FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

USING DEAL.II TO SOLVE PROBLEMS IN COMPUTATIONAL FLUID DYNAMICS

By

LUKAS BYSTRICKY

A Thesis submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Master of Science

2016

Lukas Bystricky defended this thesis on March 28th, 2016.
The members of the supervisory committee were:

John Burkardt

Research Associate

Janet Peterson

Professor Directing Thesis

Sachin Shanbhag

Professor Directing Thesis

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

TABLE OF CONTENTS

COMMENTS ON NOTATION

In this document I have tried to remain consistent with the following practices:

- scalars are italicized lower case letters
- vectors are bold faced italic lower case letters
- matrices (or tensors of rank greater than or equal to 2) are upper cased bold faced italic letters
- scalar components of vectors or tensors are italic, but not bold faced

For example:

- a is a scalar
- $\boldsymbol{a}$ is a vector, a_i is a component of $\boldsymbol{a}$
- $\boldsymbol{A}$ is a matrix (or higher order tensor), A_{ij} is a component of $\boldsymbol{A}$

Index notation is used only where necessary; in general the more compact vector notation is preferred. The following vector notations will be used:

- ∇ for the gradient
- $\nabla\cdot$ for the divergence
- $\nabla\times$ for the curl
- Δ for the Laplacian

The general domain Ω, with boundary Γ is a convex domain with Lipschitz continuous boundary. When taking integrals over Ω, only a single integral sign is used, regardless of dimension. Thus:

$$\int_\Omega \mathrm{d}\Omega = \begin{cases} \int_\Omega \mathrm{d}x & \text{in 1D} \\ \iint_\Omega \mathrm{d}x\mathrm{d}y & \text{in 2D} \\ \iiint_\Omega \mathrm{d}x\mathrm{d}y\mathrm{d}z & \text{in 3D} \end{cases}.$$

The number of integral signs actually needed will be clear based on context.

ABSTRACT

Finite element methods are a common tool to solve problems in computational fluid dynamics (CFD). This thesis explores the finite element package deal.ii and specific applications to incompressible CFD. Some notation and results from finite element theory are summarised, and a brief overview of some of the features of deal.ii is given. Following this, several CFD applications are presented, including the Stokes equations, the Navier-Stokes equations and the equations for Darcy flow in porous media. Comparison with benchmark problems are provided for the Stokes and Navier-Stokes equations and a case study looking at foam deformation is provided for Darcy flow. Code is provided where applicable.

CHAPTER 1

INTRODUCTION

Computational fluid dynamics (CFD) is a widely used technique to investigate a range of problems arising in physics, chemistry, biology and engineering. Of particular interest in this thesis is the investigation of incompressible flow. This thesis will look at three sets of equations: the Stokes equations for viscous slow moving flow, the more general Navier-Stokes equations, and the equations for Darcy flow in porous media.

There are many techniques for solving the partial differential equations arising in CFD. These include finite difference, finite volume, finite element and integral equations methods. This thesis focuses exclusively on the use of finite elements. The first chapter provides a brief introduction to finite elements, including the notation used and some basic results from finite element analysis. A one dimensional example is provided to help clarify some important concepts.

In order to solve finite element problems quickly and efficiently we turn to the C++ finite element package deal.ii. Deal.ii was developed in Germany, but is now maintained primarily at Texas A&M. It is currently one of the most widely used open source finite element libraries. The second chapter provides some details on deal.ii, including the different quadrature and mapping routines as well as the linear solvers available.

The third chapter deals with the incompressible Stokes equations for slow moving viscous fluids. The steady state governing equations are given and a finite element discretization is derived. Key results concerning the stability and the accuracy of the resulting approximation are presented, and the Taylor-Hood element pair is introduced. A convergence study and benchmark problems for different boundary conditions are given. Some code is provided to demonstrate how to solve these kind of problems in deal.ii. Next, the unsteady incompressible Stokes equations are presented and discretized using a first order time stepping scheme. Code and a convergence study are provided.

The fourth chapter introduces the incompressible Navier-Stokes equations. These are the most widely used equations in CFD. The steady governing equations are introduced as is the concept of the Reynolds number. The equations are linearized and a finite element discretization is presented.

A convergence study is performed, and benchmark problems are looked at. Code is provided where applicable. Next, the unsteady incompressible equations are presented and discretized with a first order time stepping scheme. Again convergence studies and benchmark problems are investigated.

The fifth and final chapter looks at the equations for Darcy flow in porous media. The governing equations are presented, and a finite element discretization is provided. This finite element discretization necessitates the introduction of a new element, the Raviart-Thomas ($\mathcal{RT}$) element. Some results concerning the RT element are given and a convergence study provided. The final section deals with a case study which combines Hooke's law and the Darcy equations to investigate the effect of fluid flow on foam deformation.

CHAPTER 2

THE FINITE ELEMENT METHOD

Finite element methods (FEM) are a common technique used to numerically solve partial differential equations (PDEs). There are many desirable features provided by FEM. Solving PDEs on arbitrary domains is made relatively simple, as is increasing the accuracy of the approximation using higher order elements. The implementation of Neumann or mixed boundary conditions becomes very natural an easy to implement. In addition, the underlying theory behind finite elements is robust and well understood. Because of this, FEM is used in a variety of fields from solid mechanics to electromagnetics to fluid dynamics.

This chapter introduces some basic concepts from finite element analysis, including the definitions of various function spaces and norms we will be using throughout the rest of the document. A one dimensional example is provided to illustrate many of the key parts of FEM.

2.1 Function Spaces and Norms

In order to discuss finite elements, we must first define the appropriate function spaces, norms and inner products. The specific class of function spaces we will be interested in are called Hilbert spaces. Hilbert spaces are defined as complete vector spaces whose norm is derived from an inner product. The relationship between the norm and inner product allows us to compute errors in terms of projections. The completeness requirement means that any sequence of approximate solutions that converge will in fact converge to a unique element in the space. For more details on Hilbert spaces, including the precise definition of completeness, see [3].

The first such space, denoted $L^2(\Omega)$, is the space of all square integrable functions over Ω, i.e.:

$$L^2(\Omega) = \{q : \int_\Omega q^2 \mathrm{d}\Omega < \infty\}.$$

This space comes equipped with the norm and inner product:

$$(p, q) = \int_\Omega pq \mathrm{d}\Omega \qquad ||q||_0 = (q, q)^{1/2}.$$

To simplify notation in the definition of the remaining spaces, we will rewrite the partial differential operator using a multi-index. Let $\alpha = (\alpha_1, \alpha_2. \cdots, \alpha_n)$, where α_i, $i = 1, \cdots, n$ is a non-negative integer. Then the partial differential operator can be expressed as:

$$D^\alpha = \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \cdots \partial {x_n}^{\alpha_n}}.$$

For example in $\mathbb{R}^2$, $\alpha = (\alpha_1, \alpha_2)$, so for $|\alpha| = 2$ we have the following possibilities:

$$D^{(2,0)} = \frac{\partial^2}{\partial x_1^2} \qquad D^{(1,1)} = \frac{\partial^2}{\partial x_1 \partial x_2} \qquad D^{(0,2)} = \frac{\partial^2}{\partial x_2^2}$$

The space of all continuous functions on Ω is denoted as $C^0(\Omega)$. For non-negative integers k we can define the space $C^k(\Omega)$ as:

$$C^k(\Omega) = \{q : D^\alpha q \in C^0(\Omega) \text{ for } |\alpha| \leq k\},$$

That is, $C^k(\Omega)$ is the space of all functions that have all derivatives of up to and including order k continuous over Ω. In addition we denote by $C_0^k(\Omega)$ functions that are in $C^k(\Omega)$ and 0 on the boundary.

Before defining the remaining spaces, we must provide the definition of a weak derivative. A function $q \in L^2(\Omega)$ in $\mathbb{R}^n$ has a weak $L^2(\Omega)$ derivative of order α if there exists a function $v \in L^2(\Omega)$ such that:

$$\int_\Omega q D^\alpha \phi \mathrm{d}\Omega = (-1)^{|\alpha|} \int_\Omega v \phi \mathrm{d}\Omega,$$

for all $\phi \in C_0^\infty(\Omega)$.

We can now define the Sobolev spaces, defined for any non-negative integer k as:

$$H^k(\Omega) = \{q \in L^2(\Omega) : D^\alpha q \in L^2(\Omega) \quad \text{for } |\alpha| \leq k\}.$$

Thus a function in $H^k(\Omega)$ will be k times weakly differentiable. The space $H^k(\Omega)$ comes equipped with the norm:

$$||q||_k = \left(||q||_0 + \sum_{|\alpha| \leq k} ||D^\alpha q||_0^2 \right)^{1/2}.$$

A specific Sobolev space that we will be using often throughout the remainder of this thesis is the space $H^1(\Omega)$ and the subspace $H_0^1(\Omega)$ defined by:

$$H_0^1(\Omega) = \{q \in H^1(\Omega) : q = 0 \text{ on } \Gamma\}.$$

On the domain $\Omega \subset \mathbb{R}^n$, Both these spaces have the norm:

$$||q||_1 = \left(||q||_0^2 + \sum_{i=1}^{n} \left|\left| \frac{\partial q}{\partial x_i} \right|\right|_0^2 \right)^{1/2},$$

and the subspace $H_0^1(\Omega)$ has an equivalent semi-norm:

$$|q|_1 = \left(\sum_{i=1}^{n} \left|\left| \frac{\partial q}{\partial x_i} \right|\right|_0^2 \right)^{1/2}.$$

The space of all bounded linear functionals on $H_0^1(\Omega)$ is defined as $H^{-1}(\Omega)$:

$$H^{-1}(\Omega) = \{q : (q, v) < \infty \text{ for all } v \in H_0^1(\Omega)\},$$

while the space of all functions in $H^1(\Omega)$ restricted to the boundary Γ are denoted $H^{1/2}(\Gamma)$.

For vector valued functions with n components we will use the spaces:

$$\mathbf{L}^2(\Omega) = \{\boldsymbol{v} : v_i \in L^2(\Omega) \text{ for } i = 1, \cdots, n\}$$
$$\mathbf{C}^k(\Omega) = \{\boldsymbol{v} : v_i \in C^k(\Omega) \text{ for } i = 1, \cdots, n\}$$
$$\mathbf{H}^k(\Omega) = \{\boldsymbol{v} : v_i \in H^k(\Omega) \text{ for } i = 1, \cdots, n\}$$
$$\mathbf{H}_0^1(\Omega) = \{\boldsymbol{v} : v_i \in H_0^1(\Omega) \text{ for } i = 1, \cdots, n\}$$
$$\mathbf{H}^{-1}(\Omega) = \{\boldsymbol{v} : v_i \in H^{-1}(\Omega) \text{ for } i = 1, \cdots, n\}$$

and

$$\mathbf{H}^{1/2}(\Gamma) = \{\boldsymbol{v} : v_i \in H^{1/2}(\Gamma) \quad \text{for } i = 1, \cdots, n\}.$$

For $k \geq 0$, $\mathbf{H}^k(\Omega)$ comes equipped with the norm:

$$||\boldsymbol{v}||_k = \left(\sum_{i=1}^{n} ||v_i||_k^2 \right)^{1/2},$$

with the equivalent semi-norm for $\mathbf{H}_0^1(\Omega)$:

$$|\boldsymbol{v}|_1 = \left(\sum_{i=1}^{n} |v_i|_1^2 \right)^{1/2}.$$

The inner product for functions in $\mathbf{L}^2(\Omega)$ is defined as:

$$(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{u} \cdot \boldsymbol{v} \mathrm{d}\Omega.$$

Another Sobolev space, which is defined only for vector valued functions, is $\mathbf{H}_{\text{div}}(\Omega)$, defined as:

$$\mathbf{H}_{\text{div}}(\Omega) = \{\boldsymbol{v} \in \mathbf{L}^2(\Omega) : \nabla \cdot \boldsymbol{v} \in L^2(\Omega)\}.$$

This space comes equipped with the norm:

$$||\boldsymbol{v}||_{\text{div}} = \left(||\boldsymbol{v}||_0^2 + ||\nabla \cdot \boldsymbol{v}||_0^2\right)^{1/2},$$

and the equivalent semi-norm:

$$|\boldsymbol{v}|_{\text{div}} = ||\nabla \cdot \boldsymbol{v}||_0.$$

The final function spaces we will make use of are polynomial spaces $P_m(\Omega)$ and $Q_m(\Omega)$. Over a domain Ω, $P_m(\Omega)$ is defined as the space of all polynomials of total degree m or less and $Q_m(\Omega)$ is defined as all polynomials of degree m or less in each coordinate direction. The vector valued spaces $\mathbf{P}_m(\Omega)$ and $\mathbf{Q}_m(\Omega)$ are defined as expected:

$$\mathbf{P}_m(\Omega) = \{\boldsymbol{v} : v_i \in P_m(\Omega) \text{ for } i = 1, \cdots, n\}$$
$$\mathbf{Q}_m(\Omega) = \{\boldsymbol{v} : v_i \in Q_m(\Omega) \text{ for } i = 1, \cdots, n\}.$$

2.2 Weak Formulation

In order to put a given PDE into the finite element framework, we must first put it into its weak form. To do this, we take the L^2 inner product of our PDE with a test function v over the domain Ω. We then use integration by parts, or in dimensions higher than one, Green's identity, to balance the derivatives between v and the objective function in the PDE. In this section we will define the abstract weak problem an provide conditions which garantee a unique solution. We conclude by demonstrating how to determine the weak formulation of the 1D Poisson equation.

We wish to state the problem as follows. For a Hilbert space V, find $u \in V$ such that:

$$A(u, v) = F(v) \qquad \text{for all } v \in V. \tag{2.1}$$

where $A(\cdot, \cdot)$ is a bilinear form $V \times V \to \mathbb{R}^1$ and $F(\cdot)$ is a linear functional $V \to \mathbb{R}^1$.

The Lax-Miligram theorem states that (2.1) has a unique solution if the following three conditions apply:

1. A is bounded on V, i.e.:

$$|A(u,v)| \leq M||u||||v|| \qquad \text{for all } u,v \in V,$$

for some positive constant M.

2. A is coercive on V, i.e.:

$$A(u,u) \geq m||u||^2 \qquad \text{for all } u \in V,$$

for some positive constant m.

3. F is bounded on V, i.e.:

$$\sup_{v \in V} \frac{|F(v)|}{||v||} < \infty, \quad v \neq 0.$$

In general V is an infinite dimensional vector space. In order to compute an approximate solution, we choose $V^h \subset V$ to be a finite dimensional vector space. This means we can form a basis for V^h, $\{\phi_i\}$, $i = 1, \cdots, N$, and write a discrete problem as: Seek $u^h \in V^h$ such that:

$$A(u^h, v^h) = F(v^h) \quad \text{for all } v^h \in V^h \tag{2.2}$$

Since $u^h \in V^h$, it can be written as a linear combination of the basis functions:

$$u = \sum_{j=1}^{N} c_j \phi_j. \tag{2.3}$$

Testing against any $v^h \in V^h$ is equivalent to testing against every basis function of V^h. This allows us to rewrite (2.2) as:

$$A\left(\sum_{j=1}^{n} c_j \phi_j, \phi_i\right) = F(\phi_i) \quad i = 1, \cdots, N,$$

which is equivalent to the linear system:

$$\boldsymbol{Ac} = \boldsymbol{f},$$

with $\boldsymbol{A}_{ij} = A(\phi_j, \phi_i)$ and $\boldsymbol{f}_i = F(\phi_i)$. This linear system can be solved for the coefficients $\boldsymbol{c} = c_j$, $j = 1, \cdots, N$ in (2.3).

2.2.1 Example: Poisson Equation

As an example, consider the Poisson equation in 1D:

$$-u'' = f(x) \qquad 0 \leq x \leq 1, \tag{2.4}$$

along with boundary conditions:

$$u(0) = u(1) = 0. \tag{2.5}$$

Taking the L^2 inner product of (2.4) with a test function $v \in L^2([0,1])$ over the interval $[0,1]$ gives:

$$-(u'', v) = (f(x), v).$$

Applying integration by parts on the left hand side:

$$-(u'', v) = -\int_0^1 u''v\mathrm{d}x = -u'v\Big|_0^1 + \int_0^1 u'v'\mathrm{d}x = (f, v)$$

If we enforce $v(0) = v(1) = 0$, then the boundary term vanishes and we can write:

$$(u', v') = (f, v). \tag{2.6}$$

Note now that u and v are both required to have one derivative. Furthermore we have that u and v are both zero on the endpoints of our domain. This means that u and v are both in $H_0^1([0,1])$. Since v is an arbitrary function, we can state our weak problem as: Find $u \in H_0^1([0,1])$ such that:

$$A(u, v) = F(v) \qquad \text{for all } v \in H_0^1([0,1]),$$

where $A(u, v)$ is the bilinear form (u', v') and $F(v)$ is the linear functional (f, v).

As stated, v lives in an infinite dimensional space. In order to compute a solution, we look at V^h, a finite dimensional subspace of $H_0^1([0,1])$. We then require that (2.6) hold for functions belonging to V^h. The discrete weak problem is then: Find $u^h \in V^h$ such that:

$$A(u^h, v^h) = F(v^h) \qquad \text{for all } v^h \in V^h. \tag{2.7}$$

We can show by using the Lax-Milgram theorem that (2.7) has a unique solution $u^h \in V^h$ for any $V^h \in H_0^1([0,1])$. If we let $\{\phi_i\}$, $i = 1, \cdots, N$ be a basis for V^h, we can let $u = \sum_{j=1}^{N} c_j\phi_j$ and we can rewrite (2.7) as the linear system:

$$\boldsymbol{Ac} = \boldsymbol{f},$$

where:

$$\boldsymbol{A}_{ij} = (\phi_j', \phi_i')$$
$$\boldsymbol{f}_i = (f, \phi_i),$$

which we can solve for the coefficients c_j, $j = 1, \cdots, N$.

2.3 Finite Element Spaces

We still haven't specified a choice of V^h. For finite elements the typical choice is a space of piecewise continuous polynomials. This basis will be chosen to have compact support. This is done to ensure a sparse matrix $\boldsymbol{A}$.

Let $\mathcal{J}_h$ be a mesh of $\bar{\Omega}$, where $\bar{\Omega}$ is an approximation to Ω. In order to fully specify the finite element space over an element of $\mathcal{J}_h$ we must provide 3 pieces of information:

1. the geometric element
2. the degree of the polynomial within any element
3. the degrees of freedom used to determine the polynomial

One popular choice are the Lagrangian basis functions. These degrees of freedom for these elements are the function values at the nodes. To define a Lagrangian polynomial of degree k on quadrilateral requires $k+1$ nodes in 1D, and in general $(k+1)^d$ nodes in dimension d. Typically we choose the nodes to be equally spaced, although this is not necessary. See figure 2.1 for a typical placement of nodes for a bilinear and biquadratic element on a square.

On each element there will be a local basis function associated with each node. The local basis function i will be 1 at local node i, and 0 at all other nodes. To enforce continuity of the global basis functions, we will patch together local basis functions from neighbouring elements so that they match values at nodes which are common to more than 1 element.

Figure 2.1: Locations of degrees of freedom for Lagrangian elements. Left: bilinear element, right : biquadratic element.

If we denote by h the diameter of the largest element in $\mathcal{J}_h$ and assume our true solution $u(\boldsymbol{x})$ is in $H^{k+1}(\Omega)$, then using Lagrangian elements of order k, the following error estimates hold:

$$||u - u^h||_0 \leq C_1 h^{k+1} ||u||_{k+1}$$

and

$$||u - u^h||_1 \leq C_2 h^k ||u||_{k+1}$$

for some constants C_1 and C_2.

2.3.1 Example : Lagrangian Elements in 1D

Returning to our Poisson equation example, we can split the interval $[0, 1]$ into $N = 4$ equal intervals, $[0, 0.25]$, $[0.25, 0.5]$, $[0.5, 0.75]$ and $[0.75, 1]$. If we use quadratic elements, we will need 3 points in each element. We will take these to be the 2 endpoints and the midpoint of each element.

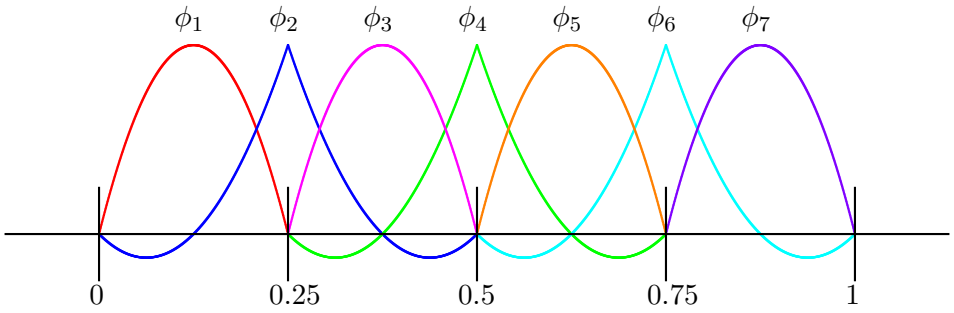

Figure 2.2: Quadratic Lagrange basis functions on the interval $[0, 1]$ using a uniform partition with $h = 1/4$. Note that since we have homogeneous Dirichlet boundary conditions we do not need basis functions at the endpoints of the domain.

It should be clear from figure 2.2 that since only a few of the basis functions have overlapping non-zero portions (i.e. they all have compact support), $\int_0^1 \phi_i' \phi_j' \mathrm{d}x = 0$ for $|i - j| > 2$. This means that our matrix $\boldsymbol{A}$ has entries:

$$\boldsymbol{A}_{ij} = \begin{cases} \int_0^1 \phi_j' \phi_i' \mathrm{d}x & \text{if } |i - j| \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

In other words, $\boldsymbol{A}$ is a sparse banded matrix, in this case with bandwidth 5. Sparse, structured matrices are in general much cheaper to store and much cheaper to solve than dense matrices. This demonstrates one of the nice features of finite elements and the importance of compact support.

2.4 Implementation

Efficiently assembling the matrix $\boldsymbol{A}$ and right hand side $\boldsymbol{f}$ is a key part of any finite element solver. The best way to do this is often to integrate over each element of $\bar{\Omega}$ individually and add the contributions to the appropriate entries in $\boldsymbol{A}$ and $\boldsymbol{f}$. These integrals (especially those in $\boldsymbol{f}$) will in general have to be computed numerically using quadrature. An efficient algorithm therefore might look like:

- loop over all elements $\mathcal{E}$
- loop over all quadrature points $\boldsymbol{x}_q$ in $\mathcal{E}$, with associated weight w_q
- loop over all basis functions ϕ_i nonzero over $\mathcal{E}$
- $\boldsymbol{f}_i = \boldsymbol{f}_i + F(\phi_i(\boldsymbol{x}_q))w_q$
- loop over all basis functions ϕ_i nonzero over $\mathcal{E}$
- $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ij} + A(\phi_j(\boldsymbol{x}_q), \phi_i(\boldsymbol{x}_q))w_q$

This implementation assumes that we know the quadrature points and weights as well as the definition of the basis function on each element. In practice however, as we'll see, this works best only for simple elements on structured meshes. In general we will calculate the integrals on a reference element (unit line, square or cube) and then map the integral to the actual element.

2.5 2d Problems

The transition from 1d to 2d problems is straightforward. The biggest difference is in generating the weak formulation. Consider the Poisson equation in 2d with homogeneous Dirichlet boundary condition:

$$-\Delta u = f(\boldsymbol{x}) \text{ in } \Omega \tag{2.8a}$$

$$u = 0 \text{ on } \Gamma. \tag{2.8b}$$

Taking the L^2 inner product of (2.8a) with a test function $v \in H_0^1(\Omega)$ we get:

$$-(\Delta u, v) = (f, v).$$

We can apply the product rule to show that:

$$-(\Delta u, v) = -\int_\Omega \Delta uv \mathrm{d}\Omega = -\int_\Omega \nabla \cdot (\nabla uv)\mathrm{d}\Omega + \int_\Omega \nabla u \cdot \nabla v \mathrm{d}\Omega.$$

From the divergence theorem we know that:

$$\int_\Omega \nabla \cdot (\nabla uv)\mathrm{d}\Omega = \oint_\Gamma v\nabla u \cdot \boldsymbol{n}\mathrm{d}\Gamma.$$

Since $v \in H_0^1(\Omega)$ it is 0 on the boundary and this term is therefore 0. This leads to our final Galerkin weak formulation:

$$A(u, v) = (\nabla u, \nabla v) = F(v)$$

The identity

$$\int_\Omega \Delta uv \mathrm{d}\Omega = \oint_\Gamma v\nabla u \cdot \boldsymbol{n}\mathrm{d}\Gamma - \int_\Omega \nabla u \cdot \nabla v \mathrm{d}\Omega \tag{2.9}$$

is known as Green's identity and is used often as the equivalent to integration by parts in higher dimensions.

The remaining steps, writing the discrete weak formulation and assembling and solving the resulting linear system are exactly the same as for the 1D case.

CHAPTER 3

DEAL.II

Deal.ii is an open source C++ finite element package [2]. It is the successor to DEAL (Differential Equations Analysis Library). The ease of adaptive meshing and the speed of the solvers provided by deal.ii as well as the flexibility of C++ are all useful tools for many finite element computations.

Deal.ii originated at the University of Heidelberg in Germany. Today it is maintained primarily at Texas A&M University, although it has contributors throughout the world. It is one of the most widely used open source finite element packages. The use of C++ templates allows problems in 1, 2 or 3 dimensions to be solved with minimal changes to the code. Deal.ii only allows quadrilateral elements in 2D or hexahedral elements in 3D.

3.1 Meshing

Meshing can be done on various simple objects in 2 or 3D directly in deal.ii using the **GridGenerator** class. They can also be loaded in from external programs like Gmsh, Lagrit and Cubit. Once a mesh is created or loaded it is stored in the **Triangulation** class, where it can be refined or otherwise modified as needed. All the meshes on non-rectangular domains later in this thesis were created in Gmsh.

