesh*), 124
mesh (*fe_utils.function_spaces.FunctionSpace attribute*), 123
module
    fe_utils, 126
    fe_utils.finite_elements, 121
    fe_utils.function_spaces, 122
    fe_utils.mesh, 124
    fe_utils.quadrature, 125
    fe_utils.reference_elements, 126
    fe_utils.scripts, 120
    fe_utils.scripts.plot_function_space_nodes, 119
    fe_utils.scripts.plot_interpolate_lagrange, 119
    fe_utils.scripts.plot_lagrange_basis_functions, 119
    fe_utils.scripts.plot_lagrange_points, 119
    fe_utils.scripts.plot_mesh, 119
    fe_utils.scripts.plot_sin_function, 119
    fe_utils.solvers, 121
    fe_utils.solvers.helmholtz, 120
    fe_utils.solvers.mastery, 120
    fe_utils.solvers.poisson, 120
    fe_utils.utils, 126

N

name (*fe_utils.function_spaces.Function attribute*), 123
node_count (*fe_utils.finite_elements.FiniteElement attribute*), 121
node_count (*fe_utils.function_spaces.FunctionSpace attribute*), 123
nodes (*fe_utils.finite_elements.FiniteElement attribute*), 121
nodes_per_entity (*fe_utils.finite_elements.FiniteElement attribute*), 121

P

plot() (*fe_utils.function_spaces.Function method*), 123
plot_function_space_nodes() (*in module fe_utils.scripts.plot_function_space_nodes*), 119
plot_interpolate_lagrange() (*in module fe_utils.scripts.plot_interpolate_lagrange*), 119
plot_lagrange_basis_functions() (*in module fe_utils.scripts.plot_lagrange_basis_functions*), 119
plot_lagrange_points() (*in module fe_utils.scripts.plot_lagrange_points*), 119
plot_mesh() (*in module fe_utils.scripts.plot_mesh*), 119
plot_sin_function() (*in module fe_utils.scripts.plot_sin_function*), 119
point_in_entity() (*fe_utils.reference_elements.ReferenceCell method*), 126
points (*fe_utils.quadrature.QuadratureRule attribute*), 125

Q

QuadratureRule (*class in fe_utils.quadrature*), 125

R

ReferenceCell (*class in fe_utils.reference_elements*), 126
ReferenceInterval (*in module fe_utils.reference_elements*), 126
ReferenceTriangle (*in module fe_utils.reference_elements*), 126

S

solve_helmholtz() (*in module fe_utils.solvers.helmholtz*), 120
solve_mastery() (*in module fe_utils.solvers.mastery*), 120
solve_poisson() (*in module fe_utils.solvers.poisson*), 120

T

tabulate() (*fe_utils.finite_elements.FiniteElement method*), 121
topology (*fe_utils.reference_elements.ReferenceCell attribute*), 126

U

UnitIntervalMesh (*class in fe_utils.mesh*), 124
UnitSquareMesh (*class in fe_utils.mesh*), 124

V

values (*fe_utils.function_spaces.Function attribute*), 123
vandermonde_matrix() (*in module fe_utils.finite_elements*), 122
vertex_coords (*fe_utils.mesh.Mesh attribute*), 124
vertices (*fe_utils.reference_elements.ReferenceCell attribute*), 126

W

weights (*fe_utils.quadrature.QuadratureRule attribute*), 125