3.2 Quadrature

To calculate the integrals arising in the matrices, quadrature must be used. Any integral over a domain Ω can be approximated by a summation:

$$\int_{\Omega} f(\boldsymbol{x}) \mathrm{d}\Omega \approx \sum_{i=1}^{n} f(\boldsymbol{q}_i) w_i$$

The quadrature points $\{\boldsymbol{q}_i\}$ and weights $\{w_i\}$, $i = 1, \cdots, n$ depend upon the quadrature rule being used. Deal.ii provides many kinds of quadrature routines such as Gauss-Legendre, Gauss-Chebyshev, Gauss-Lobatto, Trapezoid, Milne and Simpson. The **Quadrature** class in deal.ii cre-

n	$\boldsymbol{q}_i$	w_i
1	0	2
2	$\pm\sqrt{1/3}$	1
3	0 $\pm\sqrt{3/5}$	$8/9$ $5/9$
4	$\pm\sqrt{3/7 - 2/7\sqrt{6/5}}$ $\pm\sqrt{3/7 + 2/7\sqrt{6/5}}$	$\frac{18+\sqrt{30}}{36}$ $\frac{18-\sqrt{30}}{36}$
5	0 $\pm\frac{1}{3}\sqrt{5 - 2\sqrt{10/7}}$ $\pm\frac{1}{3}\sqrt{5 + 2\sqrt{10/7}}$	$128/225$ $\frac{322+13\sqrt{70}}{900}$ $\frac{322-13\sqrt{70}}{900}$

Table 3.1: First 5 Gaussian quadrature rules on the interval $[-1, 1]$.

ates and stores quadrature points and weights on the unit line $[0, 1]$, the unit square $[0, 1] \times [0, 1]$ or the unit cube $[0, 1] \times [0, 1] \times [0, 1]$.

The most popular choice for quadrature is Gauss-Legendre. This rule is designed to integrate polynomials exactly using the fewest number of points. Specifically in 1D a Gauss-Legendre rule using n points can integrate any polynomial of degree $2n - 1$ exactly. The first 5 rules over the interval $[-1, 1]$ are given in table 3.1. Deal.ii maps these to the unit line. Higher dimensional quadrature rules are created by taking tensor products of 1D quadrature rules.

3.3 Mappings

When assembling the finite element matrices we will have to compute integrals of the form:

$$\int_{\mathcal{Q}} \omega(\boldsymbol{x}) \mathrm{d}\mathcal{Q} \tag{3.1}$$

where $\mathcal{Q}$ is an arbitrary quadrilateral in 2D or a hexahedron in 3D. Restricting ourselves to 2D, the quadrature rules stored in the **Quadrature** class are only valid for integrals over the unit box. The integral (3.1) can be calculated over the unit box $\hat{\mathcal{Q}}$:

$$\int_{\mathcal{Q}} \omega(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \int_{\hat{\mathcal{Q}}} \omega(F(\hat{\boldsymbol{x}})) |\det \boldsymbol{J}(\hat{\boldsymbol{x}})| \mathrm{d}\hat{\mathcal{Q}}$$

where $\boldsymbol{x} = F(\hat{\boldsymbol{x}})$ for some mapping $F : \hat{\mathcal{Q}} \to \mathbb{R}^2$, and

$$\boldsymbol{J}(\hat{\boldsymbol{x}}) = \hat{\nabla} F = \begin{pmatrix} \partial x/\partial \hat{x} & \partial x/\partial \hat{y} \\ \partial y/\partial \hat{x} & \partial y/\partial \hat{y} \end{pmatrix}$$

is the Jacobian of the transformation. The mappings described below are available in the deal.ii **Mapping** class.

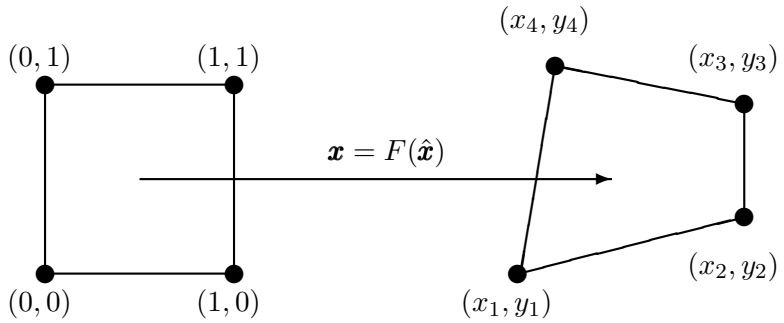

Figure 3.1: A mapping from the reference square.

3.3.1 Bilinear mapping

A bilinear mapping is a mapping of the form:

$$F(\hat{x}, \hat{y}) = \begin{pmatrix} A & B & C \\ E & G & H \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{x}\hat{y} \end{pmatrix} + \begin{pmatrix} D \\ K \end{pmatrix}.$$

Since we want $(0, 0)$ to map to (x_1, y_1), $(1, 0)$ to map to (x_2, y_2) and so on, we can create a linear system to solve for the coefficients:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ G \\ H \\ K \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

Bilinear mappings map straight lines in the unit square to straight lines in the actual quadrilateral. They also preserve things like the midpoints of the edges and the centre of mass of the element.To evaluate basis functions on $\hat{\mathcal{K}}$, we can simply evaluate the corresponding basis function on the reference square. In other words $\phi(\boldsymbol{x}) = \hat{\phi}(\hat{\boldsymbol{x}})$. To evaluate the gradient of the basis functions we must use the chain rule:

$$\frac{\partial \phi}{\partial \hat{x}_i} = \frac{\partial \hat{\phi}}{\partial \hat{x}_i} = \frac{\partial \hat{\phi}}{\partial x}\frac{\partial x}{\partial \hat{x}_i} + \frac{\partial \hat{\phi}}{\partial y}\frac{\partial y}{\partial \hat{x}_i}$$

This leads to teh following relationship:

$$\begin{pmatrix} \partial\phi/\partial x \\ \partial\phi/\partial y \end{pmatrix} = (\boldsymbol{J}^T)^{-1} \begin{pmatrix} \partial\hat{\phi}/\partial\hat{x} \\ \partial\hat{\phi}/\partial\hat{y} \end{pmatrix}$$

3.3.2 Piola transformation

For vector valued functions in $\mathbf{H}_{\text{div}}(\Omega)$ we need the mapping to preserve the normal components of the vector, since for certain elements (e.g. Raviart-Thomas elements) the normal component of the basis function may be a degree of freedom. In general a bilinear mapping does not do this. A mapping that does this is called the Piola transformation. Given an affine mapping $F : \hat{\mathcal{Q}} \to \mathbb{R}^2$, for example the bilinear mapping discussed in the previous section, and a function $\hat{\mathbf{u}}(\hat{\boldsymbol{x}})$ defined on the unit square, then we can define the Piola transformation as:

$$\boldsymbol{u}(x) = \frac{1}{\det \boldsymbol{J}(\hat{\boldsymbol{x}})}\boldsymbol{J}(\hat{\boldsymbol{x}})\hat{\boldsymbol{u}}(\hat{\boldsymbol{x}})$$

where $\boldsymbol{J}$ is the Jacobian of F.

For $\hat{u} \in \mathbf{H}_{\text{div}}(\hat{\mathcal{Q}})$ and $\hat{p} \in \mathbf{H}^1(\hat{\mathcal{Q}})$ this transformation has the properties [1]:

$$\int_{\mathcal{Q}} \boldsymbol{u} \cdot \nabla p \mathrm{d}\mathcal{Q} = \int_{\hat{\mathcal{Q}}} \hat{\boldsymbol{u}} \cdot \hat{\nabla}\hat{p}\mathrm{d}\hat{\mathcal{Q}}$$
$$\int_{\mathcal{Q}} \nabla \cdot \boldsymbol{u} p \mathrm{d}\mathcal{Q} = \int_{\hat{\mathcal{Q}}} \hat{\nabla} \cdot \hat{\boldsymbol{u}}\hat{p}\mathrm{d}\hat{\mathcal{Q}}$$
$$\int_{\partial\mathcal{K}} \boldsymbol{u} \cdot \boldsymbol{n} p \mathrm{d}s = \int_{\partial\hat{\mathcal{K}}} \hat{\boldsymbol{u}} \cdot \hat{\mathbf{n}}\mathrm{d}\hat{s}$$

3.4 Linear Solvers

The deal.ii **Solver** class supports a wide array of linear solvers including:

- Conjugate gradient
- Biconjugate gradient stabilized method

- Generalized minimal residual method
- Richardson
- Sparse direct solve

3.5 Adaptive Meshing

In order to achieve a desired accuracy when solving real world problems it often necessary to reduce the mesh size. Rather than uniformly refining the mesh, we can instead refine only where the solution is changing most rapidly. This helps to gain a more accurate solution while limiting the increase in the number of unknowns. One way to quantify how much a solution is changing is to consider for each element $\mathcal{Q}$, with edges e_1, e_2, e_3 and e_4, the quantity

$$\eta_Q^2 = \frac{h_Q}{2p} \sum_{i=1}^{4} \int_{e_i} \left[a \frac{\partial w^h}{\partial \boldsymbol{n}_i} \right] \mathrm{d}e_i,$$

where:

- $\left[\quad \cdot \quad \right]$ denotes the jump across e_i
- w^h is the computed solution
- h_Q is the largest diagonal of the element (the element diameter)
- p is the degree of the approximating polynomial
- a is a scaling coefficient

This is called the Kelly error estimate [6] and it measures the jump in the gradient of the solution across each element. Deal.ii implements this in the **KellyErrorEstimator** class. After computing this for each cell we can then refine a percentage of cells with the highest η_Q.

3.6 Graphical Output

The class **DataOut** can output matrices to various formats including:

- EPS
- VTK
- GNUPLOT

All visualizations of finite element solutions provided later in this thesis have been created using the VTK format in VisIt.

CHAPTER 4

STOKES FLOW

The Stokes equations describe the motion of a creeping incompressible Newtonian fluid. They are used extensively in applications involving slow moving viscous fluids, as they are a linear system on PDEs and are generally much easier to solve than the full nonlinear Navier-Stokes equations. This chapter will introduce the steady and unsteady Stokes equations and their finite element discretization. To show existence and uniqueness we will review a well known result from the theory of mixed FEM called the Ladyzhenskaya-Babuska-Brezzi condition and introduce the Taylor-Hood element pair, which is a key element in CFD and one we will use in this chapter and also when working with the Navier-Stokes equations. Convergence studies and comparisons to benchmark problems will be provided throughout for a variety of boundary conditions.

4.1 Steady Governing Equations

Inside a domain Ω with boundary Γ, the non-dimensional steady state Stokes equations are given by:

$$-\nu\Delta\boldsymbol{u} + \nabla p = \boldsymbol{f}(\boldsymbol{x}) \tag{4.1a}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{4.1b}$$

where:

- $\boldsymbol{u}$ is the velocity of the fluid; in 2D $\boldsymbol{u} = \langle u, v \rangle$
- p is the fluid pressure
- $\boldsymbol{f}(\boldsymbol{x}) \in \mathbf{H}^{-1}(\Omega)$ is a known forcing function
- ν is the (constant) kinematic viscosity of the fluid

Initially we will assume homogeneous Dirichlet boundary conditions on the velocity, i.e. $\boldsymbol{u} = \mathbf{0}$ on Γ.

4.2 Weak Formulation

4.2.1 Galerkin Weak Formulation

We begin by taking the $\mathbf{L}^2$ inner product of (4.1a) with a vector valued function $\boldsymbol{v}(\boldsymbol{x}) \in \mathbf{H}_0^1(\Omega)$:

$$-\nu\left(\Delta\boldsymbol{u}, \boldsymbol{v}\right) + \left(\nabla p, \boldsymbol{v}\right) = \left(\boldsymbol{f}, \boldsymbol{v}\right). \tag{4.2}$$

We can use Green's identity, 2.9, on the first term to obtain (using index notation):

$$\begin{aligned}\int_\Omega \Delta\boldsymbol{u}\cdot\boldsymbol{v}\mathrm{d}\Omega &= \int_\Omega \frac{\partial}{\partial x_i}\left(\frac{\partial u_j}{\partial x_i}\right)\omega_j \mathrm{d}\Omega \\ &= \oint_\Gamma \frac{\partial u_j}{\partial x_i} n_i \omega_j \mathrm{d}\Gamma - \int_\Omega \frac{\partial u_j}{\partial x_i}\frac{\partial \omega_j}{\partial x_i}\mathrm{d}\Omega = \oint_\Gamma \left(\nabla\boldsymbol{u}\cdot\boldsymbol{n}\right)\cdot\boldsymbol{v}\mathrm{d}\Gamma - \int_\Omega \nabla\boldsymbol{u} : \nabla\boldsymbol{v}\mathrm{d}\Omega.\end{aligned}$$

Applying the same technique on the $(\nabla p, \boldsymbol{v})$ term:

$$\begin{aligned}\int_\Omega \nabla p\cdot\boldsymbol{v}\mathrm{d}\Omega &= \int_\Omega \frac{\partial p}{\partial x_i}\omega_i \mathrm{d}\Omega \\ &= \oint_\Gamma p\omega_i n_i \mathrm{d}\Gamma - \int_\Omega p\frac{\partial \omega_i}{\partial x_i}\mathrm{d}\Omega = \oint_\Gamma p\boldsymbol{v}\cdot\boldsymbol{n}\mathrm{d}\Gamma - \int_\Omega p\nabla\cdot\boldsymbol{v}\mathrm{d}\Omega.\end{aligned}$$

Plugging these expressions back into (4.2) yields:

$$\nu\left(\nabla\boldsymbol{u}, \nabla\boldsymbol{v}\right) - \left(p, \nabla\cdot\boldsymbol{v}\right) = \left(\boldsymbol{f}, \boldsymbol{v}\right) + \oint_\Gamma \left(\left(\nabla\boldsymbol{u}\cdot\boldsymbol{n}\right)\cdot\boldsymbol{v} - p\boldsymbol{v}\cdot\boldsymbol{n}\right)\mathrm{d}\Gamma.$$

The boundary integral term can be rewritten:

$$\oint_\Gamma \left(\nabla\boldsymbol{u}\cdot\boldsymbol{n}\right)\cdot\boldsymbol{v} - p\boldsymbol{v}\cdot\boldsymbol{n}\mathrm{d}\Gamma = \oint_\Gamma \left(\boldsymbol{n}\cdot\nabla\boldsymbol{u}\cdot\boldsymbol{n} - p\right)\boldsymbol{v}\cdot\boldsymbol{n}\mathrm{d}\Gamma. \tag{4.3}$$

If we choose $\boldsymbol{v} \in \mathbf{H}_0^1(\Omega)$, then this boundary integral is equal to 0.

The weak form of the second equation can be found by multiplying (4.1b) by a scalar test function $q(\boldsymbol{x}) \in L^2(\Omega)$:

$$-\left(q, \nabla\cdot\boldsymbol{u}\right) = 0.$$

Then the final weak form of the Stokes equations can be written as:

$$a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = \left(\boldsymbol{f}, \boldsymbol{v}\right) \qquad \text{for all } \boldsymbol{v} \in \mathbf{H}_0^1(\Omega) \tag{4.4a}$$

$$b(\boldsymbol{u}, q) = 0 \qquad \text{for all } q \in L^2(\Omega) \tag{4.4b}$$

where:

- $a(\boldsymbol{u}, \boldsymbol{v}) = \nu\,(\nabla \boldsymbol{u}, \nabla \boldsymbol{v})$
- $b(\boldsymbol{v}, q) = -(q, \nabla \cdot \boldsymbol{v})$

Note that in the weak form we only require the gradient of $\boldsymbol{u}$ to exist, and we have no differentiability requirements on p. Thus we are looking for functions $\boldsymbol{u}$, p such that $\boldsymbol{u} \in \mathbf{H}_0^1(\Omega)$ and $p \in L^2(\Omega)$.

4.2.2 Discrete Weak Formulation

In order to compute a solution we will have to restrict $\boldsymbol{u}$ and p to finite dimensional spaces. Thus we will be looking for approximations to $\boldsymbol{u}$ and p, $\boldsymbol{u}^h$ and p^h, such that $\boldsymbol{u}^h \in \mathbf{V}^h \subset \mathbf{H}_0^1(\Omega)$ and $p^h \in S^h \subset L^2(\Omega)$. Our discrete weak problem reads as follows: seek $\boldsymbol{u}^h \in \mathbf{V}^h$ and $p^h \in S^h$ such that:

$$a(\boldsymbol{u}^h, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^h) = (\boldsymbol{f}, \boldsymbol{v}^h) \qquad \text{for all } \boldsymbol{v}^h \in \mathbf{V}^h \tag{4.5a}$$

$$b(\boldsymbol{u}^h, q^h) = 0 \qquad \text{for all } q^h \in S^h \tag{4.5b}$$

The next step is to choose a basis for $\mathbf{V}^h$ and S^h. If we let $\{\boldsymbol{v}_k(\boldsymbol{x})\}$, $k = 1, \cdots, K$, be a basis for $\mathbf{V}^h$ and $\{q_j(\boldsymbol{x})\}$, $j = 1, \cdots, J$, be a basis for S^h, we can express our solutions $\boldsymbol{u}^h$ and p^h as a linear combination of these basis functions:

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{k=1}^{K} \alpha_k \boldsymbol{v}_k(\boldsymbol{x}) \qquad p^h(\boldsymbol{x}) = \sum_{j=1}^{J} \beta_j q_j(\boldsymbol{x})$$

Testing against all functions $\boldsymbol{v}^h \in \mathbf{V}^h$ or $q^h \in S^h$ is equivalent to testing against the basis for that space, so (4.5) can be rewritten as:

$$\sum_{k=1}^{K} \alpha_k a(\boldsymbol{v}_k, \boldsymbol{v}_\ell) + \sum_{j=1}^{J} \beta_j b(q_j, \boldsymbol{v}_\ell) = (\boldsymbol{f}, \boldsymbol{v}_\ell) \qquad \text{for } \ell = 1, \cdots, K \tag{4.6a}$$

$$\sum_{k=1}^{K} \alpha_k b(\boldsymbol{v}_k, q_i) = 0 \qquad \text{for } i = 1, \cdots, J \tag{4.6b}$$

which is a linear system of size $(K + J) \times (K + J)$ which can be solved for the unknown coefficients $\{\alpha_k\}$ and $\{\beta_j\}$. This linear system is symmetric, but it turns out that it it not positive definite. This effects our choice of linear solver. For example we cannot use conjugate gradient. Instead we will use a sparse direct solver.

One important point to note here is that this system as written will be singular. This is due to the fact that the pressure only appears as a gradient and without any boundary conditions. This

means that if we find a solution p that satisfies (4.1a), then $p + C$ where C is a constant would also satisfy it. There are several ways around this; in our setup we will simply set the pressure at some particular point in Ω to be 0.

4.3 Finite Element Space

In order for the discrete weak problem (4.5) to have a unique solution we can no longer use the Lax-Milgram theorem described in section 2.2. Since $\boldsymbol{u}^h, \boldsymbol{v}^h \in \mathbf{V}_0^h \subset \mathbf{H}_0^1(\Omega)$ and $p^h, q^h \in S^h \subset L^2(\Omega)$, the finite element method applied to the Stokes equations is known as a mixed finite element method. In order to have a unique solution to the discrete weak problem we must have that:

1. $(\boldsymbol{f}, \boldsymbol{v}^h)$ is bounded for all $\boldsymbol{v}^h \in \mathbf{V}^h$
2. $a(\boldsymbol{u}^h, \boldsymbol{v}^h)$ is bounded and coercive for all $\boldsymbol{u}^h, \boldsymbol{v}^h \in \mathbf{V}^h$
3. $b(\boldsymbol{v}^h, q^h)$ must be bounded for all $\boldsymbol{v}^h \in \mathbf{V}^h$ and $q^h \in S^h$
4. $b(\boldsymbol{v}^h, q^h)$ must satisfy:

$$\inf_{\substack{q^h \in S^h \\ q^h \neq 0}} \sup_{\substack{\boldsymbol{v}^h \in \mathbf{V}^h \\ \boldsymbol{v}^h \neq 0}} \left(\frac{b(\boldsymbol{v}^h, q^h)}{||\boldsymbol{v}^h||_V ||q^h||_S} \right) \geq m. \tag{4.7}$$

It turns out that the first three of these conditions are automatically true for any choice of $\mathbf{V}^h$ and S^h. The fourth conition is called the Ladyzhenskaya-Babuska-Brezzi (LBB) or inf-sup condition. In mixed finite elements this replaces the coercivity requirement in the Lax-Milgram theorem. See [5] for a more detailed discussion of this condition and techniques to verify that it is satisfied.

We can use the bilinear form $b(\cdot, \cdot)$ to define the subspace $\mathbf{Z}$ consisting of weakly divergence free functions:

$$\mathbf{Z} = \{\boldsymbol{v} \in \mathbf{H}_0^1(\Omega) : b(\boldsymbol{v}, q) = 0 \text{ for all } q \in L^2(\Omega)\}.$$

Likewise we can define the space consisting of discretely weak divergence free functions, $\mathbf{Z}^h$:

$$\mathbf{Z} = \{\boldsymbol{v}^h \in \mathbf{V}^h : b(\boldsymbol{v}^h, q) = 0 \text{ for all } q^h \in S^h\}.$$

Note that in general $\mathbf{Z}^h \not\subset \mathbf{Z}$, meaning that discretely divergence free functions are not actually divergence free. A measure of the angle between the spaces $\mathbf{Z}^h$ and $\mathbf{Z}$ is given by:

$$\Theta = \sup_{\substack{\boldsymbol{z}^h \in \mathbf{Z}^h \\ |\boldsymbol{z}^h|_1 = 1}} \inf_{\boldsymbol{z} \in \mathbf{Z}} |\boldsymbol{z} - \boldsymbol{z}^h|_1.$$

Provided (4.7) is satisfied, then we can derive the following error estimates:

$$|\boldsymbol{u} - \boldsymbol{u}^h|_1 \leq C_1 \inf_{\boldsymbol{v}^h \in \mathbf{V}^h} |\boldsymbol{u} - \boldsymbol{v}^h|_1 + C_2 \Theta \inf_{q^h \in S^h} ||p - q^h||_0 \tag{4.8a}$$

$$||p - p^h||_0 \leq C_3 \inf_{\boldsymbol{v}^h \in \mathbf{V}^h} |\boldsymbol{u} - \boldsymbol{v}^h|_1 + C_4 \inf_{q^h \in S^h} ||p - q^h||_0 \tag{4.8b}$$

where C_i, $i = 1, \cdots, 4$, are constants independent of h. It is efficient to make the rates of convergence of the two terms on the right hand sides equal. Note the error estimates depend on the $\mathbf{H}^1$ seminorm for the velocity and the L^2 norm for the pressure. In practice this means that if we are using polynomial approximating functions, then we should take one power higher for the velocity than the pressure.

4.3.1 The Taylor-Hood Element Pair

One choice of $\boldsymbol{V}^h$ and S^h which satisfy (4.7) for the Stokes equations is the Taylor-Hood element pair. Let $\mathcal{J}_h$ be a quadrilateral mesh of $\bar{\Omega}$ where $\bar{\Omega}$ is an approximation to Ω. Then the Taylor-Hood element pair is defined as:

$$\mathbf{V}^h = \{\boldsymbol{v} : \boldsymbol{v} \in \mathbf{Q}_2(\square), \square \in \mathcal{J}_h, \quad \boldsymbol{v} \in \mathbf{C}^0(\bar{\Omega})\}$$
$$S^h = \{q : q \in Q_1(\square), \square \in \mathcal{J}_h, \quad q \in C^0(\bar{\Omega})\}$$

In other words we choose $\boldsymbol{V}^h$ to be the space of all piecewise continuous polynomials of degree 2 in each component, and S^h to be the space of all piecewise continuous bilinear polynomials. The degrees of freedom for the velocity space are the function values at corners of the quadrilaterals, as well as the midpoints of each edge and the midpoint of the element. The degrees of freedom for the pressure space are just the function values at the corners of each quadrilateral. Assuming $\boldsymbol{u} \in \mathbf{H}^3(\Omega)$ and $p \in H^2(\Omega)$, we get the following error estimates for this element pair:

$$|\boldsymbol{u} - \boldsymbol{u}^h|_1 = O(h^2) \tag{4.9a}$$

$$||\boldsymbol{u} - \boldsymbol{u}^h||_0 = O(h^3) \tag{4.9b}$$

$$||p - p^h||_0 = O(h^2) \tag{4.9c}$$

where h is the diameter of the largest element in $\mathcal{J}_h$. Note that both $|\boldsymbol{u} - \boldsymbol{u}^h|_1$ and $||p - p^h||_0$ have the same order of convergence which is what we wanted from (4.8).

4.4 Inhomogeneous Velocity Boundary Conditions

So far we have only considered homogeneous Dirichlet boundary conditions on $\boldsymbol{u}$. Suppose instead we consider the boundary condition:

$$\boldsymbol{u} = \boldsymbol{g} \text{ on } \Gamma \text{ with } \int_\Gamma \boldsymbol{g} \cdot \boldsymbol{n} \mathrm{d}\Gamma = 0, \tag{4.10}$$

where $\boldsymbol{g}(\boldsymbol{x}) \in \mathbf{H}^{1/2}(\Gamma)$. The compatibility condition on $\boldsymbol{g} \cdot \boldsymbol{n}$ is needed to ensure global conservation of mass. Define the set:

$$\mathbf{V}_g = \{\boldsymbol{u} \in \mathbf{H}^1(\Omega) : \boldsymbol{u} = \boldsymbol{g} \text{ on } \Gamma, \quad \boldsymbol{q} \in \mathbf{H}^{1/2}(\Gamma)\}.$$

In order to solve our discrete weak problem (4.5) we must define a space $\mathbf{V}^h$ for our velocity. Functions that will belong to $\mathbf{V}^h$ will be piecewise polynomials and thus will not in general satisfy the boundary condition (4.10). Therefore we choose an interpolant $\boldsymbol{g}^h$ which is in the space $\mathbf{V}^h$ and restricted to the boundary Γ.

If we let $\mathbf{V}^h$ be a Lagrangian finite element space, then the degrees of freedom are exclusively values at points. Let $\{\boldsymbol{v}_k\}$, $k = 1, \cdots, K$ be a basis for V^h, with the first K_1 of these basis functions being associated with interior nodes, i.e. $\boldsymbol{v}_k = 0$ on Γ for $k = 1, \cdots, K_1$. The remaining basis functions are associated with nodes on Γ. This allows us to write:

$$\boldsymbol{g}^h(\boldsymbol{x}) = \sum_{k=K_1+1}^{K} \tilde{\alpha}_k \boldsymbol{v}_k(\boldsymbol{x}),$$

where $\tilde{\alpha}$ is the appropriate component of g evaluated at node $\boldsymbol{x}_k$. Thus if $\boldsymbol{u}^h \in \mathbf{V}^h$ satisfies $\boldsymbol{u}^h(\boldsymbol{x}) = \boldsymbol{g}^h(\boldsymbol{x})$ for $\boldsymbol{x} \in \Gamma$, we can write:

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{k=1}^{K_1} \alpha_k \boldsymbol{v}_k + \sum_{k=K_1+1}^{K} \tilde{\alpha}_k \boldsymbol{v}_k(\boldsymbol{x}).$$

Note that the only unknowns here are the α_k, $k = 1, \cdots, K_1$. The second summation becomes part of the data of the discrete system.

If we define the set:

$$\mathbf{V}_g^h = \{\boldsymbol{v} \in \mathbf{V}^h : \boldsymbol{v} = \boldsymbol{g}^h \text{ on } \Gamma\},$$

then we can write our weak problem as: Seek $\boldsymbol{u}^h \in \boldsymbol{V}_g^h$ and $p^h \in S^h$, such that (4.5) holds for all $\boldsymbol{v}^h \in \mathbf{V}_0$ and $q^h \in S^h$. It turns out that all error estimates, in particular those for the Taylor-Hood element pair given in section 4.3.1 are still valid for this problem.

The linear system is now of size $K \times K$, whereas if the boundary conditions were homogeneous the system would be of size $K_1 \times K_1$. However since $K - K_1$ of the coefficients are already known from the boundary data, we can zero out those rows and simply set $\tilde{\alpha}_k$ to be the appropriate component of $\boldsymbol{g}$ evaluated at $\boldsymbol{x}_k$.

4.5 Implementation

Consider the discrete weak formulation of the Stokes equations given by:

$$a(\boldsymbol{u}^h, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^h) = (\boldsymbol{f}, \boldsymbol{v}^h) \qquad \text{for all } \boldsymbol{v}^h \in \mathbf{V}_0^h$$

$$b(\boldsymbol{u}^h, q^h) = 0 \qquad \text{for all } q^h \in S^h$$

along with the boundary conditions $\boldsymbol{u}^h = \boldsymbol{g}^h$ on Γ.

We start by providing the definitions neccessary for our **StokesSolver** class:

```
template<int dim>
class StokesSolver
{
    public:
        StokesSolver();
        void run();

        Function<dim> *forcing_function;
        Function<dim> *boundary_values;

    private:
        void setup_geometry(int cycle);
        void assemble_system();
        void solve();

        Triangulation<dim>     mesh;
        FESystem<dim>          fe;
        DoFHandler<dim>        dof_handler;

        ConstraintMatrix       constraints;
        SparsityPattern        sparsity_pattern;
        SparseMatrix<double>   system_matrix;
        Vector<double>         system_rhs;
        Vector<double>         solution;
};
```

```
template<int dim>
StokesSolver<dim>::StokesSolver():
    fe(FE_Q<dim>(2), dim, FE_Q<dim>(1), 1),
    dof_handler(mesh)
{}
```

Here, $<dim>$ is the dimension of the problem, 2 for all cases we consider. The initialization of *fe* is therefore asking for 2 quadratic Lagrange elements and 1 linear Lagrange element.

```
template<int dim>
void StokesSolver<dim>::setup_geometry (int cycle)
{
    GridIn<dim> grid_in;
    grid_in.attach_triangulation(mesh);
    std::ifstream input_file(input.gmesh_file.c_str());
    grid_in.read_msh(input_file);

    dof_handler.distribute_dofs(fe);

    FEValuesExtractors::Vector velocities(0);
    FEValuesExtractors::Scalar pressure(dim);

    typename DoFHandler<dim>::active_cell_iterator
            cell = dof_handler.begin_active(), endc = dof_handler.end();

    std::vector<bool> boundary_dofs(dof_handler.n_dofs(), false);

    constraints.clear();

    //set Dirichlet boundary conditions on the velocity everywhere
    VectorTools::interpolate_boundary_values(dof_handler, 0,
                *boundary_values, constraints, fe.component_mask(velocities));

    //constrain first pressure dof to be 0
    DoFTools::extract_boundary_dofs(dof_handler, fe.component_mask(pressure),
                    boundary_dofs);

    const unsigned int first_boundary_dof = std::distance(boundary_dofs.begin(),
            std::find (boundary_dofs.begin(), boundary_dofs.end(), true));

    constraints.add_line(first_boundary_dof);
    constraints.close();

    CompressedSparsityPattern c_sparsity(dof_handler.n_dofs(), dof_handler.n_dofs());
    DoFTools::make_sparsity_pattern (dof_handler, c_sparsity);
```

```
    constraints.condense(c_sparsity);
    sparsity_pattern.copy_from(c_sparsity);

    system_matrix.reinit(sparsity_pattern);
    solution.reinit(dof_handler.n_dofs());
    system_rhs.reinit(dof_handler.n_dofs());
}
```

To assemble the matrix, we follow the procedure outlined in section 2.4. The deal.ii class **DofHandler** keeps track of the degrees of freedom of our problem. We can ask the **DofHandler** object for an iterator object to each cell in our mesh. On each cell we evaluate the forcing function, as well as $\boldsymbol{v}_k$, $\nabla \boldsymbol{v}_k$, $\nabla \cdot \boldsymbol{v}_k$ and q_k for each of the basis functions active over the cell at each quadrature point. The assembly routine in deal.ii looks like:

```
template <int dim>
void StokesSolver<dim>::assemble_system()
{
    QGauss<dim>                  quadrature_formula(fe.degree + 1);

    const int                    dofs_per_cell = fe.dofs_per_cell;
    const int                    n_q_points = quadrature_formula.size();

    std::vector<Vector<double> > rhs_values (n_q_points, Vector<double>(dim+1));
    std::vector<Tensor<2,dim> > grad_phi_u (dofs_per_cell);
    std::vector<double>          div_phi_u (dofs_per_cell);
    std::vector<double>          phi_p (dofs_per_cell);
    std::vector<Tensor<1,dim> > phi_u (dofs_per_cell);

    Vector<double>               cell_rhs(dofs_per_cell);
    FullMatrix<double>           cell_matrix (dofs_per_cell, dofs_per_cell);

    std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
    const FEValuesExtractors::Vector velocities (0);
    const FEValuesExtractors::Scalar pressure (dim);

    FEValues<dim> fe_values(fe, quadrature_formula, update_values | update_gradients
                                |update_JxW_values | update_quadrature_points);

    typename DoFHandler<dim>::active_cell_iterator
            cell = dof_handler.begin_active(), endc = dof_handler.end();

    for (; cell!=endc; ++cell)
    {
```

```
        fe_values.reinit(cell);
        cell_matrix = 0;
        cell_rhs = 0;

        forcing_function->vector_value_list(fe_values.get_quadrature_points(),
                                            rhs_values);

        //calculate cell contribution to system
        for (int q = 0; q < n_q_points; q++)
        {
            for (int k=0; k<dofs_per_cell; k++)
            {
                grad_phi_u[k] = fe_values[velocities].gradient (k, q);
                div_phi_u[k]  = fe_values[velocities].divergence (k, q);
                phi_p[k]      = fe_values[pressure].value (k, q);
                phi_u[k]      = fe_values[velocities].value (k, q);
            }

            for (int i = 0; i < dofs_per_cell; i++)
            {
                for (int j = 0; j < dofs_per_cell; j++)
                {
                    cell_matrix(i,j) +=
                        (input.mu*double_contract(grad_phi_u[i],grad_phi_u[j])
                                - phi_p[i]*div_phi_u[j]
                                - phi_p[j]*div_phi_u[i])
                                *fe_values.JxW(q);
                }

                int equation_i = fe.system_to_component_index(i).first;
                cell_rhs[i] += fe_values.shape_value(i,q)
                             *rhs_values[q](equation_i)*fe_values.JxW(q);
            }
        }

        cell->get_dof_indices(local_dof_indices);
        constraints.distribute_local_to_global(cell_matrix, cell_rhs,
                                local_dof_indices, system_matrix, system_rhs);
    }
}
```

To solve our system, we use one of the solvers from UMFPACK. UMFPACK provides a set of routines for solving sparse linear systems. We will use a direct method.

```
template<int dim>
```

```
void StokesSolver<dim>::solve()
{
    SparseDirectUMFPACK A_direct;
    A_direct.initialize(system_matrix);

    A_direct.vmult(solution, system_rhs);
    constraints.distribute(solution);
}
```

4.6 Convergence Study

To test our code we can use the method of manufactured solutions. Let:

$$\boldsymbol{u}(x,y) = \begin{pmatrix} \cos(\pi x) \\ y\pi \sin(\pi x) \end{pmatrix} \tag{4.11}$$

$$p(x,y) = (xy)^2 \tag{4.12}$$

Note that $\nabla \cdot \boldsymbol{u} = 0$. We can use these solutions to generate a forcing function $\boldsymbol{f}(x,y)$ and boundary conditions on the unit square. In deal.ii we can define these functions as:

```
template<int dim>
class ExactSolutionBoundaryValues : public Function<dim>
{
    public:
        ExactSolutionBoundaryValues() : Function<dim>(3) {}
        virtual void vector_value(const Point<dim> &p,
                Vector<double> &values) const;
};

template<int dim>
void ExactSolutionBoundaryValues<dim>::vector_value(const Point<dim> &p,
                Vector<double> &values) const
{
    double x = p[0];
    double y = p[1];

    values(0) = cos(M_PI*x);
    values(1) = y*M_PI*sin(M_PI*x);
    values(2) = 0;
 }

template<int dim>
class ExactSolutionForcingFunction : public Function<dim>
```

```
{
    public:
        ExactSolutionForcingFunction() : Function<dim>(3) {};
        virtual void vector_value(const Point<dim> &p,
                Vector<double> &values) const;
};

template<int dim>
void ExactSolutionForcingFunction<dim>::vector_value(const Point<dim> &p,
                Vector<double> &values) const
{
    double x = p[0];
    double y = p[1];

    values(0) = cos(M_PI*x)*pow(M_PI,2) + 2*x*pow(y,2);
    values(1) = sin(M_PI*x)*y*pow(M_PI,3) + 2*y*pow(x,2));
    values(2) = 0;
}
```

By uniformly refining the mesh we can test the convergence of our code and verify we get the theoretical rates. Doing so, we can generate the convergence tables 4.1 and 4.2. These tables demonstrate the theoretical rates predicted by (4.9).

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	1.140×10^{-2}	3.421×10^{-3}	-	1.054×10^{-1}	-
64	659	1.498×10^{-3}	4.270×10^{-4}	3.00	2.642×10^{-2}	2.00
256	2467	1.895×10^{-4}	5.335×10^{-5}	3.00	6.609×10^{-3}	2.00

Table 4.1: Convergence rates for $\boldsymbol{u}(\boldsymbol{x})$ in the steady Stokes equations using Taylor-Hood elements.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	2.386×10^{-1}	1.171×10^{-1}	-	2.386×10^{-1}	-
64	659	5.811×10^{-2}	4.699×10^{-2}	1.87	6.459×10^{-2}	1.63
256	2467	1.500×10^{-2}	1.234×10^{-2}	1.93	1.179×10^{-2}	1.33

Table 4.2: Convergence rates for $p(\boldsymbol{x})$ in the steady Stokes equations using Taylor-Hood elements.

4.7 Steady Lid Driven Cavity

A standard test problem in CFD is the lid driven cavity problem. In this problem we have fluid in a box of height H and width W with a lid moving to the right with velocity u_0. We

assume no-slip boundary conditions, so the fluid has velocity $\boldsymbol{u} = \mathbf{0}$ on the sides and bottom, and $\boldsymbol{u} = \begin{pmatrix} u_0 & 0 \end{pmatrix}^T$ on the top. See the setup in figure 4.1.

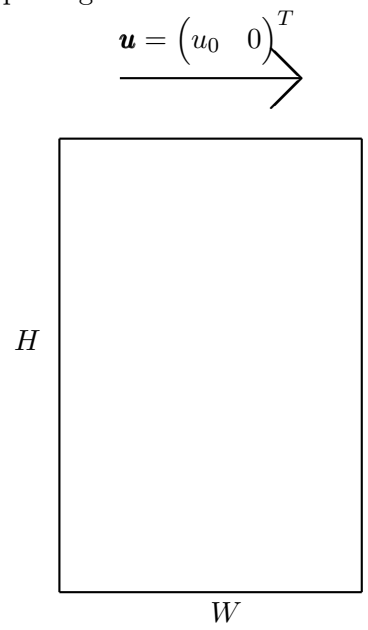

Figure 4.1: Setup of the lid driven cavity problem.

If we take $H = 5$, $W = 1$ and $u_0 = 1$, we can qualitatively recreate the streamlines from figure 3.8 in [9]. As shown in figure 4.2, we get 4 primary eddies running down the height of the box.

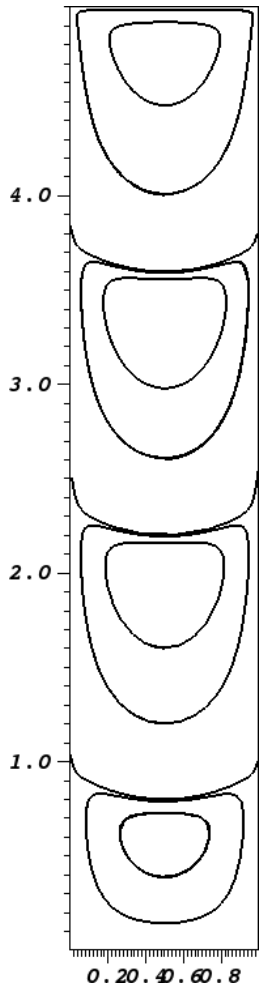

Figure 4.2: Streamlines for the lid driven cavity problem with $u_0 = 1$, $\nu = 0.01$, $H = 5$ and $W = 1$.

4.8 Alternate Boundary Conditions

Up until now we have assumed Dirichlet boundary conditions on the velocity. In many applications we may with to prescribe different boundary conditions. Looking at the boundary integral (4.3) from our weak formulation, we see that this is a known quantity if either $\boldsymbol{v} \cdot \boldsymbol{n} = 0$, as we have assumed up until now, or we prescribe the natural boundary condition $\boldsymbol{n} \cdot \nabla \boldsymbol{u} \cdot \boldsymbol{n} - p$. This section will explore other boundary conditions that can be applied, in particular the stress boundary condition. There are other possible boundary conditions not discussed here, see [5] for more details.

4.8.1 Addition to Weak Formulation

Let us begin by denoting two segments of the boundary Γ, Γ_n and Γ_τ. These segments may overlap, may be empty and their union need not cover the entire domain, see figure 4.3 for an example of the various partitions.

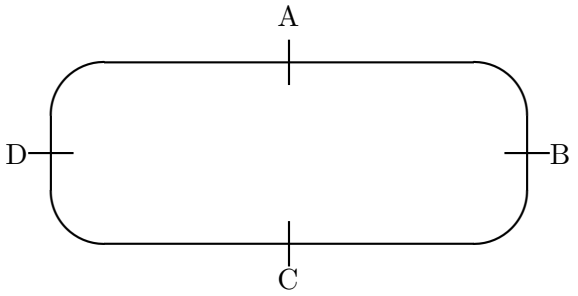

Figure 4.3: An example of the various possible domain segments: ABC $= \Gamma_n$, BCD $= \Gamma_\tau$, BC $= \Gamma_n \cap \Gamma_\tau$, AD $= \Gamma/(\Gamma_n \cup \Gamma_\tau)$.

Let us now define the spaces:

$$\mathbf{V}_g = \{\boldsymbol{v} \in \mathbf{H}^1(\Omega) : \boldsymbol{v} \cdot \boldsymbol{n} = g_n \text{ on } \Gamma_n \text{ and } \boldsymbol{n} \times \boldsymbol{v} \times \boldsymbol{n} = \boldsymbol{g}_\tau \text{ on } \Gamma_\tau\}$$
$$\mathbf{V}_0 = \{\boldsymbol{v} \in \mathbf{H}^1(\Omega) : \boldsymbol{v} \cdot \boldsymbol{n} = 0 \text{ on } \Gamma_n \text{ and } \boldsymbol{v} \times \boldsymbol{n} = \boldsymbol{0} \text{ on } \Gamma_\tau\}$$

Suppose that we wish to prescribe the normal velocity on Γ_n and the tangential velocity on Γ_τ. Then we have the boundary conditions:

$$\boldsymbol{u} \cdot \boldsymbol{n} = g_n \text{ on } \Gamma_n$$
$$\boldsymbol{n} \times \boldsymbol{u} \times \boldsymbol{n} = \boldsymbol{g}_\tau \text{ on } \Gamma_\tau$$

In other words $\boldsymbol{u} \in \mathbf{V}_g$. For $i = 1, 2$, we will be considering weak formulations of the form:

$$a_i(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{u}, p) = (\boldsymbol{f}, \boldsymbol{v}) + d_i(\boldsymbol{v}) \qquad \text{for all } \boldsymbol{v} \in \mathbf{V}_0$$
$$b(\boldsymbol{u}, q) = 0 \qquad \text{for all } q \in L^2(\Omega)$$

where $b(\cdot, \cdot)$ is defined as in the previous sections, but now $a_i(\cdot, \cdot)$ takes the form:

$$a_i(\boldsymbol{u}, \boldsymbol{v}) = \begin{cases} \nu \int_\Omega \nabla \boldsymbol{u} : \boldsymbol{v} \mathrm{d}\Omega & \text{for } i = 1 \\ \frac{1}{2} \int_\Omega \nu \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right) : \left(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T\right) \mathrm{d}\Omega & \text{for } i = 2 \end{cases}$$

Obviously $a_1(\cdot,\cdot)$ is the same as the $a(\cdot,\cdot)$ from the previous sections and comes from applying Green's identity to the Laplacian $\Delta\boldsymbol{u}$. The other bilinear form, $a_2(\cdot,\cdot)$ comes from applying Green's identity to the equivalent tensor $\nabla\cdot\left(\nu\left(\nabla\boldsymbol{u}+(\nabla\boldsymbol{u})^T\right)\right)$. The linear functional $d_i(\cdot)$ is a boundary integral arising from Green's identity. We define $d_i(\boldsymbol{v})$ as:

$$d_i(\boldsymbol{v}) = \int_{\Gamma/\Gamma_n} r_i\boldsymbol{v}\cdot\boldsymbol{n}\mathrm{d}\Gamma + \int_{\Gamma/\Gamma_\tau} \boldsymbol{s}_i\cdot\boldsymbol{v}\times\boldsymbol{n}\mathrm{d}\Gamma$$

where:

$$r_i = \begin{cases} -p+\nu\boldsymbol{n}\cdot\nabla\boldsymbol{u}\cdot\boldsymbol{n} & \text{for } i=1 \\ -p+\nu\boldsymbol{n}\cdot\left(\nabla\boldsymbol{u}+(\nabla\boldsymbol{u})^T\right)\cdot\boldsymbol{n} & \text{for } i=2 \end{cases}$$

$$\boldsymbol{s}_i = \begin{cases} \nu\boldsymbol{n}\cdot\nabla\boldsymbol{u}\times\boldsymbol{n} & \text{for } i=1 \\ \nu\boldsymbol{n}\cdot\left(\nabla\boldsymbol{u}+(\nabla\boldsymbol{u})^T\right)\times\boldsymbol{n} & \text{for } i=2 \end{cases}$$

If we specify the velocity $\boldsymbol{u}$, we are by definition specifying both the normal and tangential velocity. In this case the space $\mathbf{V}_0$ is the space $\mathbf{H}_0^1$, and so $\boldsymbol{v}\cdot\boldsymbol{n}=0$ and $\boldsymbol{v}\times\boldsymbol{n}=\mathbf{0}$. Therefore in order to evaluate $d_i(\boldsymbol{v})$ we must specify:

1. the velocity $\boldsymbol{u}$ on $\Gamma_n\cap\Gamma_\tau$
2. r_i and $\boldsymbol{s}_i$ on $\Gamma/(\Gamma_n\cup\Gamma_\tau)$
3. the normal velocity and $\boldsymbol{s}_i$ on $\Gamma_n/(\Gamma_n\cap\Gamma_\tau)$
4. the tangential velocity and $\boldsymbol{r}_i$ on $\Gamma_\tau/(\Gamma_n\cap\Gamma_\tau)$

For $i=1$, r_i and $\boldsymbol{s}_i$ have no physical meaning. For $i=2$ however, r_i is the normal stress, σ, and $\boldsymbol{s}_i$ is the tangential stress, τ. These are the boundary conditions we will use for slit flow.

4.8.2 Slit Flow

For a slit flow problem we will consider a rectangular domain of height h and length ℓ. On the top and bottom of this domain we will specify no-slip conditions on the velocity. In addition we will prescribe the normal stress on both the inlet (left side) and outlet (right side). On both the inlet and the outlet we will set the tangential velocity, v to be 0.

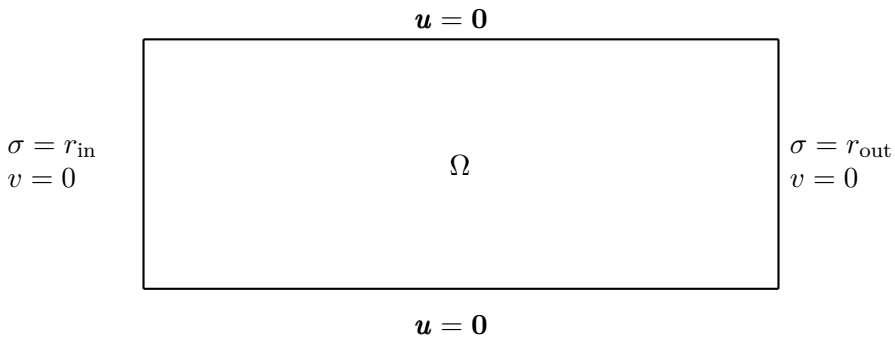

Figure 4.4: Sketch of the domain used for slit flow problems.

In 2D, the normal stress is given by:

$$\begin{aligned} r &= -p + \nu \boldsymbol{n} \cdot \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T \right) \cdot \boldsymbol{n} \\ &= -p + \nu \boldsymbol{n} \cdot \left(\begin{pmatrix} u_x & u_y \\ v_x & v_y \end{pmatrix} + \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix} \right) \cdot \boldsymbol{n} \\ &= -p + \nu \boldsymbol{n} \cdot \begin{pmatrix} 2u_x & u_y + v_x \\ u_y + v_x & 2v_y \end{pmatrix} \cdot \boldsymbol{n} \end{aligned}$$

For slit flow we know that on the inlet $\boldsymbol{n} = (-1, 0)^T$ and $\boldsymbol{n} = (1, 0)^T$ on the outlet. On both the inlet and the outlet $v_x = v_y = 0$, as shown in figure 4.4. Using this information we can calculate r_{in} and r_{out}:

$$r_{\text{in}} = r_{\text{out}} = -p + 2\nu u_x$$

Let $\boldsymbol{v} = (v_1, v_2)^T$, then we can calculate $d(\boldsymbol{v})$ on the inlet and outlet as:

$$\begin{aligned} d(\boldsymbol{v})_{\text{in}} &= \int_{\Gamma_{\text{in}}} r_{\text{in}} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} (-1, 0) \mathrm{d}\Gamma = -r_{\text{in}} \int_0^h v_1 \mathrm{d}y \\ d(\boldsymbol{v})_{\text{out}} &= \int_{\Gamma_{\text{out}}} r_{\text{out}} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} (1, 0) \mathrm{d}\Gamma = r_{\text{out}} \int_0^h v_1 \mathrm{d}y \end{aligned}$$

In this problem, since we are prescribing boundary conditions that involve the pressure, the pressure solution is unique and we do not have to modify our matrix to avoid it being singular.

Implementation. To implement slit flow in deal.ii, we have to modify a couple parts of the code. First, the boundary conditions on the inlet and outlet set only the v component of the velocity to 0. To do this we have to label the inlet and the outlet separately from the top and bottom. In the *setup_geometry* routine we must add the following code:

```
//set boundary labels:
// (1) : inlet
// (2) : outlet
// (3) : everything else
for (; cell!=endc; ++cell)
{
    for (int f=0; f<GeometryInfo<dim>::faces_per_cell; f++)
    {
        if (cell->face(f)->at_boundary())
        {
            if (fabs(cell->face(f)->center()[0]) < 1e-8)
            {
                cell->face(f)->set_boundary_indicator(1);
            }
            else if (fabs(cell->face(f)->center()[0] - input.domain_length) < 1e-8)
            {
                cell->face(f)->set_boundary_indicator(2);
            }
            else
            {
                cell->face(f)->set_boundary_indicator(3);
            }
        }
    }
}
```

To apply the boundary conditions on only the v component of the velocity, we can use a scalar **ComponentMask** on boundaries 1 and 2:

```
//apply velocity boundary conditions to top and bottom only
//set tangential (vertical) velocity at inlet and outlet
FEValuesExtractors::Scalar velocity_y(1);

VectorTools::interpolate_boundary_values(dof_handler, 3,
            *boundary_values, constraints, fe.component_mask(velocities));

VectorTools::interpolate_boundary_values(dof_handler, 1,
            *boundary_values, constraints, fe.component_mask(velocity_y));

VectorTools::interpolate_boundary_values(dof_handler, 2,
            *boundary_values, constraints, fe.component_mask(velocity_y));
```

The matrix assemble routine changes slightly. During the matrix assembly we will have to evaluate the line integral $d(\boldsymbol{v}^{\boldsymbol{h}})$. To do this we will need to specify a quadrature rule for the line

integral:

```
QGauss<dim-1>     face_quadrature_formula(3);
const int         n_q_points_face = face_quadrature_formula.size();
FEFaceValues<dim> fe_face_values (fe, face_quadrature_formula, update_values
        | update_quadrature_points | update_gradients | update_JxW_values);
```

The dense matrix for each cell, *cell_matrix*, is now populated by:

```
cell_matrix(i,j) +=
            (phi_u[i]*phi_u[j] + 0.5*input.mu*
            double_contract(grad_phi_u[i] + transpose(grad_phi_u[i]),
                            grad_phi_u[j] + transpose(grad_phi_u[j]))
            - phi_p[i]*div_phi_u[j] - phi_p[j]*div_phi_u[i])
            *fe_values.JxW(q);
```

Also inside the assembly routine for each cell we will have to check and see if it borders the inlet or the outlet, and if it does, use quadrature to evaluate the line integral $d(\boldsymbol{v}^{\boldsymbol{h}})$ on the portion of that cell that is on the inlet or outlet:

```
//add line integral term to rhs for inlet and outlet normal stress
for (unsigned int face=0; face < GeometryInfo<dim>::faces_per_cell; face++)
{
    if (cell->face(face)->at_boundary())
    {
        fe_face_values.reinit (cell, face);

        for (unsigned int q_boundary = 0; q_boundary < n_q_points_face;
                                                q_boundary++)
        {
            for (int k=0; k<dofs_per_cell; k++)
            {
                phi_u[k] = fe_face_values[velocities].value (k, q_boundary);
            }

            Point<dim> q_point = fe_face_values.quadrature_point(q_boundary);
            double x = q_point[0];

            for (int i = 0; i < dofs_per_cell; i++)
            {
                if (fabs(x) < 1e-8)
                {
                    cell_rhs[i] -= input.r_in
                            *phi_u[i][0]*fe_face_values.JxW(q_boundary);
                }
                else if (fabs(x - input.domain_length) < 1e-8)
```

```
                {
                    cell_rhs[i] += input.r_out
                            *phi_u[i][0]*fe_face_values.JxW(q_boundary);
                }
            }
        }
    }
}
```

4.8.3 Poiseuille flow

A simple test case for slit flow is pressure driven Poiseuille flow. For this flow we assume a linear pressure drop. We prescribe a pressure P_{in} at the inlet and P_{out} at the outlet. Since $v = 0$ at the inlet and outlet, we can assume that $v = 0$ everywhere in our domain. Then the continuity equation tells us that:

$$u_x + v_y = 0 \Rightarrow u_x = 0.$$

This leads us to the exact solutions:

$$\boldsymbol{u}(x, y) = \begin{pmatrix} -\frac{P_{\text{in}} - P_{\text{out}}}{2\ell}(y^2 - hy) \\ 0 \end{pmatrix}$$
$$p(x, y) = P_{\text{in}} - \frac{P_{\text{in}} - P_{\text{out}}}{\ell} x$$

Note that in this case, since $u_x = 0$, the normal stresses at the inlet and outlet are simply $-P_{\text{in}}$ and $-P_{\text{out}}$ respectively, The computed solution with $P_{\text{in}} = 1e5$ and $P_{\text{out}} = 0$ is shown in figure 4.5.

We can compare our computed solutions to the exact solution. Since $\boldsymbol{u}$ is a second order polynomial and p is a linear polynomial we expect our finite element solver using $\mathbf{Q}_2$ elements for the velocity and Q_1 elements for the pressure to be able to compute the solution exactly. If we look at the errors in tables 4.3 and 4.4, we see that all the error norms for both the pressure and the velocity are very small and independent of mesh resolution. This leads us to conclude that we are finding the true solution.

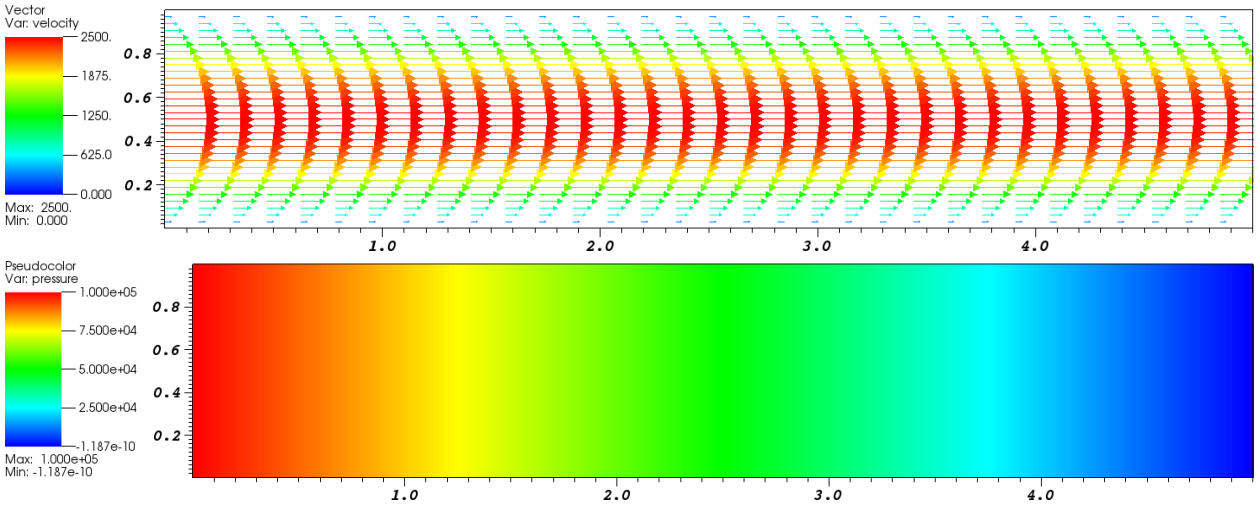

Figure 4.5: Steady Stokes Poiseuille flow with $P_{\text{in}} = 1 \times 10^5$ and $P_{\text{out}} = 0$. Top: velocity vector field, bottom: pressure.

h	# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{H}^1$ error
$1/4$	196	2032	1.156×10^{-15}	8.074×10^{-16}	1.437×10^{-14}
$1/8$	784	7589	2.194×10^{-15}	1.596×10^{-15}	3.744×10^{-14}
$1/16$	3136	29287	3.442×10^{-15}	1.685×10^{-15}	7.722×10^{-15}

Table 4.3: Errors for $\boldsymbol{u}(\boldsymbol{x})$ in steady Poiseuille flow.

h	# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{H}^1$ error
$1/4$	196	2032	2.220×10^{-14}	8.653×10^{-15}	9.032×10^{-14}
$1/8$	784	7589	7.550×10^{-14}	2.276×10^{-14}	4.297×10^{-13}
$1/16$	3136	29287	1.767×10^{-13}	3.519×10^{-14}	1.218×10^{-12}

Table 4.4: Errors for $p(\boldsymbol{x})$ in steady Poiseuille flow..

4.9 Adaptive Meshing

Recall that as long as $\boldsymbol{u}$ and p are sufficiently smooth, the Taylor-Hood element pair has order h^3 accuracy in $\boldsymbol{u}$ and h^2 accuracy in p in the L^2 norm. This means that uniformly halving the diameter of each element should reduce the error in $\boldsymbol{u}$ by a factor of 8 and the error in p by a factor of 4. In 2D, halving the diameter of each element means splitting it in 4. Each element has 4 pressure degrees of freedom and 18 velocity degrees of freedom. Of course a lot of them are shared between elements, but as is shown in the convergence table in section 4.5, the number of unknowns grows quite quickly under uniform refinement. To get better approximations without such a large increase in the number of unknowns we turn to the adaptive mesh refinement techniques described in section 3.5.

For fluid flow problems we may not be equally interested in the pressure and the velocity. In that case we can consider the Kelly error estimate for only the velocity for example, or we could choose a different scaling coefficient a for the pressure and the velocity. If are only interested in the velocity profile, the Kelley error estimate for just the velocity would be:

$$\eta_Q^2 = \frac{h_Q}{4} \sum_{i=1}^{4} \int_{e_i} \left[\frac{\partial u^h}{\partial \boldsymbol{n}_i}\right] + \left[\frac{\partial v^h}{\partial \boldsymbol{n}_i}\right] \mathrm{d}e_i. \tag{4.13}$$

Implementing this is deal.ii is quite simple. We define an additional function in the **Stokes-Solver** class, *refine_grid()*:

```
template<int dim>
void StokesSolver<dim>::refine_grid()
{
    const FEValuesExtractors::Vector velocities (0);

    Vector<float> estimated_error_per_cell (mesh.n_active_cells());
    KellyErrorEstimator<dim>::estimate (dof_handler, QGauss<1>(dim + 1),
            typename FunctionMap<dim>::type(), solution, estimated_error_per_cell,
            fe.component_mask(velocities));

    GridRefinement::refine_and_coarsen_fixed_fraction (mesh,
                    estimated_error_per_cell, 0.7, 0.1);

    mesh.prepare_coarsening_and_refinement();
    mesh.execute_coarsening_and_refinement ();
}
```

Figure 4.6 shows how adaptive mesh refinement for the lid driven cavity problem. Notice that it refines in the top corners where there exists singularities, but leaves the bottom unrefined. This refinement results in a much more appealing solution.

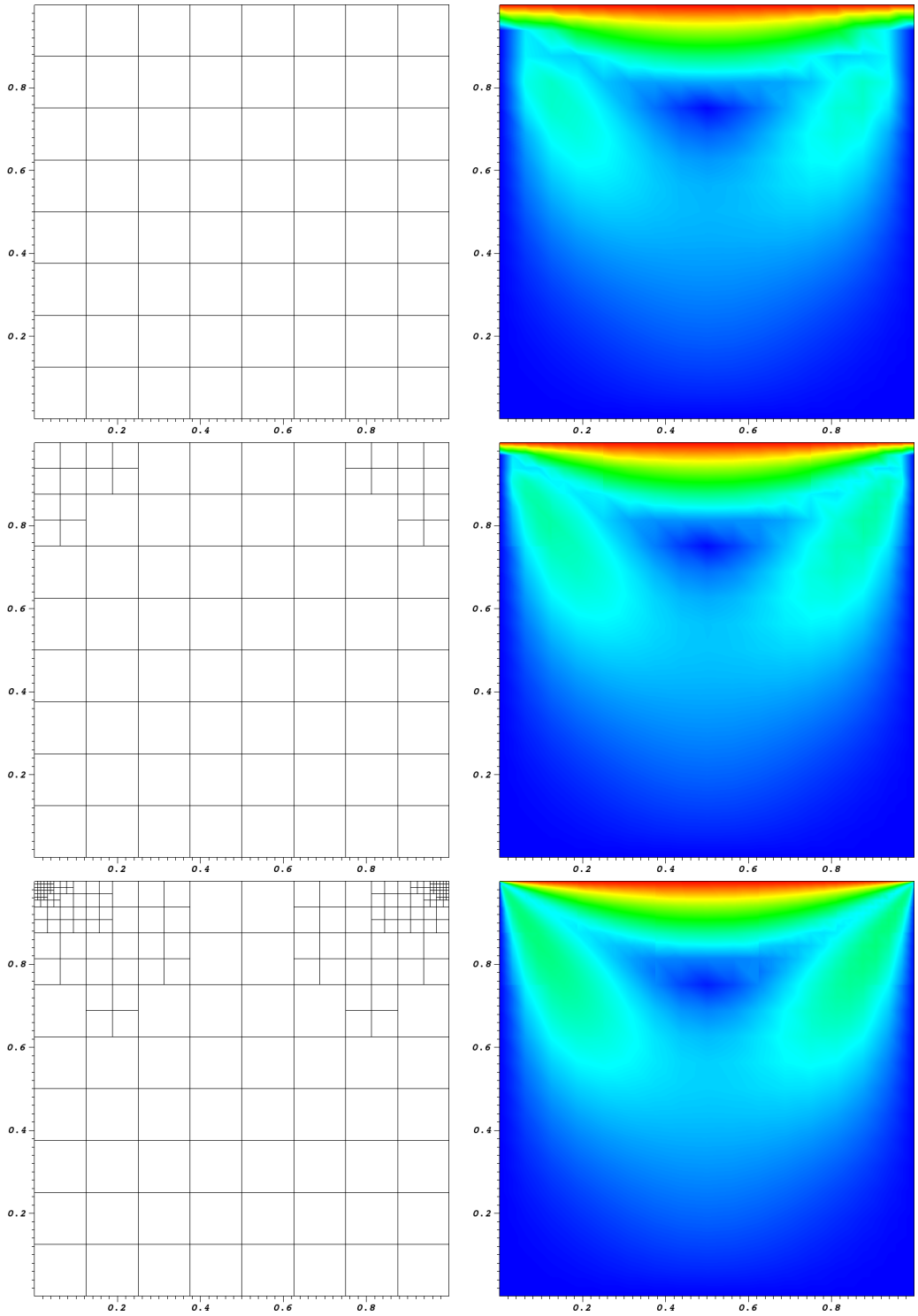

Figure 4.6: Adaptive mesh refinement for the lid driven cavity problem on the unit square. Mesh and velocity magnitude shown for, top to bottom: uniform mesh, single adaptive refinement, three adaptive refinements.

4.10 Unsteady Flow

The time dependent Stokes flow equations are:

$$\boldsymbol{u}_t - \nu\Delta\boldsymbol{u} + \nabla p = \boldsymbol{f}(\boldsymbol{x}, t) \qquad \text{for } t \in [0, T], \boldsymbol{x} \in \Omega \tag{4.14a}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{4.14b}$$

$$\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}) \tag{4.14c}$$

Using the same techniques as in 4.2.1 we can derive a Galerkin weak formulation:

$$(\boldsymbol{u}_t, \boldsymbol{v}) + a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) \qquad \text{for all } \boldsymbol{v} \in \mathbf{H}_0^1(\Omega)$$

$$b(\boldsymbol{u}, q) = 0 \qquad \text{for all } q \in L^2(\Omega)$$

and the discrete weak formulation:

$$(\boldsymbol{u}_t^h, \boldsymbol{v}^h) + a(\boldsymbol{u}^h, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^h) = (\boldsymbol{f}, \boldsymbol{v}^h) \qquad \text{for all } \boldsymbol{v}^h \in \mathbf{V}^h \tag{4.15a}$$

$$b(\boldsymbol{u}^h, q^h) = 0 \qquad \text{for all } q^h \in S^h \tag{4.15b}$$

where $a(\cdot,\cdot)$, $b(\cdot,\cdot)$ $\mathbf{V}^h$ and S^h are defined as before. To discretize time we begin by splitting our time interval $[0, T]$ into M intervals of uniform length $\delta = T/M$. Then let $\boldsymbol{u}^m$, p^m and $\boldsymbol{f}^m$ be equal to $\boldsymbol{u}^h(\boldsymbol{x}, m\delta)$, $p^h(\boldsymbol{x}, m\delta)$ and $\boldsymbol{f}(\boldsymbol{x}, m\delta)$ respectively. We will use the backward Euler method to discretize the $\boldsymbol{u}_t$ term:

$$\boldsymbol{u}_t(\boldsymbol{x}, (m+1)\delta) = \frac{\boldsymbol{u}^{m+1} - \boldsymbol{u}^m}{\delta}$$

Substituting this into (4.15) yields:

$$\frac{1}{\delta}(\boldsymbol{u}^{m+1}, \boldsymbol{v}^h) + a(\boldsymbol{u}^{m+1}, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^{m+1}) = (\boldsymbol{f}^{m+1}, \boldsymbol{v}^h) + \frac{1}{\delta}(\boldsymbol{u}^m, \boldsymbol{v}^h) \qquad \text{for all } \boldsymbol{v}^h \in \mathbf{V}^h$$

$$b(\boldsymbol{u}^{m+1}, q^h) = 0 \qquad \text{for all } q^h \in S^h$$

Note that an initial condition $\boldsymbol{u}^0$ is required for the velocity, but since there is no p_t term we do not require an initial condition for the pressure.

4.10.1 Implementation

Implementing an unsteady Stokes solver with backwards Euler is not much more difficult than the steady case. In our class definition we will need a few additional things:

- an initial conditions function, *Function<dim> *forcing_function*
- the solution from the previous timestep, *Vector<double> old_solution*
- a mass matrix, *SparseMatrix<double> mass_matrix*, as well as a routine to assemble it *void assemble_mass_matrix()*
- a routine to run the time loop, *void run_time_loop()*

Assembling the mass matrix and the new system matrix requires only minor modifications to the assembly routines discussed above.

```
template<int dim>
void StokesSolver<dim>::run_time_loop()
{
    assemble_matrix();
    assemble_mass_matrix();

    VectorTools::interpolate(dof_handler, *initial_conditions, old_solution);
    solution = old_solution;

    double t = input.t0;

    Vector<double> tmp;
    tmp.reinit(solution.size());//tmp is contribution from previous time step

    while (t < input.tf)
    {
        t += input.dt;

        forcing_function->set_time(t);
        boundary_values->set_time(t);

        mass_matrix.vmult(tmp, old_solution);

        //built in routine to create the rhs (f,phi)
        VectorTools::create_right_hand_side(dof_handler, QGauss<dim>(fe.degree+1),
                *forcing_function, system_rhs);

        system_rhs *= input.dt;
        system_rhs.add(tmp);

        constraints.condense(system_matrix, system_rhs);

        solve();
```

```
        old_solution = solution;
    }
}
```

4.10.2 Convergence Study

Implementing this problem is very similar to the steady state case. To test our code we again use the method of manufactured solutions. Given the exact solution:

$$\boldsymbol{u}(x,y,t) = \begin{pmatrix} e^{-t}\cos(\pi x) \\ e^{-t} y\pi \sin(\pi x) \end{pmatrix} \tag{4.16}$$

$$p(x,y,t) = (xy)^2 \tag{4.17}$$

we can generate an initial condition, a forcing function and boundary conditions on the unit square. Since backwards Euler is order δ accurate in time, in order to see the expected h^3 accuracy for $\boldsymbol{u}^h$, we take $\delta = h^3$. Taking $T = 1/64$, we can generate tables 4.5 and 4.6 that demonstrate the convergence rates predicted by (4.9).

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	3.574×10^{-3}	1.069×10^{-3}	-	3.302×10^{-2}	-
64	659	4.593×10^{-4}	1.337×10^{-4}	3.00	8.278×10^{-3}	2.00
256	2467	5.938×10^{-5}	1.672×10^{-5}	3.00	2.071×10^{-3}	2.00

Table 4.5: Convergence rates for $\boldsymbol{u}(\boldsymbol{x}, 1/64)$ in the unsteady Stokes equations using Taylor-Hood elements in space and backwards Euler in time.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	7.156×10^{-2}	5.599×10^{-2}	-	1.082×10^{-1}	-
64	659	1.548×10^{-2}	1.260×10^{-2}	2.15	4.727×10^{-2}	1.19
256	2467	4.006×10^{-3}	3.335×10^{-3}	1.92	2.304×10^{-2}	1.04

Table 4.6: Convergence rates for $p(\boldsymbol{x}, 1/64)$ in the unsteady Stokes equations using Taylor-Hood elements in space and backwards Euler in time.

4.10.3 Lid Driven Cavity

The time dependent lid driven cavity problem is again commonly used to test unsteady flows. The setup is the exact same as in section 4.7: the lid moves to the right with speed $u = 1$ and all other faces are stationary. The initial condition is $\boldsymbol{u} = \boldsymbol{0}$ everywhere. The computed velocity magnitudes at various times are shown in figure 4.7.

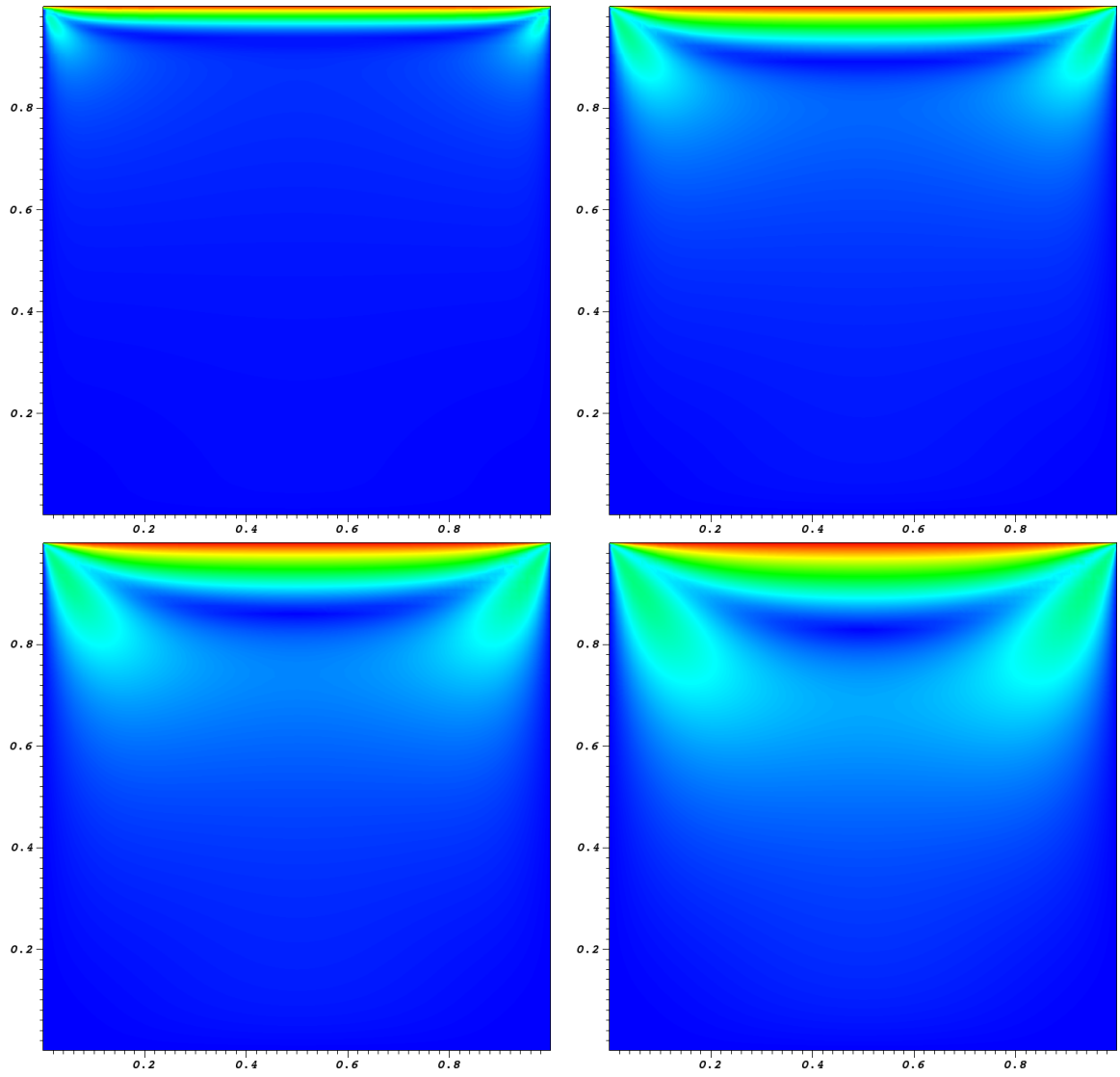

Figure 4.7: Velocity magnitude for time dependent driven cavity flow with $\nu = 0.01$. From top left clockwise, $t = 0.05$, $t = 0.25$, $t = 0.5$, $t = 1$.

CHAPTER 5

NAVIER-STOKES FLOW

The Navier-Stokes equations describe the motion of a fluid. Unlike the Stokes equations, these form a system of nonlinear PDEs. This chapter will cover linearization of the equations, using the the Gateaux derivative, as well as the weak formulation. The derivation of the both the Galerkin weak formulation and the discrete weak formulation follow closely to those of the Stokes equations given in section 4.2.1 and 4.2.2. Code will be provided that demonstrates how to resolve the nonlinear term using Newton's method. Convergence studies are given for both the steady and unsteady equations and some comparisons to published results are provided.

5.1 Steady Governing Equations

Inside a domain Ω with boundary Γ, the non-dimensional steady state Navier-Stokes equations for an incompressible Newtonian fluid are given by:

$$-\nu\Delta\boldsymbol{u} + \boldsymbol{u}\cdot\nabla\boldsymbol{u} + \nabla p = \boldsymbol{f}(\boldsymbol{x}) \tag{5.1a}$$

$$\nabla\cdot\boldsymbol{u} = 0 \tag{5.1b}$$

where:

- $\boldsymbol{u}$ is the velocity of the fluid; in 2D $\boldsymbol{u} = \langle u, v\rangle$
- p is the fluid pressure
- $\boldsymbol{f}(\boldsymbol{x}) \in \mathbf{H}^{-1}(\Omega)$ is a known forcing function
- ν is the (constant) kinematic viscosity of the fluid

These equations come of course with boundary conditions, the simplest of which is just Dirichlet boundary conditions on the velocity. Other possible boundary conditions exist, see section 4.8 for a discussion of stress boundary conditions or [5] for other boundary conditions. All results discussed concerning boundary conditions for the Stokes equations apply for the Navier-Stokes equations.

5.1.1 Reynolds Number

The Reynolds number is a dimensionless number which is used to compare fluid flow characteristics across different problems. It is defined as the ratio of inertial to viscous forces, given by the formula:

$$Re = \frac{UL}{\nu},$$

where L is a characteristic length and U is a characteristic velocity. In general, flows at low Reynolds numbers will be laminar and the flow becomes more turbulent as the Reynolds number increases. The Stokes equations described in the previous section can be thought of as the limiting case of the Navier-Stokes equations as $Re \to 0$.

5.2 Linearization

A difficulty in solving the Navier-Stokes equations is the nonlinear term in (5.1a). This term can be linearized using Newton's method. In general, to find a root of a function $H(x) = 0$, Newton's method can be written as:

$$H'(x, \delta x) = -H(x)$$

or equivalently, given x^0:

$$H'(x^k, x^k - x^{k-1}) = -H(x^k) \tag{5.2}$$

for $k = 0, 1, \cdots$. For scalars, $H'(x)$ is just the ordinary derivative, while for vectors it is the Jacobian. For functions, $H'(x, \delta x)$ is the Gâteaux derivative given by:

$$H'(x, \delta x) = \lim_{\epsilon \to 0} \frac{H(x + \epsilon \delta x) - H(x)}{\epsilon} \tag{5.3}$$

This provides us with a way to linearize (5.1a). Let:

$$H(x) = -\nu \Delta \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nabla p - \boldsymbol{f}$$

then we can use (5.3) to calculate H':

$$\begin{aligned}
&H'(\{\boldsymbol{u}^k, p^k\},\{\delta\boldsymbol{u}, \delta p\}) \\
&\qquad = \lim_{\epsilon\to 0} \frac{1}{\epsilon}\Big(-\nu\Delta(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u}^k) + (\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u})\cdot\nabla(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u}) + \nabla(p^k + \epsilon\delta p) - \\
&\qquad\qquad \boldsymbol{f} - (-\nu\Delta\boldsymbol{u}^k + \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k + \nabla p^k - \boldsymbol{f})\Big) \\
&\qquad = \lim_{\epsilon\to 0} \frac{-\epsilon\nu\Delta\delta\boldsymbol{u} + \boldsymbol{u}^k\cdot\nabla(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u}) + \epsilon\delta\boldsymbol{u}\cdot\nabla(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u}^k) + \epsilon\delta p - \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k}{\epsilon} \\
&\qquad = \lim_{\epsilon\to 0} \frac{-\epsilon\nu\Delta\delta\boldsymbol{u} + \epsilon\boldsymbol{u}^k\cdot\nabla\delta\boldsymbol{u} + \epsilon\delta\boldsymbol{u}\cdot\nabla\boldsymbol{u}^k + \epsilon^2\delta\boldsymbol{u}\cdot\nabla\delta\boldsymbol{u} + \epsilon\delta p}{\epsilon} \\
&\qquad = -\nu\Delta\delta\boldsymbol{u} + \boldsymbol{u}^k\cdot\nabla\delta\boldsymbol{u} + \delta\boldsymbol{u}\cdot\nabla\boldsymbol{u} + \delta p
\end{aligned}$$

Remembering that $\delta\boldsymbol{u} = \boldsymbol{u}^{k+1} - \boldsymbol{u}^k$ and $\delta p = p^{k+1} - p^k$ this can be simplified even further:

$$H'(\{\boldsymbol{u}^k, p^k\}, \{\delta\boldsymbol{u}, \delta p\}) = -\nu\Delta\boldsymbol{u}^{k+1} + \nu\Delta\boldsymbol{u}^k + \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^{k+1} + \boldsymbol{u}^{k+1}\cdot\nabla\boldsymbol{u}^k - 2\boldsymbol{u}^k\cdot\nabla u^k + p^{k+1} - p^k$$

Plugging this into (5.2) yields:

$$\begin{aligned}
-\nu\Delta\boldsymbol{u}^{k+1} + \nu\Delta\boldsymbol{u}^k + \boldsymbol{u}^{k+1}\cdot\nabla\boldsymbol{u}^k + \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^{k+1} - 2\boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k + p^{k+1} - p^k &= -(-\nu\Delta\boldsymbol{u}^k + \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k + \nabla p^k - \boldsymbol{f}) \\
\Rightarrow -\nu\Delta\boldsymbol{u}^{k+1} + \boldsymbol{u}^{k+1}\nabla\cdot\boldsymbol{u}^k + \boldsymbol{u}\nabla\cdot\boldsymbol{u}^{k+1} + p^{k+1} &= \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k + \boldsymbol{f}
\end{aligned}$$

In other words, given $\boldsymbol{u}^0$, we can solve this sequence of linear systems:

$$-\nu\Delta\boldsymbol{u}^{k+1} + \boldsymbol{u}^{k+1}\nabla\cdot\boldsymbol{u}^k + \boldsymbol{u}\nabla\cdot\boldsymbol{u}^{k+1} + p^{k+1} = \boldsymbol{u}^k\cdot\nabla\boldsymbol{u}^k + \boldsymbol{f} \tag{5.4a}$$

$$\nabla\cdot\boldsymbol{u}^{k+1} = 0 \tag{5.4b}$$

to find a solution to (5.1a) and (5.1b). Note that no initial guess for the pressure is needed, since (5.1a) is linear in p. If $\boldsymbol{u}^0$ is sufficiently close to the actual solution, then this method will converge quadratically.

5.3 Weak Formulation

5.3.1 Galerkin Weak Formulation

The Galerkin weak formulation for the Navier-Stokes equations (5.1a) and (5.1b) is derived in exactly the same way as for the Stokes equations in section 4.2.1. Taking the inner product of (5.1a) with a test function $\boldsymbol{v} \in \mathbf{H}_0^1(\Omega)$ and (5.1b) with a test function $q \in L^2(\Omega)$ and applying Green's identity where appropriate, we get:

$$a(\boldsymbol{u}, \boldsymbol{v}) + c(\boldsymbol{u}, \boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) \qquad \text{for all } \boldsymbol{v} \in \mathbf{H}_0^1(\Omega) \tag{5.5a}$$

$$b(\boldsymbol{u}, q) = 0 \qquad \text{for all } q \in L^2(\Omega) \tag{5.5b}$$

where:

- $a(\boldsymbol{u}, \boldsymbol{v}) = \nu\,(\nabla \boldsymbol{u}, \nabla \boldsymbol{v})$
- $b(\boldsymbol{v}, q) = -(q, \nabla \cdot \boldsymbol{v})$
- $c(\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v}) = \boldsymbol{u} \cdot \nabla \boldsymbol{w} \cdot \boldsymbol{v}$

Given the linearization (5.4), we can turn (5.5a) and (5.5b) into a sequence of linear problems:

$$a(\boldsymbol{u}^{k+1}, \boldsymbol{v}) + c(\boldsymbol{u}^{k+1}, \boldsymbol{u}^k, \boldsymbol{v}) + c(\boldsymbol{u}^k, \boldsymbol{u}^{k+1}, \boldsymbol{v}) + b(\boldsymbol{v}, p^{k+1}) = (\boldsymbol{f}, \boldsymbol{v}) + c(\boldsymbol{u}^k, \boldsymbol{u}^k, \boldsymbol{v}) \quad \text{for all } \boldsymbol{v} \in \mathbf{H}_0^1(\Omega) \tag{5.6a}$$

$$b(\boldsymbol{u}^{k+1}, q) = 0 \quad \text{for all } q \in L^2(\Omega) \tag{5.6b}$$

5.3.2 Discrete Weak Formulation

We begin again by choosing $\mathbf{V}^h \subset \mathbf{H}_0^1(\Omega)$ and $S^h \subset L^2(\Omega)$, such that:

$$a(\boldsymbol{u}^{k+1}, \boldsymbol{v}^h) + c(\boldsymbol{u}^{k+1}, \boldsymbol{u}^k, \boldsymbol{v}^h) + c(\boldsymbol{u}^k, \boldsymbol{u}^{k+1}, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^{k+1}) = (\boldsymbol{f}, \boldsymbol{v}^h) + c(\boldsymbol{u}^k, \boldsymbol{u}^k, \boldsymbol{v}^h) \quad \text{for all } \boldsymbol{v}^h \in \mathbf{V}^h \tag{5.7a}$$

$$b(\boldsymbol{u}^{k+1}, q^h) = 0 \quad \text{for all } q^h \in S^h \tag{5.7b}$$

where it is understood that for the sake of cleaner notation, in this context $\boldsymbol{u}^k$ and p^k are the k-th iteration of the finite element approximation to $\boldsymbol{u}$ and p. Thus $\boldsymbol{u}^k \in \mathbf{V}^h$ and $p^k \in S^h$.

Just like the Stokes equations, in order for a unique solution to exist we must choose $\mathbf{V}^h$ and S^h such that $b(\boldsymbol{v}^h, q^h)$ satisfies the LBB condition (4.7). Again, the Taylor-Hood element pair described in section 4.3.1 satisfies this condition.

5.4 Implementation

Implementation of a finite element solver for the Navier-Stokes equations is very similar to the implementation for the Stokes equations solver described in section 4.5. The main difference is that now we have to do a nonlinear solve.

In our class definition for **NavierStokesSolver** we will need to keep track of the previous Newton iteration, *Vector<double> previous_newton_step*. During the matrix assembly we will have to evaluate the previous velocity solution at the quadrature points in each cell.

```
template<int dim>
void NavierStokesSolver<dim>::assemble_system()
{
    //clear the system matrix and rhs, because these change every iteration
    system_matrix.reinit(sparsity_pattern);
    system_rhs.reinit(dof_handler.n_dofs());

    QGauss<dim>                 quadrature_formula (fe.degree + 1);
```

```
const int                   dofs_per_cell = fe.dofs_per_cell;
const int                   n_q_points = quadrature_formula.size();
const int                   n_q_points_face = face_quadrature_formula.size();

std::vector<Tensor<1,dim> >   previous_newton_velocity_values (n_q_points);
std::vector<Tensor< 2, dim> > previous_newton_velocity_gradients (n_q_points);

std::vector<Vector<double> > rhs_values (n_q_points, Vector<double>(dim+1));
std::vector<Tensor<2,dim> >  grad_phi_u (dofs_per_cell);
std::vector<double>          div_phi_u (dofs_per_cell);
std::vector<double>          phi_p (dofs_per_cell);
std::vector<Tensor<1,dim> >  phi_u (dofs_per_cell);

Vector<double>              cell_rhs (dofs_per_cell);
FullMatrix<double>          cell_matrix (dofs_per_cell, dofs_per_cell);

std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
const FEValuesExtractors::Vector velocities (0);
const FEValuesExtractors::Scalar pressure (dim);

FEValues<dim> fe_values(fe, quadrature_formula, update_values |
              update_gradients | update_JxW_values | update_quadrature_points);

typename DoFHandler<dim>::active_cell_iterator
        cell = dof_handler.begin_active(), endc = dof_handler.end();

for (; cell!=endc; ++cell)
{
    fe_values.reinit(cell);
    cell_matrix = 0;
    cell_rhs = 0;

    //Calculate velocity values and gradients from previous newton iteration
    //at each quadrature point in cell
    fe_values[velocities].get_function_values(previous_newton_step,
                                    previous_newton_velocity_values);
    fe_values[velocities].get_function_gradients(previous_newton_step,
                                    previous_newton_velocity_gradients);

    forcing_function->vector_value_list(fe_values.get_quadrature_points(),
                                        rhs_values);

    //calculate cell contribution to system
    for (int q = 0; q < n_q_points; q++)
    {
        for (int k=0; k<dofs_per_cell; k++)
        {
            grad_phi_u[k] = fe_values[velocities].gradient (k, q);
            div_phi_u[k]  = fe_values[velocities].divergence (k, q);
            phi_p[k]      = fe_values[pressure].value (k, q);
            phi_u[k]      = fe_values[velocities].value (k, q);
```

```
            }

            for (int i = 0; i < dofs_per_cell; i++)
            {
                for (int j = 0; j < dofs_per_cell; j++)
                {

                    cell_matrix(i,j) +=
                                (input.nu*double_contract(grad_phi_u[i],grad_phi_u[j])
                                            + phi_u[j]
                                            *transpose(
                                               previous_newton_velocity_gradients[q])
                                            *phi_u[i]
                                            + previous_newton_velocity_values[q]
                                            *transpose(grad_phi_u[j])*phi_u[i]
                                            - phi_p[j]*div_phi_u[i]
                                            - phi_p[i]*div_phi_u[j])
                                        *fe_values.JxW(q);

                }

                int equation_i = fe.system_to_component_index(i).first;
                cell_rhs[i] +=
                            (fe_values.shape_value(i,q)*rhs_values[q](equation_i)
                             + previous_newton_velocity_values[q]
                             *transpose(
                                 previous_newton_velocity_gradients[q])*phi_u[i])
                             *fe_values.JxW(q);
            }
        }

        cell->get_dof_indices(local_dof_indices);
        constraints.distribute_local_to_global(cell_matrix, cell_rhs,
                                local_dof_indices, system_matrix, system_rhs);
    }
}
```

We need two additional things for our nonlinear solve: an initial guess and a stopping criteria. For our initial guess we will simply use the solution of the Stokes equations for the same ν, boundary conditions and forcing function.

Recall that we are solving for the coefficients α and β in our basis expansions:

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{k=1}^{K} \alpha_k \boldsymbol{v}_k(\boldsymbol{x})$$

and

$$p^h(\boldsymbol{x}) = \sum_{j=1}^{J} \beta_j q_j(\boldsymbol{x}).$$

Then to test for convergence we look at the norm of the difference in our solution vector:

$$\text{residual} = \frac{\left(\sum_{\ell=1}^{L}(\alpha_\ell^k - \alpha_\ell^{k-1})^2 + \sum_{j=1}^{J}(\beta_j^k - \beta_j^{k-1})^2\right)^{1/2}}{L+J}.$$

If this residual gets below a certain tolerance we say the sequence has converged.

```
template<int dim>
void NavierStokesSolver<dim>::run_newton_loop(int cycle)
{
    int MAX_ITER = 10;
    double TOL = 1e-8;

    int iter = 0;
    double residual = 0;

    //solve Stokes equations for initial guess
    assemble_stokes_system();
    solve();

    previous_newton_step =  solution;

    while (iter == 0 || (residual > TOL && iter < MAX_ITER))
    {
        assemble_system(input.nu);
        solve();

        Vector<double> res_vec = solution;
        res_vec -= previous_newton_step;

        residual = res_vec.l2_norm()/(dof_handler.n_dofs());
        previous_newton_step = solution;

        iter++;
        printf("Residual = %4.10e\n", residual);
    }

    if (iter == MAX_ITER)
    {
        printf("WARNING: Newton's method failed to converge\n");
    }
}
```

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	1.140×10^{-2}	3.42×10^{-3}	-	1.054×10^{-1}	-
64	659	1.498×10^{-3}	4.275×10^{-4}	3.00	2.642×10^{-2}	2.00
256	2467	1.895×10^{-4}	5.337×10^{-5}	3.00	6.609×10^{-3}	2.00

Table 5.1: Convergence rates for $\boldsymbol{u}(\boldsymbol{x})$ in the steady Navier-Stokes equations using Taylor-Hood elements.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	2.189×10^{-1}	1.667×10^{-1}	-	1.959×10^{-1}	-
64	659	5.801×10^{-2}	4.661×10^{-2}	1.84	6.440×10^{-2}	1.61
256	2467	1.499×10^{-2}	1.231×10^{-2}	1.92	2.565×10^{-2}	1.33

Table 5.2: Convergence rates for $p(\boldsymbol{x})$ in the steady Navier-Stokes equations using Taylor-Hood elements.

5.4.1 Convergence Study

We can use the same functions (4.11) and (4.12) as we did for Stokes flow to generate a forcing function and boundary conditions on the unit square. Doing so leads to the convergence tables 5.1 and 5.2. These tables demonstrate the convergence rates predicted by (4.9).

We are also interested in the performance of our Newton solver. For the above problem with $\nu = 1$ and $h = 1/8$, Newton's method converges in 2 iterations to a tolerance of 1×10^{-8} with residuals of 3.07×10^{-2} and 1.20×10^{-11}. Iteration counts for other values of ν are given in table 5.3. As ν decreases the Stokes solution becomes a worse initial guess, so we require more iterations to find a solution. Eventually when we reach a small enough ν, Newton's method fails to converge.

Recall that the convergence rate of Newton's method should be quadratic. Given a sequence of residuals, $\{r_i\}$, $i = 1, \cdots, n$, the convergence rates $\{q_j\}$, $j = 1, \cdots, n-2$ of Newton's method can

ν	iterations
1	2
0.1	3
0.05	4
0.01	8
0.005	did not converge in 10 iterations

Table 5.3: Performance of Newton's method for various values of ν using convergence test problem.

residual	convergence
1.675×10^{-2}	-
1.566×10^{-2}	-
6.356×10^{-3}	13.40
4.724×10^{-3}	0.32
2.497×10^{-3}	2.15
7.751×10^{-4}	1.83
1.333×10^{-4}	1.51
5.302×10^{-6}	1.83
6.302×10^{-9}	2.09

Table 5.4: Convergence rates for Newton's method in the test problem at $\nu = 0.01$.

be calculated from the formula:

$$q_i = \frac{\log(r_{i+2}/r_{i+1})}{\log(r_{i+1}/r_i)}.$$

For $\nu = 0.01$, the residuals and convergence rates are given in table 5.4. The residuals appear to approach quadratic convergence.

5.4.2 Lid Driven Cavity

For the Lid Driven Cavity problem, as described in section 4.7, we can compute the Reynolds number. On the unit box, the characteristic length, L is 1. The characteristic velocity for this problem is the speed of the lid, u_0. If we take this speed to be 1 as well, then the Reynolds number for this problem is simply ν^{-1}.

Our results compare well with those in [4] as shown in figure 5.1.

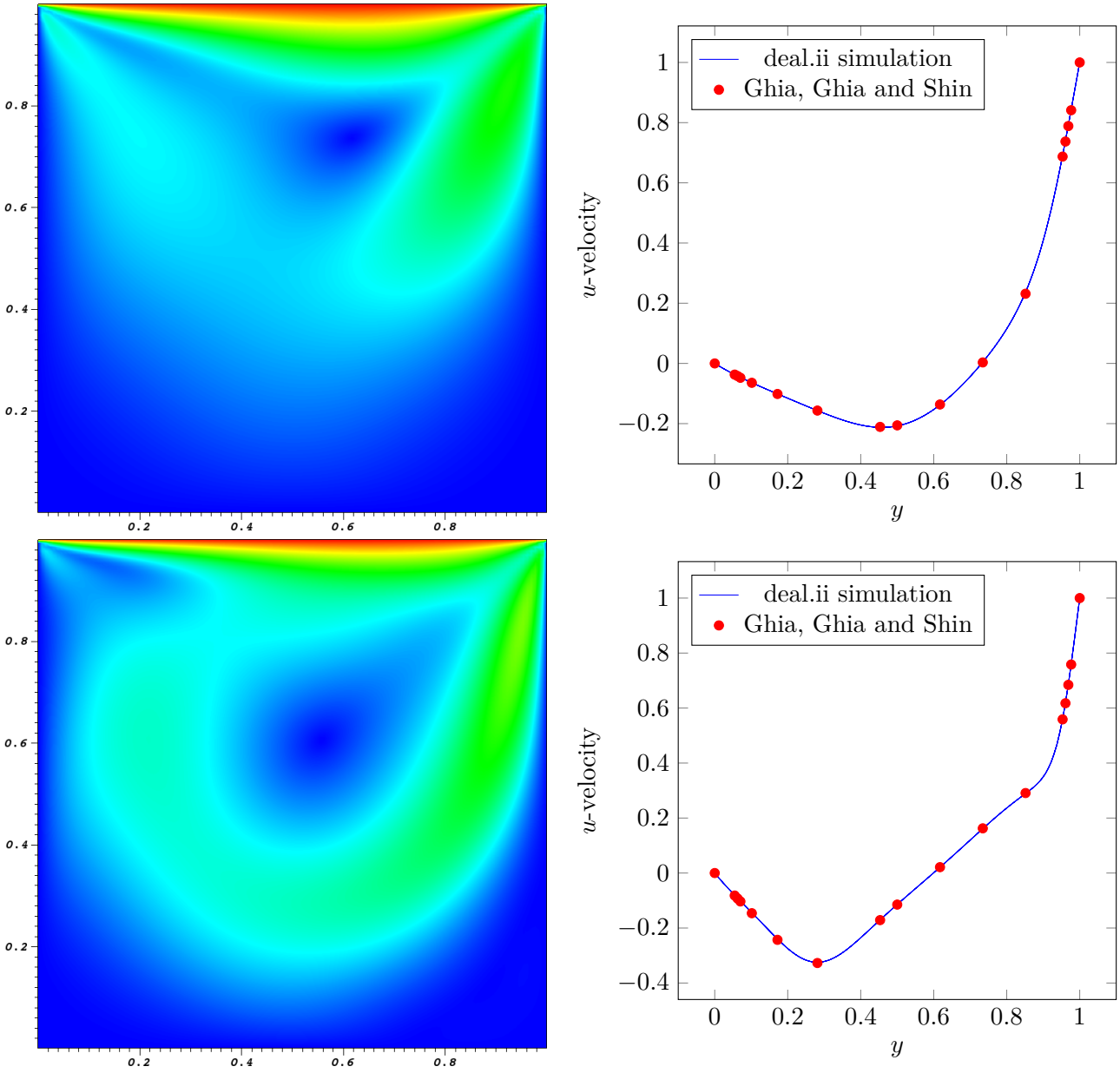

Figure 5.1: Steady state velocity magnitude for the driven cavity problem and comparison with published results of u velocity at $x = 0.5$. Top: $Re = 100$, bottom: $Re = 400$.

5.4.3 Flow Around a Cylinder

Another benchmark problem we can look at is the 2D flow past a cylinder problem. The geometry is given in figure 5.2.

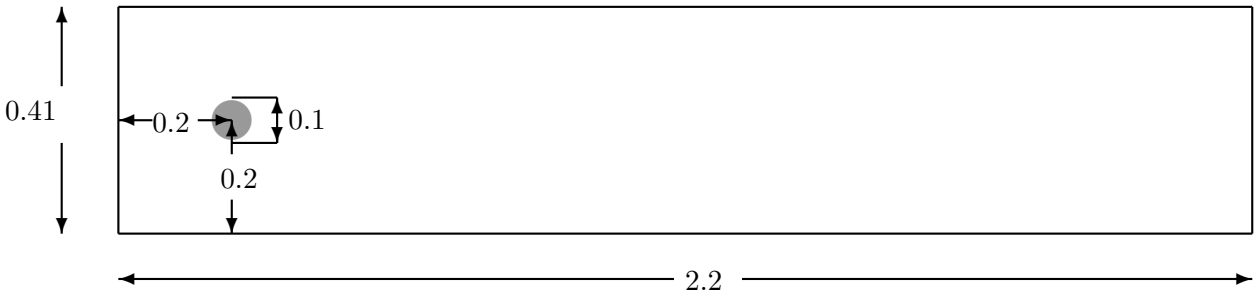

Figure 5.2: Sketch of the domain used for flow around a cylinder, see [8].

For low Re we can assume there exists a steady state solution. Let us prescribe the inflow and outflow velocity as:

$$\boldsymbol{u}(\boldsymbol{x}) = \begin{pmatrix} \frac{4U_{\max}y(h-y)}{h^2} \\ 0 \end{pmatrix},$$

where h is the domain height, in our case 0.41, and $U_{\max}$ is the maximum inflow speed. Assume no-slip boundary conditions on all other boundaries.

The Reynolds number can be calculated, with U being the average inflow velocity:

$$U = \frac{1}{h}\int_0^h \frac{4U_{\max}y(h-y)}{h^2}\mathrm{d}y = \frac{4U_{\max}}{h^3}\left(\frac{hy^2}{2} - \frac{y^3}{3}\right)\Big|_0^h = \frac{2}{3}U_{\max}.$$

The characteristic length scale for this problem is the diameter of the circle, thus $L = 0.1$. This means that we can compute Re as:

$$Re = \frac{2U_{\max}(0.1)}{3\nu}. \tag{5.8}$$

If we set $U_{\max} = 0.3$ and $\nu = 0.01$, then we get that $Re = 20$.

Some quantities of interest are the lift and drag on the circle, as well as the pressure drop, $p_2 - p_1$, between the points $(0.15, 2)$ and $(0.25, 2)$, which are just in front of and behind the circle.

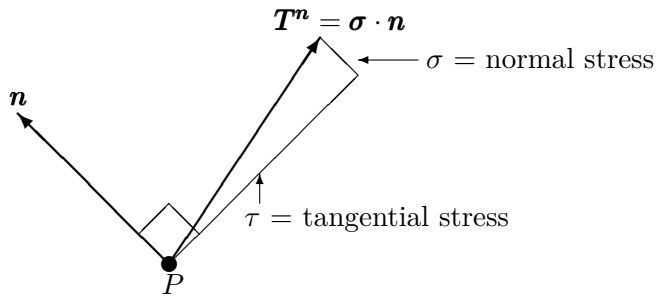

Figure 5.3: Normal and tangential stresses at a point P on the circle.

As shown in figure 5.3, the stress vector normal to a point on the circle can be calculated by:

$$\boldsymbol{T^n} = \langle \tau, \sigma \rangle = \boldsymbol{\sigma} \cdot \boldsymbol{n},$$

where τ is the component of $\boldsymbol{T^n}$ tangential to the surface, σ is the component normal to the surface, and the stress tensor $\boldsymbol{\sigma}$ is given by:

$$\boldsymbol{\sigma} = \nu \nabla \boldsymbol{u} - p\boldsymbol{I}.$$

The total drag force, F_D, and lift force, F_L, on the circle are calculated by taking the line integral of the normal stress vector over the entire circle:

$$\langle F_D, F_L \rangle = \oint_S \boldsymbol{T^n} \mathrm{d}s.$$

From these values we can calculate the dimensionless lift and drag coefficients:

$$C_D = \frac{2}{U^2 L} F_D \qquad C_L = \frac{2}{U^2 L} F_L.$$

In deal.ii the lift and drag coefficients, along with the pressure drop can be calculated as follows:

```
template<int dim>
void NavierStokesSolver<dim>::calculate_lift_and_drag()
{
    QGauss<dim-1>               face_quadrature_formula(3);
    const int                   n_q_points = face_quadrature_formula.size();

    const FEValuesExtractors::Vector velocities (0);
    const FEValuesExtractors::Scalar pressure (dim);

    std::vector<double>     pressure_values(n_q_points);
```

```
std::vector<Tensor< 2, dim> > velocity_gradients(n_q_points);

Tensor<1,dim>    normal_vector;
Tensor<2,dim>    fluid_stress;
Tensor<2,dim>    fluid_pressure;
Tensor<1,dim>    forces;

FEFaceValues<dim> fe_face_values (fe, face_quadrature_formula, update_values
        | update_quadrature_points | update_gradients | update_JxW_values
        | update_normal_vectors );

typename DoFHandler<dim>::active_cell_iterator
        cell = dof_handler.begin_active(), endc = dof_handler.end();

double drag = 0;
double lift = 0;

for (; cell!=endc; ++cell)
{
    for (int face=0; face < GeometryInfo<dim>::faces_per_cell; face++)
    {
        if (cell->face(face)->at_boundary())
        {
            fe_face_values.reinit (cell, face);
            std::vector<Point<dim> > q_points =
                        fe_face_values.get_quadrature_points();

            if (cell->face(face)->boundary_indicator() == 4)//on the cirlce
            {
                fe_face_values[velocities].
                    get_function_gradients(solution, velocity_gradients);
                fe_face_values[pressure].
                    get_function_values(solution, pressure_values);

                for (int q = 0; q < n_q_points; q++)
                {
                    normal_vector = -fe_face_values.normal_vector(q);

                    fluid_pressure[0][0] = pressure_values[q];
                    fluid_pressure[1][1] = pressure_values[q];

                    fluid_stress = input.nu*velocity_gradients[q]
                                        - fluid_pressure;

                    forces = fluid_stress*normal_vector*fe_face_values.JxW(q);
```

```
                        drag += forces[0];
                        lift += forces[1];
                    }
                }
            }
        }
    }

    //calculate pressure drop
    Point<dim> p1, p2;
    p1[0] = 0.15;
    p1[1] = 0.2;
    p2[0] = 0.25;
    p2[1] = 0.2;
    Vector<double> solution_values1(dim+1);
    Vector<double> solution_values2(dim+1);

    VectorTools::point_value(dof_handler, solution, p1, solution_values1);
    VectorTools::point_value(dof_handler, solution, p2, solution_values2);

    double p_diff = solution_values1(dim) - solution_values2(dim);
}
```

The computed velocity magintude is shown in figure 5.4. We can compare our results with those in [8]. In that paper several simulations were made of this problem using various techniques. Our computed quantites of interest after 2 adaptive mesh refinements are:

	deal.ii	published results [8]
C_D	5.573086	5.5700 - 5.5900
C_L	0.009396	0.0104 - 0.0110
$p_2 - p_1$	0.117475	0.1172-0.1176

The lift is slightly low, but these values are all within the ranges calculated by the other models.

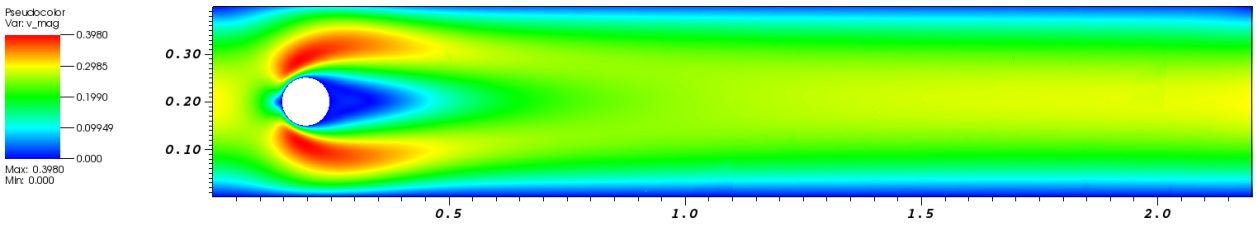

Figure 5.4: Velocity magnitude for flow around a cylinder at $Re = 20$.

5.5 Unsteady Flow

There are a wide range of problems for the Navier-Stokes equations that do not exhibit steady state solutions. In order to study these problems we must look at the time dependent Navier-Stokes equations, given by:

$$\boldsymbol{u}_t - \nu\Delta\boldsymbol{u} + \boldsymbol{u}\cdot\nabla\boldsymbol{u} + \nabla p = \boldsymbol{f}(\boldsymbol{x},t) \qquad \boldsymbol{x}\in\Omega, t\in[0,T]$$

$$\nabla\cdot\boldsymbol{u} = 0$$

$$\boldsymbol{u}(\boldsymbol{x},0) = \boldsymbol{u}_0(\boldsymbol{x})$$

These equations come with appropriate boundary conditions. We can discretize in time using backwards Euler as we did for Stokes flow in section 4.10, in which case we arrive at the sequence of linear systems:

$$\frac{1}{\delta}(\boldsymbol{u}^{m+1,(k+1)},\boldsymbol{v}^h) + a(\boldsymbol{u}^{m+1,(k+1)},\boldsymbol{v}^h) + c(\boldsymbol{u}^{m+1(k+1)},\boldsymbol{u}^{m+1,(k)},\boldsymbol{v}^h) + c(\boldsymbol{u}^{m+1,(k)},\boldsymbol{u}^{m+1,(k+1)},\boldsymbol{v}^h)$$
$$+ b(\boldsymbol{v}^h, p^{m+1,(k+1)}) = (\boldsymbol{f}^{m+1},\boldsymbol{v}^h) + c(\boldsymbol{u}^{m+1,(k)},\boldsymbol{u}^{m+1,(k)},\boldsymbol{v}^h) + (\boldsymbol{u}^m,\boldsymbol{v}^h)$$
$$\nabla\cdot\boldsymbol{u}^{m+1,(k+1)} = 0$$

where $\boldsymbol{u}^{m,(k)}$ and $p^{m,(k)}$ are the velocity and the pressure respectively at time $m\delta$ and Newton iteration k. Note that again we do not require any initial condition for the pressure and that we can use the initial velocity condition $\boldsymbol{u}_0(x)$ as $\boldsymbol{u}^0$. We can also use the solution from the previous timestep as the initial guess for Newton's method, i.e. $\boldsymbol{u}^{m+1,(0)} = \boldsymbol{u}^m$.

This method requires solving a nonlinear system at each timestep. This is in general very expensive. One way to get around this is to time lag the velocity in the nonlinear term, either partially, or fully. Partially lagging the velocity yields the now linear system:

$$\frac{1}{\delta}(\boldsymbol{u}^{m+1},\boldsymbol{v}^h) + a(\boldsymbol{u}^{m+1},\boldsymbol{v}^h) + c(\boldsymbol{u}^m,\boldsymbol{u}^{m+1},\boldsymbol{v}^h) + b(\boldsymbol{v}^h,p^{m+1}) = (\boldsymbol{f}^{m+1},\boldsymbol{v}^h) + \frac{1}{\delta}(\boldsymbol{u}^m,\boldsymbol{v}^h)$$
$$\nabla\cdot\boldsymbol{u}^{m+1)} = 0$$

This now requires solving a different linear system at each timestep. If we fully lag the velocity we get the system:

$$\frac{1}{\delta}(\boldsymbol{u}^{m+1},\boldsymbol{v}^h) + a(\boldsymbol{u}^{m+1},\boldsymbol{v}^h) + b(\boldsymbol{v}^h,p^{m+1}) = (\boldsymbol{f}^{m+1},\boldsymbol{v}^h) - c(\boldsymbol{u}^m,\boldsymbol{u}^m,\boldsymbol{v}^h) + (\boldsymbol{u}^m,\boldsymbol{v}^h)$$
$$\nabla\cdot\boldsymbol{u}^{m+1)} = 0$$

which require us to solve the same linear system at each timestep. This enables us to do an LU factorization of the matrix once outside the time loop and reuse it at each timestep. Of course since these methods do not explicitly treat the nonlinear term, they are less accurate than the full Newton solve at each timestep. This places restrictions on the size of the timestep. If the timestep is too large than time lagging the velocity may give inaccurate answers. All results in this thesis have been obtained doing a full Newton solve at each timestep.

5.5.1 Implementation

The deal.ii implementation of this is very similar to the implementation of the unsteady Stokes equations discussed in section 4.10.1. The main difference here is that we now have to implement Newton's method inside the time loop.

```
template<int dim>
void NavierStokesSolver<dim>::run_time_loop()
{
    VectorTools::interpolate(dof_handler, *initial_conditions, old_solution);
    solution = old_solution;

    assemble_mass_matrix();

    double t = input.t0;

    Vector<double> tmp; //tmp is contribution from previous time step
    tmp.reinit(solution.size());

    while (t < input.tf)
    {
        t += input.dt;

        forcing_function->set_time(t);
        boundary_values->set_time(t);

        constraints.close();

        int MAX_ITER = 10;
        double TOL = 1e-8;
        int iter = 0;
        double residual = 0;

        previous_newton_step =  old_solution;
        mass_matrix.vmult(tmp, old_solution);
```

```
        while (iter == 0 || (residual > TOL && iter < MAX_ITER))
        {
            assemble_system();
            system_rhs.add(tmp);

            constraints.condense(system_matrix, system_rhs);

            solve();

            Vector<double> res_vec = solution;
            res_vec -= previous_newton_step;

            residual = res_vec.l2_norm()/(dof_handler.n_dofs());
            previous_newton_step = solution;

            iter++;

            printf("Residual = %4.10e\n", residual);
        }

        if (iter == MAX_ITER)
        {
            printf("WARNING: Newton's method failed to converge\n");
        }

        old_solution = solution;
    }
}
```

5.5.2 Convergence Study

To test our code we again use the method of manufactured solutions. Using the same exact solution (4.16) and (4.17), with $\delta = h^3$, we can generate the error tables 5.5 and 5.6 which show the desired rate of convergence.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	1.123×10^{-2}	3.258×10^{-3}	-	1.043×10^{-1}	-
64	659	1.474×10^{-3}	4.166×10^{-4}	2.97	2.605×10^{-2}	2.00
256	2467	1.865×10^{-4}	5.240×10^{-5}	2.99	6.509×10^{-3}	2.00

Table 5.5: Convergence rates for $\boldsymbol{u}(\boldsymbol{x})$ in the unsteady Navier-Stokes equations using Taylor-Hood elements and backward Euler.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	187	2.452×10^{-1}	1.869×10^{-1}	-	1.845×10^{-1}	-
64	659	4.899×10^{-2}	4.055×10^{-2}	2.20	3.859×10^{-2}	2.26
256	2467	1.261×10^{-2}	1.059×10^{-2}	1.94	9.829×10^{-3}	1.97

Table 5.6: Convergence rates for $p(\boldsymbol{x})$ in the unsteady Navier-Stokes equations using Taylor-Hood elements and backward Euler.

5.5.3 Flow Around a Cylinder

We will consider the same problem as in 5.4.3, however this time for the inflow and outflow boundary conditions:

$$\boldsymbol{u}(\boldsymbol{x}) = \begin{pmatrix} \frac{4U_{\max} y(h-y)}{h^2} \\ 0 \end{pmatrix},$$

we will take $U_{\max}$ to be 1.5. From 5.8 this gives a Reynolds number of 100.

At this Reynolds number if we start with $\boldsymbol{u} = \boldsymbol{0}$ as our initial condition, after a certain amount of time our solution becomes periodic. The velocity magnitude for various times is shown in figure 5.5.

We can again compare our model to the results obtained in [8]. The coefficients C_L and C_D as well as the pressure drop $p_2 - p_1$ are defined in section 5.4.3. In addition we will calculate the Strouhal number, St, defined as:

$$St = \frac{Lf}{U}.$$

If we run the simulation to $t = 25$, the flow field is fully developed into a periodic solution. At this point we can look at any cycle of C_L, where a cycle starts at $t = t_0$ where the lift is largest, and ends at $t = t_0 + 1/f$ where C_L is largest again. See figure 5.6 for plots of the lift and drag coefficients as well as the pressure drop over one cycle.

Over one cycle we can calculate the maximum lift and drag coefficients and the maximum pressure drop as well as the Strouhal number:

	deal.ii	published results [8]
max C_D	3.181	3.2200 - 3.2400
max C_L	0.865	0.9900 - 1.0100
max $p_2 - p_1$	2.474	2.4600-2.500
St	0.296	0.2950-0.3050

As with the stationary case, the lift coefficient is a little low, but within the ranges of solutions given in [8].

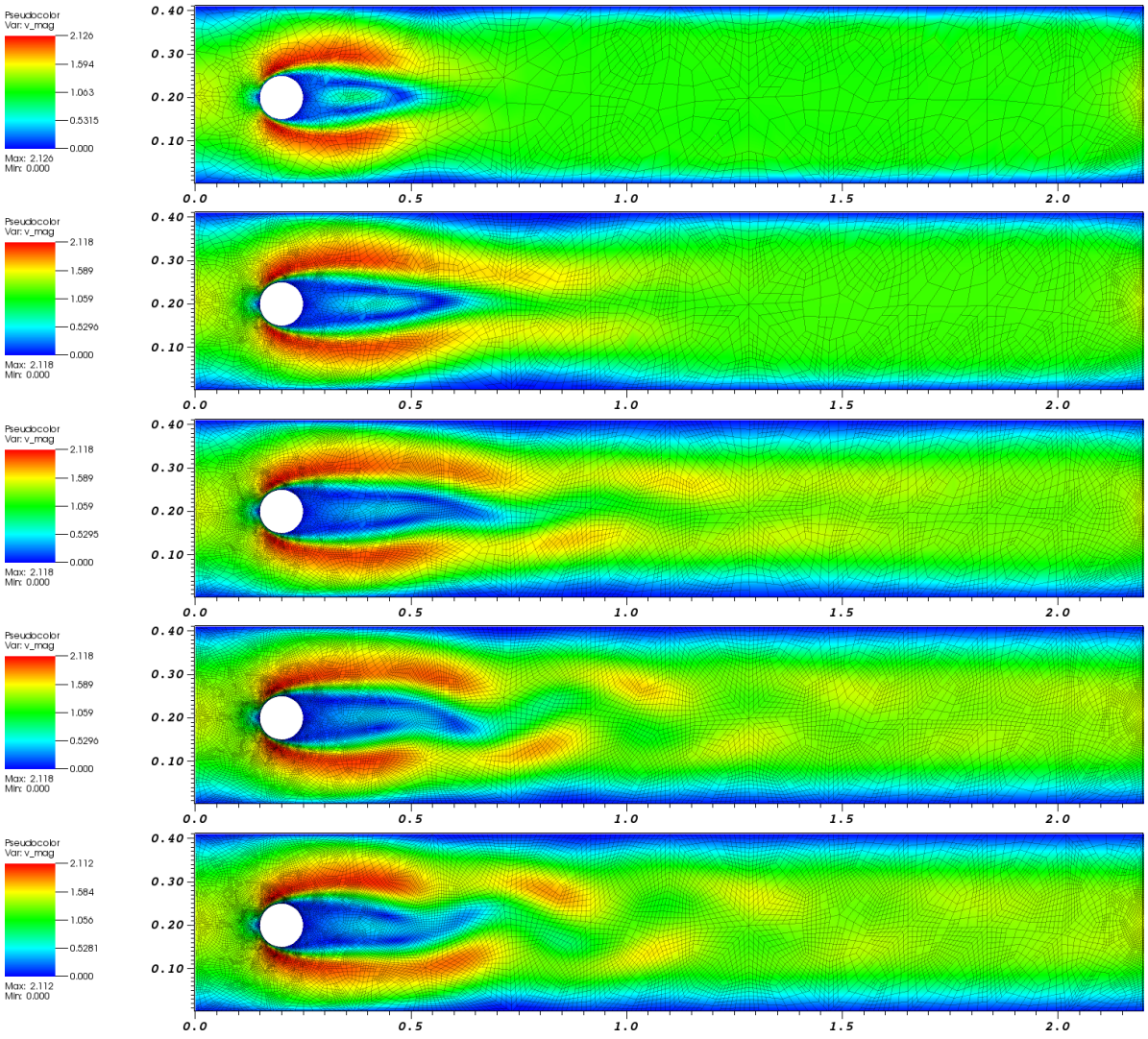

Figure 5.5: Velocity magnitude for flow past a cylinder at $Re = 100$. Top to bottom: $t = 0.5$, $t = 1.0$, $t = 2.0$, $t = 3.0$, $t = 3.2$.

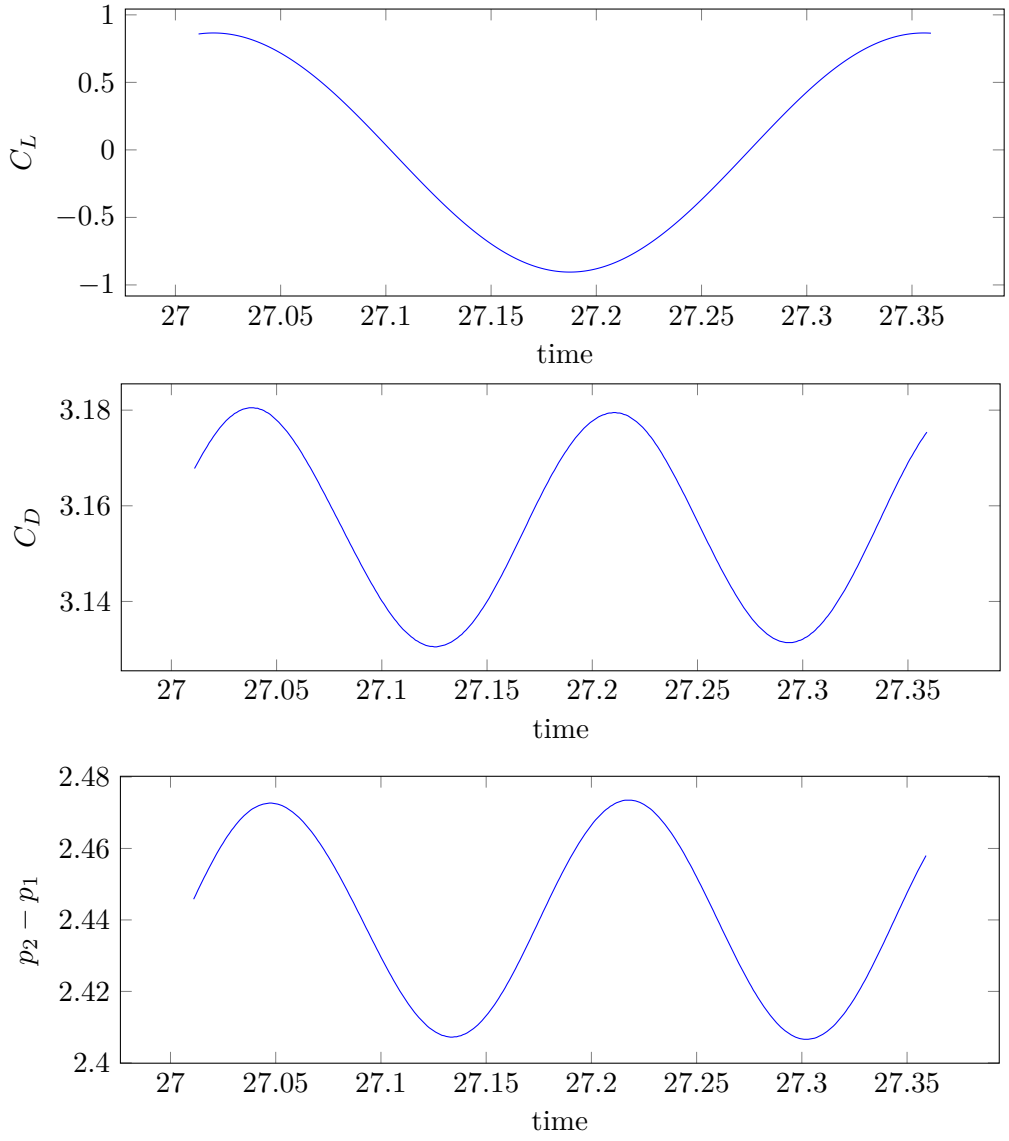

Figure 5.6: Top to bottom : Lift coefficient, drag coefficient and pressure drop for 1 cycle.

CHAPTER 6

DARCY FLOW IN POROUS MEDIA

Flow through a porous medium can be described by *Darcy's Law*. Originally determined from experiment by the French engineer Henry Darcy in 1856 to describe water flowing through sands, it has since been derived from first principles by applying a technique called volume averaging to the Stokes equations [10] [11]. In chapter we present equations for Darcy flow and a finite element discretization method. As with the Stokes and Navier-Stokes equations this will be a mixed finite element method, however unlike those equations the Taylor-Hood element pair does not guarantee existence and uniqueness. Rather, we will use the Raviart-Thomas ($\mathcal{RT}$) element for the velocity. Results concerning the RT element are provided and a convergence study on the Darcy equations is done. In the remaining sections a case study is presented that combines Hooke's Law with the Darcy equations to investigate how fluid stresses deform a polystyrene foam.

6.1 Governing Equations

The equations for Darcy flow in a domain Ω with boundary $\Gamma = \Gamma_1 \cup \Gamma_2$ are given by:

$$\frac{\nu}{\kappa(\boldsymbol{x})}\boldsymbol{u} + \nabla p = \boldsymbol{f}(\boldsymbol{x}) \qquad \text{in } \Omega \tag{6.1a}$$

$$\nabla \cdot \boldsymbol{u} = 0 \qquad \text{in } \Omega \tag{6.1b}$$

$$p = g(\boldsymbol{x}) \qquad \text{on } \Gamma_1 \tag{6.1c}$$

$$\boldsymbol{u} \cdot \boldsymbol{n} = 0 \qquad \text{on } \Gamma_2 \tag{6.1d}$$

where again $\boldsymbol{u}$ is the fluid velocity, p is the fluid pressure and ν is the kinematic viscosity of the fluid. The functions $\boldsymbol{f}(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are given as is the permeability $\kappa(\boldsymbol{x})$.

6.2 Weak Formulation

6.2.1 Galerkin Weak Formulation

Define the space:

$$\mathbf{H}_{\text{div},0} =: \{\boldsymbol{v} \in \mathbf{L}^2(\Omega) : \nabla \cdot \boldsymbol{v} \in L^2(\Omega) \text{ and } \boldsymbol{v} \cdot \boldsymbol{n} = 0 \text{ on } \Gamma_2\}$$

Taking the inner product of (6.1a) with a test function $v \in \mathbf{H}_{\text{div},0}$ and integrating the second term by parts we get:

$$(\frac{\nu}{\kappa}\boldsymbol{u}, \boldsymbol{v}) - (p, \nabla \cdot \boldsymbol{v}) = (\boldsymbol{f}, \boldsymbol{v}) - \oint_\Gamma p\boldsymbol{v} \cdot \boldsymbol{n} \mathrm{d}\Gamma.$$

Or equivalently:

$$a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) + d(\boldsymbol{v})$$

where:

- $a(\boldsymbol{u}, \boldsymbol{v}) = \nu \int_\Omega \frac{1}{\kappa}\boldsymbol{u} \cdot \boldsymbol{v} \mathrm{d}\Omega$
- $b(\boldsymbol{v}, p) = -\int_\Omega p \nabla \cdot \boldsymbol{v} \mathrm{d}\Omega$
- $d(\boldsymbol{v}) = -\int_{\Gamma_1} g\boldsymbol{v} \cdot \boldsymbol{n} \mathrm{d}\Gamma$

Taking the inner product of (6.1b) with a test function $q \in L^2(\Omega)$ yields:

$$b(\boldsymbol{u}, q) = 0.$$

Putting this all together yields the Galerkin weak problem we have to solve. Find $\boldsymbol{u} \in \mathbf{H}_{\text{div}}(\Omega)$, $p \in L^2(\Omega)$ such that:

$$\begin{aligned} a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) &= (\boldsymbol{f}, \boldsymbol{v}) + d(\boldsymbol{v}) && \text{for all } \boldsymbol{v} \in \mathbf{H}_{\text{div},0} \\ b(\boldsymbol{u}, q) &= 0 && \text{for all } q \in L^2(\Omega) \end{aligned}$$

6.2.2 Discrete Weak Formulation

As in sections 4.2.2 and 5.3.2 we will choose finite dimensional subspaces $\mathbf{V}^h$ and S^h of the infinite dimensional subspaces for $\boldsymbol{u}$ and p in which to compute a solution $\boldsymbol{u}^h$ and p^h. Our discrete problem will then read: seek $\boldsymbol{u}^h \in \mathbf{H}_{\text{div}}(\Omega)$, $p^h \in L^2(\Omega)$ such that:

$$\begin{aligned} a(\boldsymbol{u}^h, \boldsymbol{v}^h) + b(\boldsymbol{v}^h, p^h) &= (\boldsymbol{f}, \boldsymbol{v}^h) + d(\boldsymbol{v}^h) && \text{for all } \boldsymbol{v}^h \in \mathbf{V}^h \\ b(\boldsymbol{u}^h, q^h) &= 0 && \text{for all } q^h \in S^h \end{aligned} \tag{6.2}$$

As with the Stokes and Navier-Stokes equations, $S^h \in L^2(\Omega)$, however unlike the previous cases, $\mathbf{V}^h \notin \mathbf{H}^1_0(\Omega)$, but rather $\mathbf{V}^h \in \mathbf{H}_{\text{div},0}(\Omega)$.

Once we have decided on $\mathbf{V}^h$ and S^h we can turn (6.2) into a matrix problem by denoting $\{\boldsymbol{v}_k(\boldsymbol{x})\}, k = 1, \cdots, K$ and $\{q_j(\boldsymbol{x})\}, j = 1, \cdots, J$ as the basis for $\mathbf{V}^h$ and S^h respectively. Then as before, we have:

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{k=1}^{K} \alpha_k \boldsymbol{v}_k(\boldsymbol{x}) \qquad p^h(\boldsymbol{x}) = \sum_{j=1}^{J} \beta_j q_j(\boldsymbol{x})$$

and we can rewrite (6.2) as:

$$\begin{aligned} \sum_{k=1}^{K} \alpha_k a(\boldsymbol{v}_k, \boldsymbol{v}_\ell) + \sum_{j=1}^{J} \beta_j b(q_j, \boldsymbol{v}_\ell) &= (\boldsymbol{f}, \boldsymbol{v}_\ell) + d(\boldsymbol{v}_\ell) \quad \text{for } \ell = 1, \cdots, K \\ \sum_{k=1}^{K} \alpha_k b(\boldsymbol{v}_k, q_i) &= 0 \qquad\qquad\qquad\quad \text{for } i = 1, \cdots, J \end{aligned} \tag{6.3}$$

which we can solve for the unknown coefficients $\{\alpha_k\}$ and $\{\beta_j\}$. Assuming Γ_1 is not empty, we will have some boundary conditions on p, so this system will not be singular.

6.3 Finite Element Spaces

We saw in section 4.3.1 that the Taylor-Hood element pair satisfies the conditions necessary for a unique solution for the discrete weak form of the Stokes equations (4.5). It turns out that in general the Taylor-Hood element pair does not satisfy all the conditions for the weak form of the Darcy equations.

Recall that in order to guarantee a unique solution to the weak formulation we must have:

1. $(\boldsymbol{f}, \boldsymbol{v}^h)$ is bounded for all $\boldsymbol{v}^h \in \mathbf{V}^h$
2. $a(\boldsymbol{u}^h, \boldsymbol{v}^h)$ is bounded and coercive for all $\boldsymbol{u}^h, \boldsymbol{v}^h \in \mathbf{V}^h$
3. $b(\boldsymbol{v}^h, q^h)$ satisfies the LBB condition (4.7)

For Stokes flow it turns out that these conditions, with the exception of the LBB condition are all satisfied for any $\mathbf{V}^h \in \mathbf{H}^1_0(\Omega)$ and $S^h \in L^2(\Omega)$. The Taylor-Hood element pair then satisfied the LBB condtion. For the Darcy flow equations, it is no longer true that $a(\cdot, \cdot)$ is coercive for any $\mathbf{V}^h \in \mathbf{H}_{\text{div},0}$. The Taylor-Hood element pair requires too much continuity on the velocity and therefore cannot be used.

A popular choice of elements for Darcy flow problems is the Raviart-Thomas ($\mathcal{RT}$) element for velocity and discontinuous linear elements for the pressure.

6.3.1 Raviart-Thomas Elements

In order for a function $\boldsymbol{v}$ to be in $\mathbf{H}_{\text{div}}$ we have to show that $\nabla \cdot \boldsymbol{v}$ exists and that it is in $L^2(\Omega)$. For the divergence of a function to exist (in a weak sense), there must exist a function $q \in L^2(\Omega)$ such that:

$$\int_\Omega \boldsymbol{v} \cdot \nabla \phi \mathrm{d}\Omega = -\int_\Omega q\phi \mathrm{d}\Omega \qquad \text{for all } \phi \in C_0^\infty.$$

Let $\mathcal{J}_h$ be a quadrilateral mesh of $\bar{\Omega}$ where $\bar{\Omega}$ is an approximation to Ω. We can apply Green's identity over each element $\square \in \mathcal{J}_h$ with boundary $\delta\square$:

$$\begin{aligned}
\int_\Omega \boldsymbol{v} \cdot \nabla \phi d\Omega &= \sum_{\square \in \mathcal{J}_h} \int_\square \boldsymbol{v} \cdot \nabla \phi \mathrm{d}\square \\
\text{Green's identity } \Rightarrow &= -\sum_{\square \in \mathcal{J}_h} \int_\square \nabla \cdot \boldsymbol{v} \phi \mathrm{d}\square + \sum_{\square \in \mathcal{J}_h} \int_\square (\boldsymbol{v} \cdot \boldsymbol{n})|_{\delta\square} \phi \mathrm{d}\square \\
&= -\sum_{\square \in \mathcal{J}_h} \int_\square \nabla \cdot \boldsymbol{v} \phi \mathrm{d}\square + \sum_{e \in \delta\square} \int_e [(\boldsymbol{v} \cdot \boldsymbol{n})|e] \phi \mathrm{d}e
\end{aligned}$$

where $[(\boldsymbol{v} \cdot \boldsymbol{n})|e]$ is the jump of $\boldsymbol{v} \cdot \boldsymbol{n}$ across edge e. If this is 0, meaning the normal component of $\boldsymbol{v}$ is continuous across each element boundary, then we have the required conditions for a weak divergence to exist. This weak divergence is given by:

$$\nabla \cdot \mathbf{v} = \sum_{\square \in \mathcal{J}_h} \nabla \cdot \boldsymbol{v},$$

i.e. the sum of the strong divergence over each element. This is in $L^2(\Omega)$ because over each element we know that $\boldsymbol{v}$ is a polynomial.

To define the $\mathcal{RT}$ element on a quadrilateral in $\mathbb{R}^2$ [1] we will first define the space:

$$\mathcal{RT}_{[k]}(\square) = P_{k,+1,k}(\square) \times P_{k,k+1}(\square), \tag{6.4}$$

where $P_{k,\ell}$ is defined to be all polynomials of degree $\leq k$ in x and $\leq \ell$ in y. A basis for $\mathcal{RT}_{[k]}$ is given by:

$$\left\{ \begin{pmatrix} x^i y^j \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ x^j y^i \end{pmatrix} \right\}, \qquad 0 \leq i \leq k+1, 0 \leq j \leq k.$$

The dimension of this basis is $2(k+1)(k+2)$.

The degrees of freedom Θ are given by:

$$\int_\Gamma p_k \boldsymbol{\phi} \cdot \boldsymbol{n} \mathrm{d}\Gamma_i \qquad p_k \in P_k(\Gamma) \text{ for each edge of } \square \tag{6.5}$$

$$\int_\Omega \boldsymbol{\phi} \cdot \boldsymbol{p}_k \mathrm{d}\Omega \qquad \boldsymbol{p}_k \in \boldsymbol{\Psi}_k(\square) \tag{6.6}$$

where $\mathbf{\Psi}_k$ is defined as:

$$\mathbf{\Psi}_k(\square) = P_{k-1,k} \times P_{k,k-1}.$$

Appendix A provides an explicit derivation of the basis functions for $\mathcal{RT}_{[0]}$ and $\mathcal{RT}_{[1]}$.

If we use $\mathcal{RT}$ elements of order k along with discontinuous linear pressure elements, then as long as the actual solution $\boldsymbol{u} \in \mathbf{H}^k(\Omega)$ the following error estimates hold [7]:

$$||\boldsymbol{u} - \boldsymbol{u}^h||_{\text{div}} + ||p - p^h||_0 \leq Ch^k||p||_{k+1}$$
$$||\boldsymbol{u} - \boldsymbol{u}^h||_0 \leq h^{k+1}||\boldsymbol{u}||_{k+1}$$
$$||p - p^h||_0 \leq Ch^{k+1}||p||_{\max\{k+1,2\}}$$

6.4 Implementation

The Raviart-Thomas element is implemented in deal.ii in the class **FE_RaviartThomas**.

```
template<int dim>
DarcyFlow<dim>::DarcyFlow():
    fe (FE_RaviartThomas<dim>(1), 1, FE_DGQ<dim>(1), 1),
    dof_handler(mesh)
{}
```

Using this element requires little unexpected changes to our code. To apply the boundary conditions we now use the function *VectorTools::project_boundary_values_div_conforming* on the boundaries where we prescribe $\boldsymbol{u} \cdot \boldsymbol{n}$.

```
//apply through boundary conditions on top and bottom
VectorTools::project_boundary_values_div_conforming(dof_handler, 0,
                        *boundary_values, 1, constraints);
VectorTools::project_boundary_values_div_conforming(dof_handler, 0,
                        *boundary_values, 2, constraints);
```

6.4.1 Convergence Study

To test our code we turn again to the exact solution (4.11); unlike the previous cases however we choose the exact solution for p:

$$p(x, y) = \sin(\pi x).$$

This is done so that we have constant pressure on the left and right sides of the unit square. Using these solutions we can create boundary conditions $\boldsymbol{u} \cdot \boldsymbol{n}$ on the top and bottom of the unit square

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	208	6.068×10^{-1}	2.341×10^{-1}	-	9.080×10^{-1}	-
64	800	1.590×10^{-1}	5.866×10^{-2}	1.99	4.561×10^{-1}	0.99
256	3136	4.022×10^{-2}	1.474×10^{-2}	2.00	2.283×10^{-1}	1.00

Table 6.1: Convergence rates for $\boldsymbol{u}(\boldsymbol{x})$ for Darcy flow using $\mathcal{RT}_1$ elements for the velocity and discontinuous P_1 elements for the pressure.

# cells	# unknowns	$\mathbf{L}^\infty$ error	$\mathbf{L}^2$ error	$\mathbf{L}^2$ convergence	$\mathbf{H}^1$ error	$\mathbf{H}^1$ convergence
16	208	4.845×10^{-2}	3.226×10^{-2}	-	9.980×10^{-1}	-
64	800	1.266×10^{-2}	8.112×10^{-3}	1.99	5.025×10^{-1}	0.99
256	3136	3.201×10^{-3}	2.031×10^{-3}	2.00	2.517×10^{-1}	1.00

Table 6.2: Convergence rates for $p(\boldsymbol{x})$ for Darcy flow using $\mathcal{RT}_1$ elements for the velocity and discontinuous P_1 elements for the pressure.

as well as an appropriate forcing function. This leads to the convergence tables 6.1 and 6.2 which demonstrate the desired convergence rates.

6.5 Case Study : Foam Deformation

Polystyrene foam infused with resin is viewed as a potential way to control placement of additives inside a larger structure. For example if one wanted to mechanically reinforce one particular area of an aeroplane fuselage, one could place a foam containing carbon nanotubes at that point before resin infusion. In many applications, such as the one mentioned, it is important that the surface of the hardened resin remains smooth. It has been experimentally shown that in the area above the foam this is not the case, with bumps that can be around 10% of the foam height. See figure 6.1 for an SEM image of a piece of deformed foam.

One hypothesis that might explain this phenomenon relates to inhomogeneity in the foam porosity. As fluid flows through the foam the nonuniform velocity and pressure fields exert stresses on the foam. These stresses can be converted to strains, and theses strains used to calculate the deformation of the foam.

6.5.1 Problem Description

Consider a cross section of a porous foam as shown in figure 6.2. We prescribe pressure boundary conditions on the inflow and outflow and no-through boundary conditions on the velocity on the top and bottom.

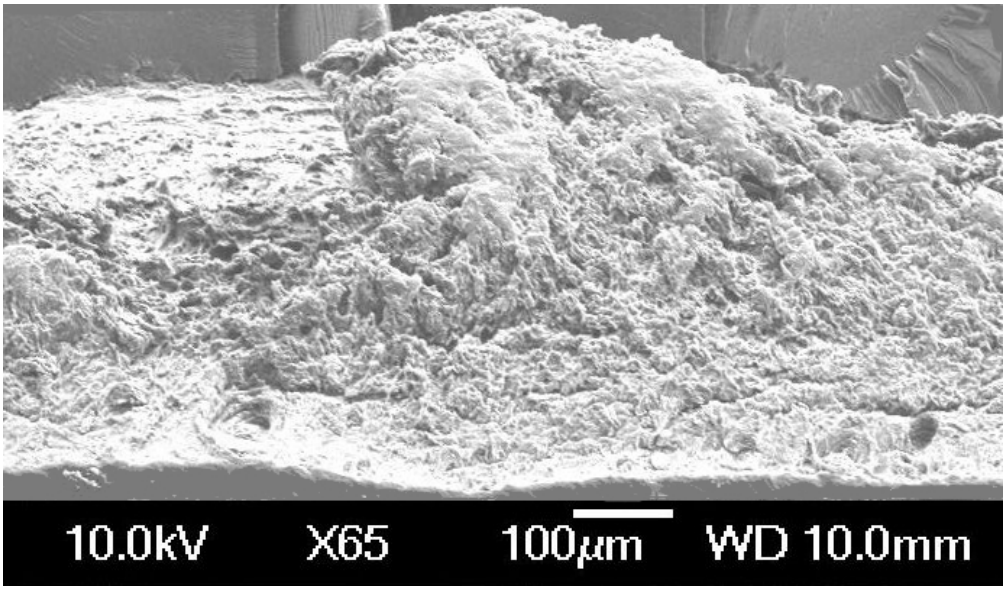

Figure 6.1: Cross section of a piece polystyrene foam after resin infusion.

Stress. We can solve (6.2) to get a pressure and velocity profile. From this, a stress tensor field can be calculated:

$$\boldsymbol{\sigma} = -p\boldsymbol{I} + \mu\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T\right).$$

This can be split into components:

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{pmatrix}$$

where:

$$\begin{aligned} \sigma_{xx} &= -p + 2\mu\frac{\partial u}{\partial x} \\ \sigma_{yy} &= -p + 2\mu\frac{\partial v}{\partial y} \\ \sigma_{xy} = \sigma_{yx} &= \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \end{aligned}$$

Note that σ_{ij} is the force per unit area acting in coordinate direction i on a plane perpendicular to coordinate direction j. This means that σ_{xx} and σ_{yy} are normal stresses, while $\sigma_{xy} = \sigma_{yx}$ are shear stresses.

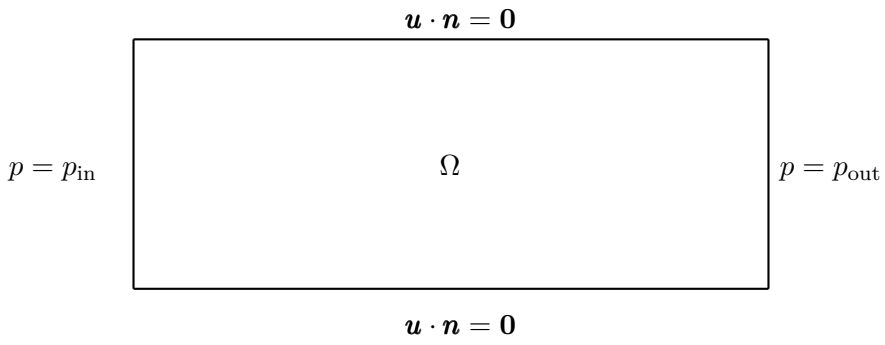

Figure 6.2: Initial cross section of porous foam.

Strain. Strain measures the deformation of an object relative to a reference length. Given a displacement field $\boldsymbol{\xi}(\boldsymbol{x}) = \langle \alpha, \beta \rangle$, the strain tensor field $\boldsymbol{\epsilon}$ can be calculated as:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\nabla \boldsymbol{\xi} + (\nabla \boldsymbol{\xi})^T\right) = \begin{pmatrix} \epsilon_{xx} & \epsilon_{xy} \\ \epsilon_{yx} & \epsilon_{yy} \end{pmatrix}$$

where:

$$\epsilon_{xx} = \frac{\partial \alpha}{\partial x} \tag{6.7a}$$

$$\epsilon_{yy} = \frac{\partial \beta}{\partial y} \tag{6.7b}$$

$$\epsilon_{xy} = \epsilon_{yx} = \frac{1}{2}\left(\frac{\partial \alpha}{\partial y} + \frac{\partial \beta}{\partial x}\right) \tag{6.7c}$$

Hooke's Law. The relationship between stress and strain can be described in the linear elastic region in figure 6.3 by Hooke's Law. In 1D, Hooke's Law states:

$$\sigma = E\epsilon,$$

where E is the elastic (Young's) modulus of the material. This modulus is in fact the constant slope of the stress-strain curve in the linear elastic region. For 2D isotropic materials, Hooke's Law can be expressed as:

$$\begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \end{pmatrix} = \frac{1}{E}\begin{pmatrix} 1 & -\mu & 0 \\ -\mu & 1 & 0 \\ 0 & 0 & 2(1+\mu) \end{pmatrix}\begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix} \tag{6.8}$$

where E is again the elastic modulus of the material and μ is its Poisson ratio. (In most literature ν is the Poisson ratio, however we have already used that for the kinematic viscosity.)

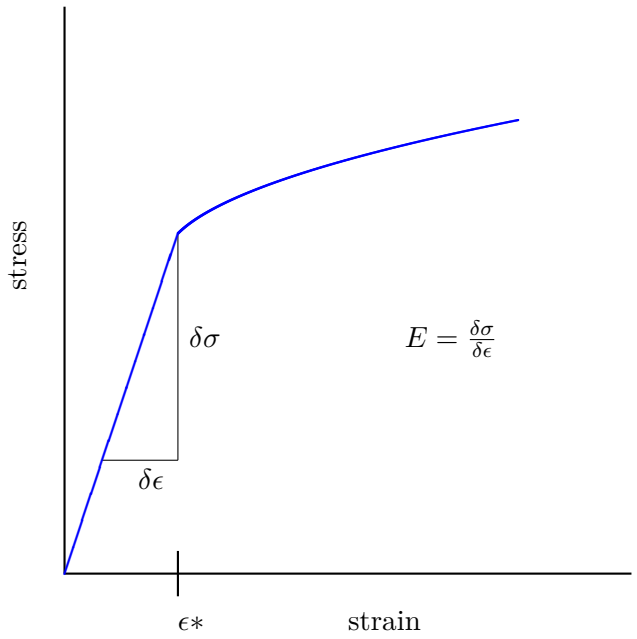

Figure 6.3: An example of a stress-strain curve for a solid material. Hooke's Law is valid in the low strain linear region, $\epsilon < \epsilon*$.

Displacement. If we are only interested in vertical displacement (i.e. finding the ridges at the top of the foam), then all we need to do is integrate (6.7b):

$$\frac{\partial \beta}{\partial y} = \epsilon_{yy} \Rightarrow \beta(x, y) = \int_0^y \epsilon_{yy}(x, Y)dY + c(x).$$

One assumption we can make is that the bottom of the foam stays fixed, i.e. $\beta(x, 0) = 0$. Doing this means that $c(x) = 0$, and we get:

$$\beta(x, y) = \int_0^y \epsilon_{yy}(x, Y)dY.$$

Plugging in ϵ_{yy} from (6.8) yields:

$$\beta = \frac{1}{E}\int_0^y \sigma_{yy} - \nu\sigma_{xx}dY \tag{6.9}$$

Porosity. To generate a porosity field $\phi(\boldsymbol{x})$, we take N circular pores at random locations $\boldsymbol{x}_i$, $i = 1, \cdots, N$ and give each one a random inner radius γ_i^{in} and a random outer radius γ_i^{out}. For each pore inside its inner radius the porosity is 1 (full void) and outside its porosity is 0. Between

its inner and its outer radii the porosity decreases linearly from 1 to 0. At each point $\boldsymbol{x} \in \Omega$, the porosity ϕ is determined by summing all the porosities of each pore. Mathematically this can be described as:

$$\phi(\boldsymbol{x}) = \min\left(\max\left(\sum_{i=1}^{N} \eta_i(\boldsymbol{x}), \phi_{\min}\right), \phi_{\max}\right)$$

where

$$\eta_i(\boldsymbol{x}) = \begin{cases} 1 & \text{if } |\boldsymbol{x} - \boldsymbol{x}_i| \leq \gamma_i^{\text{in}} \\ \frac{|\boldsymbol{x}-\boldsymbol{x}_i| - \gamma_i^{\text{out}}}{\gamma_i^{\text{in}} - \gamma_i^{\text{out}}} & \text{if } \gamma_i^{\text{in}} \leq |\boldsymbol{x} - \boldsymbol{x}_i| \leq \gamma_i^{\text{out}} \\ 0 & \text{otherwise} \end{cases}$$

Permeability. The relationship between permeability and porosity can be quite complicated, however we will assume an isotropic permeability with the simple relationship:

$$\kappa(\boldsymbol{x}) = A\phi(\boldsymbol{x})^B,$$

where A and B are constants. Note that since the foam is isotropic, we can take the permeability to be a scalar.

Local elastic modulus. The local elastic modulus will depend upon the local porosity. As the porosity increases, we expect the modulus to decrease. We will assume the linear relationship:

$$E(\boldsymbol{x}) = \bar{E}\frac{1 - \phi(\boldsymbol{x})}{1 - \bar{\phi}},$$

where $\bar{E}$ and $\bar{\phi}$ are the bulk elastic modulus and porosity respectively.

6.5.2 Results

The first step is to generate the porosity field. To do so we use the following parameters:

initial foam height	1 mm
foam length	5 mm
N	30
r_{in}	$[0.001 \text{ mm}, 0.03 \text{ mm}]$
r_{out}	$[0.05 \text{ mm}, 0.12 \text{ mm}]$
$\phi_{\min}$	1×10^{-3}
$\phi_{\max}$	0.95

This gives the porosity field shown in figure 6.4, with an average porosity of 0.672.

To solve the remaining problem, consider the following parameters:

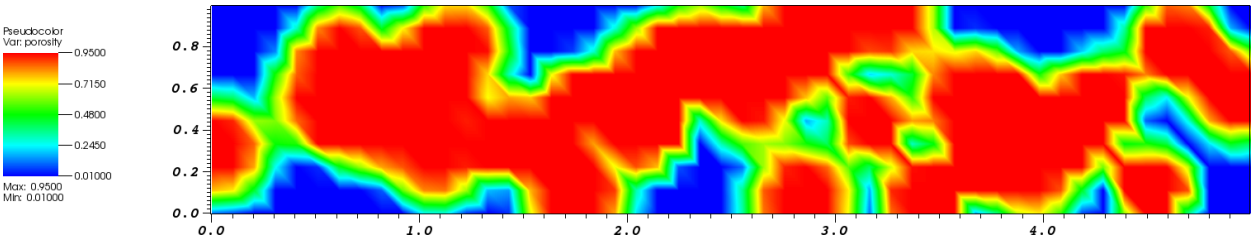

Figure 6.4: Porosity distribution for the first iteration.

ν	10 Pa s
μ	0.1
A	2×10^{-4}
B	2
$\bar{E}$	1×10^5 Pa
p_{in}	1×10^5 Pa
p_{out}	9.9×10^4 Pa

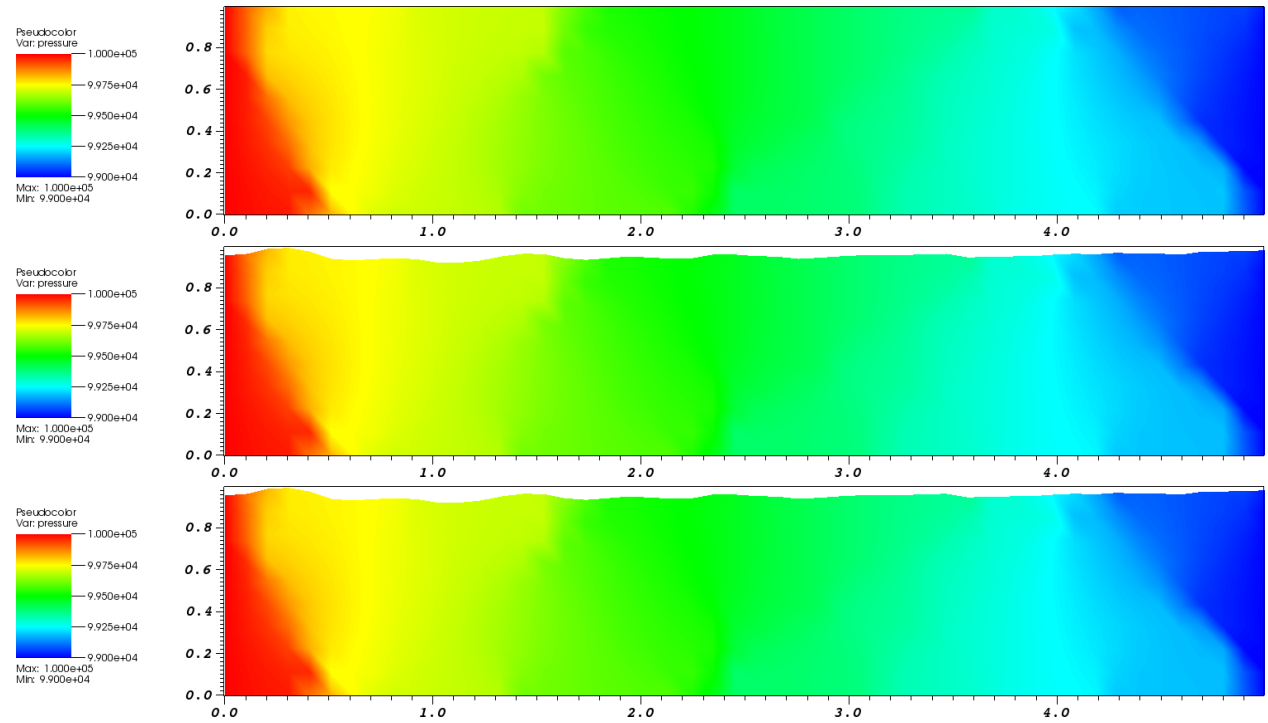

Figure 6.5: Pressure distribution for the first 3 iterations.

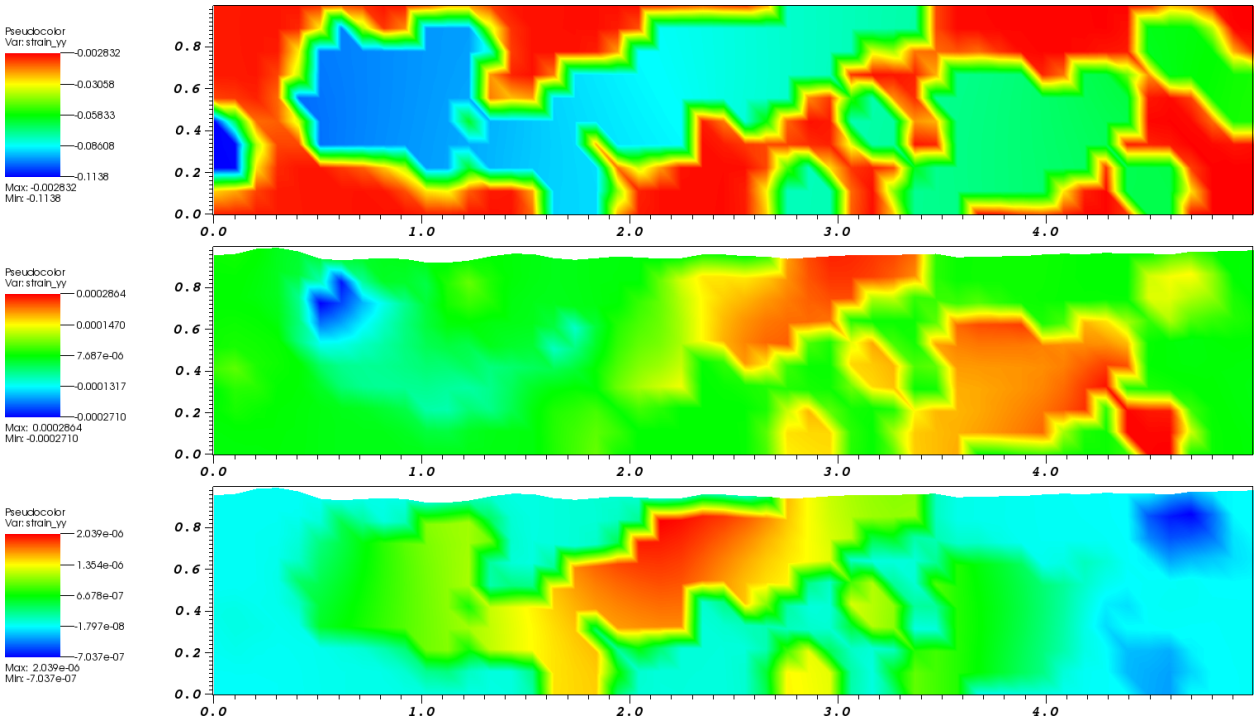

Figure 6.6: ϵ_{yy} distribution for the first 3 iterations.

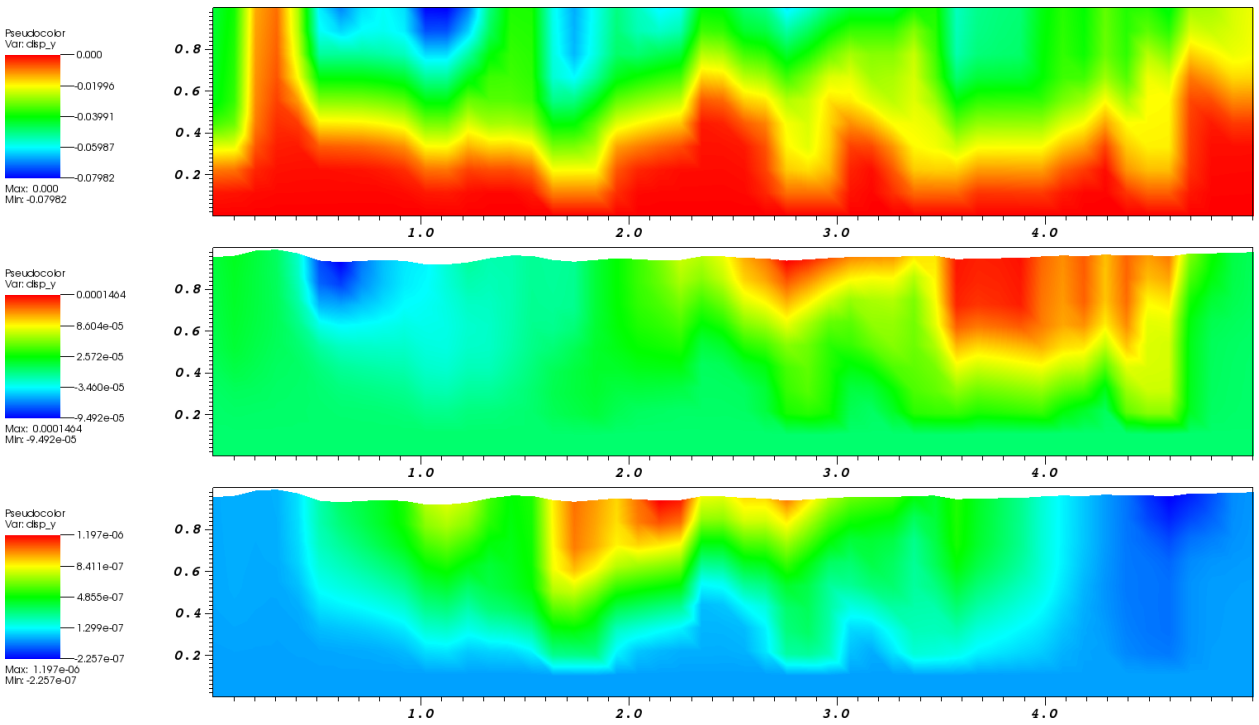

Figure 6.7: Vertical displacement distribution for the first 3 iterations.

The average displacement of the top of the foam for the first 3 iterations are: $\{6.94\times10^{-3}\text{mm}, 1.20\times 10^{-5}\text{mm}, 8.57\times10^{-8}\text{mm}\}$. We can conclude therefore that our iteration scheme converges very quickly.

To compare with experimental results, we will use the same parameters, but with $\bar{E} = 1\times10^{6}$ Pa. Using this stiffer foam we see average displacements of $\{6.96\times10^{-4}\text{mm}, 1.17\times10^{-7}\text{mm}, 8.59\times 10^{-11}\text{mm}\}$ for the first 3 iterations. This indicates that although the fluid stresses have an impact on the foam shape, our model underestimates the size of the bumps. Sensitivity studies can show us the general trends our model predicts. To start with, using the same base parameters as above, we will look at the sensitivity of the Young's modulus.

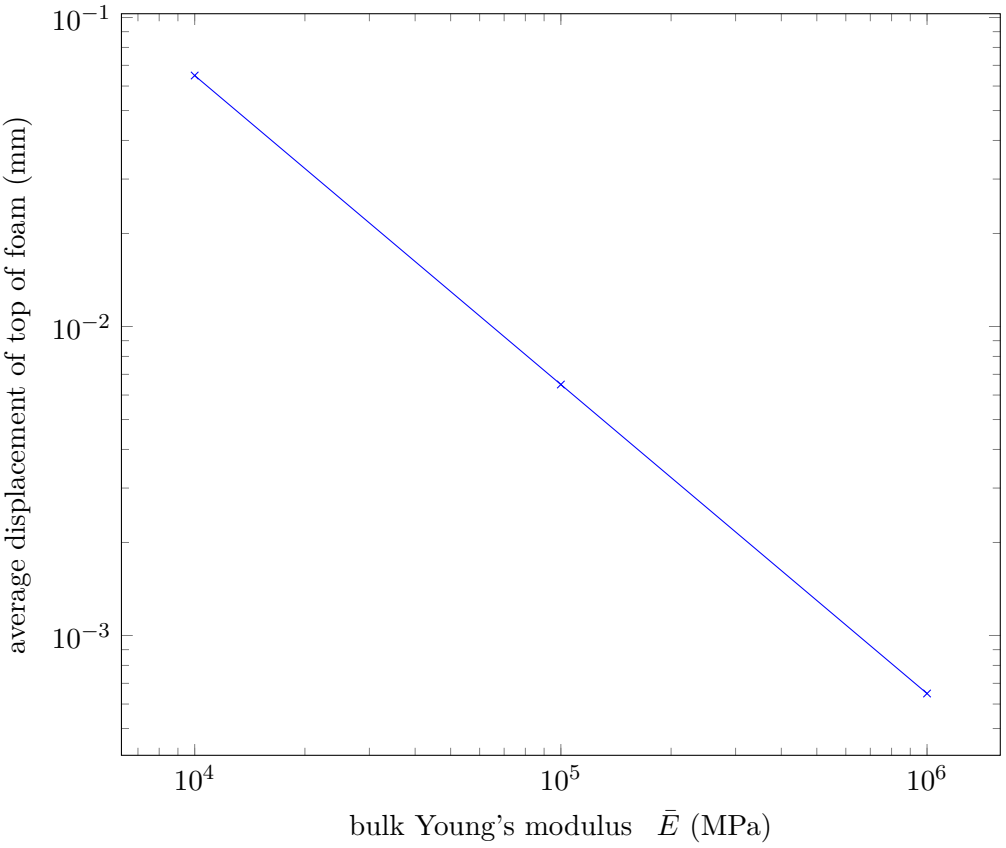

Figure 6.8: Sensitivity of bulk Young's modulus

Figure 6.8 shows that as expected from Hooke's law, the displacement scales as $1/\bar{E}$. Perhaps a more interesting sensitivity study is the sensitivity to the bulk porosity $\bar{\phi}$. Bulk porosity is not an

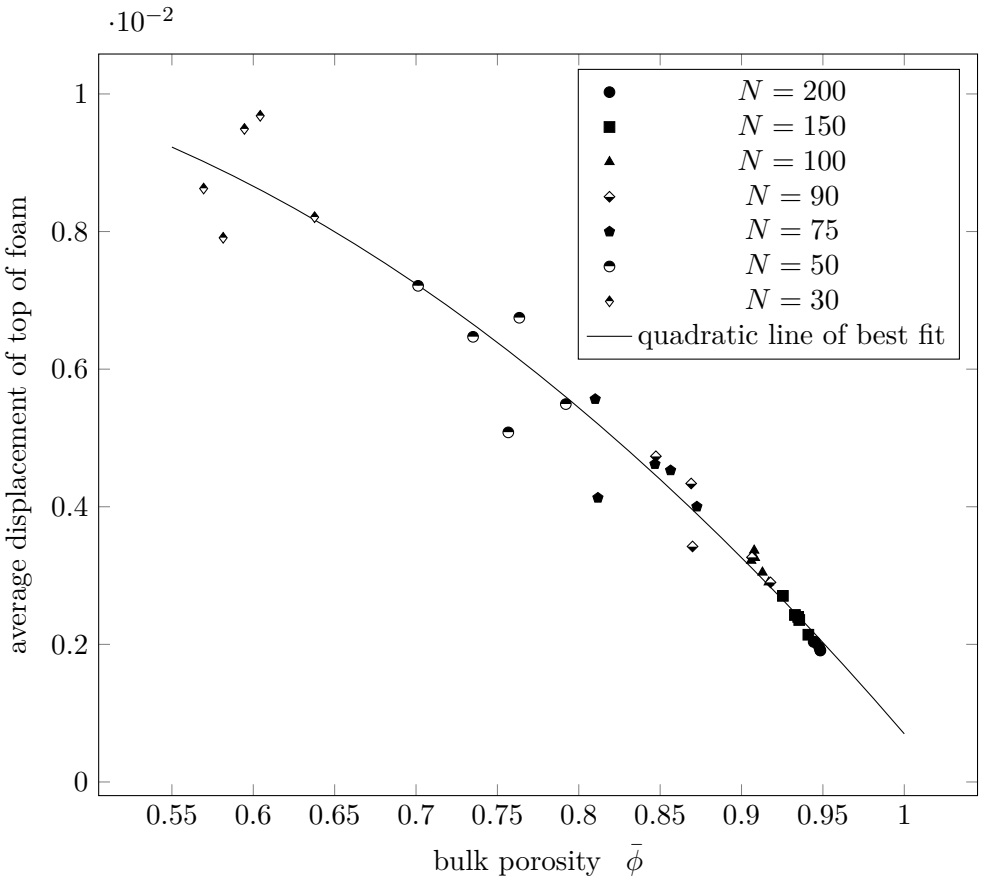

Figure 6.9: Sensitivity of bulk porosity

input to our model, however one way to control it is by varying the number of pores N. Since the porosity field is actually random, we will run the simulation 5 times for each N.

As can be seen in figure 6.9, as the bulk porosity decreases, the average displacement actually increases. This is due to the fact that local modulus depends upon the porosity. If the foam is very uniform, then the local modulus is very close to the bulk modulus everywhere so we don't get low local modulus anywhere in the foam. This limits the deformation at high bulk porosities.

A final sensitivity study is looking at the effect of the maximum porosity $\phi_{\max}$ as shown in figure 6.10.This study shows that the displacement is very sensitive to the maximum porosity.

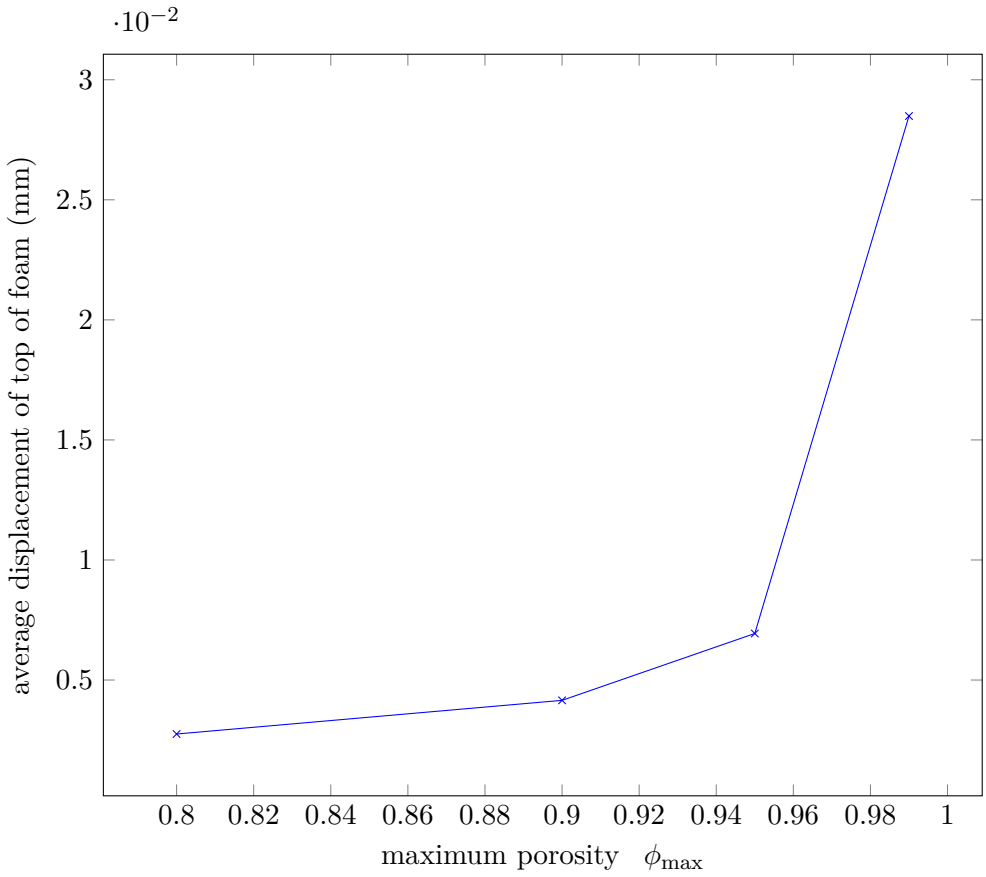

Figure 6.10: Sensitivity of maximum porosity

APPENDIX A

EXPLICIT FORM OF RAVIART-THOMAS ELEMENTS

In this appendix we derive the explicit basis functions for $\mathcal{RT}_0$ and $\mathcal{RT}_1$ on the reference square.

1. $\mathbf{k = 0}$:

   For $k = 0$, our basis functions $\mathcal{RT}_{[0]}(\square)$ will be of the form:

$$\begin{aligned}\boldsymbol{\phi}_i &= P_0(\square) + \boldsymbol{x}P_0(\square) \\ &= \begin{pmatrix} \mathrm{a}_1 \\ \mathrm{b}_1 \end{pmatrix} + \begin{pmatrix} \mathrm{a}_2 x \\ \mathrm{b}_2 y \end{pmatrix}\end{aligned}$$

   For $k = 0$, $\boldsymbol{\Psi}_k(\square)$ is empty, so we only need to consider the edge degrees of freedom given by (6.5). The integral along the boundary can be broken up along each edge, as labeled in figure A.1. For each of the four edges, $P_0(\Gamma)$ is 1. So we have 4 integrals to calculate for each basis function. For $\boldsymbol{\phi}_1$, we want the normal derivative to be 1 along E_1 and 0 along the other edges. This gives:

   (a) $E_1 : \boldsymbol{n} = \hat{\boldsymbol{i}}$.

$$\Rightarrow \int_{-1}^{1} \mathrm{a}_1 + \mathrm{a}_2 x \mathrm{d}y = 2(\mathrm{a}_1 + \mathrm{a}_2) = 1$$

   (b) $E_2 : \boldsymbol{n} = \hat{\boldsymbol{j}}$.

$$\Rightarrow \int_{1}^{-1} \mathrm{b}_1 + \mathrm{b}_2 y \mathrm{d}x = -2(\mathrm{b}_1 + \mathrm{b}_2) = 0$$

   (c) $E_3 : \boldsymbol{n} = -\hat{\boldsymbol{i}}$.

$$\Rightarrow \int_{1}^{-1} -(\mathrm{a}_1 + \mathrm{a}_2 x) \mathrm{d}y = 2(\mathrm{a}_1 - \mathrm{a}_2) = 0$$

   (d) $E_4 : \boldsymbol{n} = -\hat{\boldsymbol{j}}$.

$$\Rightarrow \int_{-1}^{1} -(\mathrm{b}_1 + \mathrm{b}_2 y) \mathrm{d}x = -2(\mathrm{b}_1 - \mathrm{b}_2) = 0$$

   This can be put into a linear system to solve for the coefficients:

$$\begin{pmatrix} 2 & 2 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 2 & -2 & 0 & 0 \\ 0 & 0 & -2 & 2 \end{pmatrix} \begin{pmatrix} \mathrm{a}_1 \\ \mathrm{a}_2 \\ \mathrm{b}_1 \\ \mathrm{b}_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

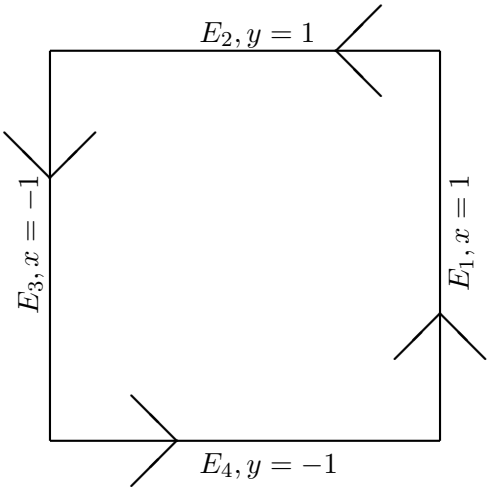

Figure A.1: The reference square.

which can be solved to give:

$$\boldsymbol{\phi_1} = \frac{1}{4}\begin{pmatrix} 1+x \\ 0 \end{pmatrix}$$

To calculate the other 3 basis functions, the matrix will stay the same, but the right hand side will change. The other basis functions are:

$$\boldsymbol{\phi_2} = -\frac{1}{4}\begin{pmatrix} 0 \\ 1+y \end{pmatrix} \qquad \boldsymbol{\phi_3} = \frac{1}{4}\begin{pmatrix} 1-x \\ 0 \end{pmatrix} \qquad \boldsymbol{\phi_4} = \frac{1}{4}\begin{pmatrix} 0 \\ y-1 \end{pmatrix}$$

2. $\mathbf{k = 1}$:

For $k = 1$, $\mathcal{RT}_{[1]}(\square) = P_1(\square) + \boldsymbol{x}P_1(\square)$. Recall that P_1 contains linear combinations of:

$$\{1, x, y, xy\}$$

Therefore the basis functions for $\mathcal{RT}_{[1]}(\square)$ will be of the form:

$$\boldsymbol{\phi}_i = \begin{pmatrix} \mathrm{a}_1 + \mathrm{a}_2 x + \mathrm{a}_3 y + \mathrm{a}_4 xy + \mathrm{a}_5 x^2 + \mathrm{a}_6 x^2 y \\ \mathrm{b}_1 + \mathrm{b}_2 x + \mathrm{b}_3 y + \mathrm{b}_4 xy + \mathrm{b}_5 y^2 + \mathrm{b}_6 xy^2 \end{pmatrix},$$

which has 12 unknown coefficients. The space $\boldsymbol{\Psi}_1(\square)$ now contains the functions:

$$\boldsymbol{\Psi}_1 = P_{0,1} \times P_{1,0} = \{1, y\} \times \{1, x\}$$

which has a basis:

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} y \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ x \end{pmatrix} \right\}$$

Again we will start by doing the line integrals over each edge.

(a) $E_1 : \boldsymbol{n} = \hat{\boldsymbol{i}}$, $x = 1$. We will have to consider $p_k = 1$ and $p_k = y$.

i. $p_k = 1$:

$$\Rightarrow \int_{-1}^{1} \mathrm{a}_1 + \mathrm{a}_2 + \mathrm{a}_3 y + \mathrm{a}_4 y + \mathrm{a}_5 + \mathrm{a}_6 y \mathrm{d}y = 2(\mathrm{a}_1 + \mathrm{a}_2 + \mathrm{a}_5) = 1$$

ii. $p_k = y$:

$$\Rightarrow \int_{-1}^{1} y(\mathrm{a}_1 + \mathrm{a}_2 + \mathrm{a}_3 y + \mathrm{a}_4 y + \mathrm{a}_5 + \mathrm{a}_6 y)\mathrm{d}y = \frac{2}{3}(\mathrm{a}_3 + \mathrm{a}_4 + \mathrm{a}_6) = 0$$

(b) $E_2 : \boldsymbol{n} = \hat{\boldsymbol{j}}$, $y = 1$. We will have to consider $p_k = 1$ and $p_k = x$.

i. $p_k = 1$:

$$\Rightarrow \int_{1}^{-1} \mathrm{b}_1 + \mathrm{b}_2 x + \mathrm{b}_3 + \mathrm{b}_4 x + \mathrm{b}_5 + \mathrm{b}_6 x \mathrm{d}x = -2(\mathrm{b}_1 + \mathrm{b}_3 + \mathrm{b}_5) = 0$$

ii. $p_k = y$:

$$\Rightarrow \int_{1}^{-1} x(\mathrm{b}_1 + \mathrm{b}_2 x + \mathrm{b}_3 + \mathrm{b}_4 x + \mathrm{b}_5 + \mathrm{b}_6 x)\mathrm{d}x = -\frac{2}{3}(\mathrm{b}_2 + \mathrm{b}_4 + \mathrm{b}_6) = 0$$

(c) $E_3 : \boldsymbol{n} = -\hat{\boldsymbol{i}}$, $x = -1$.

i. $p_k = 1$:

$$\Rightarrow \int_{1}^{-1} -(\mathrm{a}_1 - \mathrm{a}_2 + \mathrm{a}_3 y - \mathrm{a}_4 y + \mathrm{a}_5 + \mathrm{a}_6 y)\mathrm{d}y = 2(\mathrm{a}_1 - \mathrm{a}_2 + \mathrm{a}_5) = 0$$

ii. $p_k = y$:

$$\Rightarrow \int_{1}^{-1} -y(\mathrm{a}_1 - \mathrm{a}_2 + \mathrm{a}_3 x - \mathrm{a}_4 x + \mathrm{a}_5 + \mathrm{a}_6 x)\mathrm{d}y = \frac{2}{3}(\mathrm{a}_3 - \mathrm{a}_4 + \mathrm{a}_6) = 0$$

(d) $E_4 : \boldsymbol{n} = -\hat{\boldsymbol{j}}$, $y = -1$.

i. $p_k = 1$:

$$\Rightarrow \int_{-1}^{1} -(\mathrm{b}_1 + \mathrm{b}_2 x - \mathrm{b}_3 - \mathrm{b}_4 x + \mathrm{b}_5 + \mathrm{b}_6 x)\mathrm{d}x = -2(\mathrm{b}_1 - \mathrm{b}_3 + \mathrm{b}_5) = 0$$

ii. $p_k = y$:

$$\Rightarrow \int_{-1}^{1} -x(\mathrm{b}_1 + \mathrm{b}_2 x - \mathrm{b}_3 - \mathrm{b}_4 x + \mathrm{b}_5 + \mathrm{b}_6 x)\mathrm{d}x = -\frac{2}{3}(\mathrm{b}_2 - \mathrm{b}_4 + \mathrm{b}_6) = 0$$

To calculate the interior degrees of freedom we will have to calculate 4 double integrals over the reference square:

(a) $\boldsymbol{p}_k = \langle 1, 0 \rangle$:

$$\int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\phi}_1 \cdot \boldsymbol{p}_k \mathrm{d}y\mathrm{d}x = 4\mathrm{a}_1 + \frac{4}{3}\mathrm{a}_5 = 0$$

(b) $\boldsymbol{p}_k = \langle y, 0\rangle$:

$$\int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\phi}_1 \cdot \boldsymbol{p}_k \mathrm{d}y\mathrm{d}x = \frac{4}{3}\mathrm{a}_3 + \frac{4}{9}\mathrm{a}_6 = 0$$

(c) $\boldsymbol{p}_k = \langle 0, 1\rangle$:

$$\int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\phi}_1 \cdot \boldsymbol{p}_k \mathrm{d}y\mathrm{d}x = 4\mathrm{b}_1 + \frac{4}{3}\mathrm{b}_5 = 0$$

(d) $\boldsymbol{p}_k = \langle 0, x\rangle$:

$$\int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\phi}_1 \cdot \boldsymbol{p}_k \mathrm{d}y\mathrm{d}x = +\frac{4}{3}\mathrm{b}_2 + \frac{4}{9}\mathrm{b}_6 = 0$$

This leads to the linear system:

$$\begin{pmatrix}
2 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{2}{3} & \frac{2}{3} & 0 & \frac{2}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & 0 & -2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & 0 & -\frac{2}{3} & 0 & -\frac{2}{3} \\
2 & -2 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{2}{3} & -\frac{2}{3} & 0 & \frac{2}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 2 & 0 & -2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & 0 & \frac{2}{3} & 0 & -\frac{2}{3} \\
4 & 0 & 0 & 0 & \frac{4}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{4}{3} & 0 & 0 & \frac{4}{9} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & \frac{4}{3} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{4}{3} & 0 & 0 & 0 & \frac{4}{9}
\end{pmatrix}
\begin{pmatrix}
\mathrm{a}_1 \\ \mathrm{a}_2 \\ \mathrm{a}_3 \\ \mathrm{a}_4 \\ \mathrm{a}_5 \\ \mathrm{a}_6 \\ \mathrm{b}_1 \\ \mathrm{b}_2 \\ \mathrm{b}_3 \\ \mathrm{b}_4 \\ \mathrm{b}_5 \\ \mathrm{b}_6
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0
\end{pmatrix},$$

which can be solved to give:

$$\boldsymbol{\phi}_1 = \frac{1}{4}\begin{pmatrix} 1 + 4x - 3x^2 \\ 0 \end{pmatrix}.$$

By modifying the righthand side we can solve for the other 11 basis functions:

$$\boldsymbol{\phi}_2 = -\frac{1}{8}\begin{pmatrix} 3y - 6xy - 9x^2y \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}_3 = \frac{1}{8}\begin{pmatrix} 0 \\ 1 - 2y - 3y^2 \end{pmatrix} \quad \boldsymbol{\phi}_4 = \frac{1}{8}\begin{pmatrix} 0 \\ 3x - 6xy - 9xy^2 \end{pmatrix}$$

$$\boldsymbol{\phi}_5 = \frac{1}{4}\begin{pmatrix} 1 - 2x - 3x^2 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}_6 = -\frac{1}{8}\begin{pmatrix} 3y + 6xy - 9x^2y \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}_7 = \frac{1}{8}\begin{pmatrix} 0 \\ 1 + 2y - 3y^2 \end{pmatrix}$$

$$\boldsymbol{\phi}_8 = \frac{1}{4}\begin{pmatrix} 0 \\ 3x + 6xy - 9xy^2 \end{pmatrix} \quad \boldsymbol{\phi}_9 = -\frac{1}{4}\begin{pmatrix} 3x + 3x^2 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}_{10} = \frac{1}{8}\begin{pmatrix} 9y - 9y^2 \\ 0 \end{pmatrix}$$

$$\boldsymbol{\phi}_{11} = \frac{1}{8}\begin{pmatrix} 0 \\ 3 - 3y^2 \end{pmatrix} \quad \boldsymbol{\phi}_{12} = \frac{1}{8}\begin{pmatrix} 0 \\ 9x - 9xy^2 \end{pmatrix}$$

REFERENCES

[1] Douglas N. Arnold, Daniele Boffi, and Richard S. Faulk. Quadrilateral H(div) Finite Elements. *SIAM J. Numer. Anal.*, 42(6):2429–2451, 2005.

[2] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – A General Purpose Object Oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[3] Lawrence C. Evans. *Partial Differential Equations Second Edition*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, Rhode Island, 2010.

[4] U. Ghia, K.N Ghia, and C.T. Shin. High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method. *J. Comput. Phys.*, 48(3):347–411, 1982.

[5] Max D. Gunzburger. *Finite Element Methods for Viscous Incompressible Flows*. Academic Press Inc., San Diego, CA, 1989.

[6] D. W. Kelly, J. P. De S. R. Gag, O. C. Zienkiewicz, and I. Babuška. A posteriori error analysis and adaptive processes in the finite element method: Part I–Error Analysis. *Int. J. Num. Meth. Engrg.*, 19:1593, 1983.

[7] Stefano Mantica, Fausto Saleri, and Luca Bergamaschi. Mixed Finite Element Approximation of Darcy's Law in Porous Media, 1995.

[8] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. *Flow Simulation with High-Performance Computers II: DFG Priority Research Programme Results 1993–1995*, chapter Benchmark Computations of Laminar Flow Around a Cylinder, pages 547–566. Vieweg+Teubner Verlag, Wiesbaden, 1996.

[9] P N Shankar. *Slow Viscous Flows*. Imperial College Press, London, 2007.

[10] Stephen Whitaker. Flow in Porous Media I: A Theoretical Derivation of Darcy's Law. *Transport in Porous Media*, 1(1):3–25, 1986.

[11] Stephen Whitaker. *The Method of Volume Averaging*. Kluwer Academic Publishers, Dordrecht, the Netherlands, 1999.