$H(\mathrm{div})$-conforming Discontinuous Galerkin Methods for Multiphase Flow

by

Kyle Booker

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2021

Abstract

Computational fluid dynamics (CFD) is concerned with numerically solving and visualizing complex problems involving fluids with numerous engineering applications. Mathematical models are derived from basic governing equations using assumptions of the initial conditions and physical properties. CFD is less costly than experimental procedures while still providing an accurate depiction of the phenomenon. Models permit to test different parameters and sensitivity quickly, which is highly adaptable to solving similar conditions; however, these problems are often computationally costly, which necessitates sophisticated numerical methods.

Modeling multiphase flow problems involving two or more fluids of different states, phases, or physical properties. Boilers are an example of bubbly flows where accurate models are relevant for operation safety or contain turbulence, resulting in reduced efficiency. Bubbly flows are an example of continuous-dispersed phase flow, modeled using the Eulerian multiphase flow model. The dispersed phase is considered an interpenetrating continuum with the continuous phase.

In the two-fluid model, a phase fraction parameter varying from zero to one is used to describe the fraction of fluid occupying each point in space. This model is ill-posed, non-linear, non-conservative, and non-hyperbolic, which affects the stability and accuracy of the solution. There have been methods allowing the model to be well-posed to obtain stability and uniqueness, but this raises questions regarding the physicality of the solution. Approaches to increasing the well-posedness of the model include additional momentum transfer terms, virtual mass contributions, dispersion terms, or inclusion of momentum flux. There is division among which methods are valid for an accurate description of the phenomena, and more research is required to examine these effects.

While finite difference schemes are often simple to implement, they do not scale well to problems with complicated geometries or difficult boundary conditions. Numerical methods may also add ad-hoc terms that compromise the physicality of the solution. The choice of numerical method results from a time versus accuracy trade-off. In industry, efficient performing schemes have become standard; however, this might sacrifice the physical properties of the natural phenomenon.

$H(\mathrm{div})$-conforming finite element spaces contain vector functions where both the function and its divergence are continuous on each element. Examples of $H(\mathrm{div})$-conforming spaces include Raviart–Thomas and Brezzi–Marini–Douglas spaces. These spaces allow for the velocity vector function to be pointwise divergence-free with machine precision and being pressure-robust.

This thesis presents a discontinuous Galerkin $H(\mathrm{div})$-conforming method for the two-fluid model. Instead of solving the dispersed and continuous phase velocities, the dispersed and mixture velocities are solved, allowing us to easily apply our pressure-robust scheme to the divergence constraint of the two-fluid model. The viscous term numerical flux is derived from a standard interior penalty discontinuous Galerkin method flux, and the convective flux is calculated using the local Lax–Friedrichs flux.

Simulations of two-dimensional channel flow are performed using the $H(\mathrm{div})$-conforming method. While we can qualitatively assess the approximate velocity, pressure, and phase fraction solutions, there still needs to be work done to use this method for actual applications. The mixture velocity is calculated to be divergence-free within machine-precision. Limitations of the numerical scheme are discussed, and possible areas for further research.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family, for none of this would be possible without their encouragement and support.

I would like to thank my supervisors, Professor Sander Rhebergen and Professor Nasser Mohieddin Abukhdeir, for their guidance, support and insightful comments.

Lastly, I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Waterloo for their financial support and Compute Canada for the use of their computational facilities.

Dedication

To my parents, for their endless love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BDM Brezzi–Douglas–Marini. x, 15, 16, 20, 22, 23

CG Continuous Galerkin. xi, 1, 2, 5

DG Discontinuous Galerkin. xi, 1, 2, 3, 5, 6, 16, 17, 20, 46

DOFs Degrees-of-freedom. xi, 1, 2

LLF Local Lax–Friedrichs. 8, 10, 44, 45, 80, 81

PDE Partial differential equation. 2, 5, 7, 80

RT Raviart–Thomas. 15

List of Symbols

α	Time-averaged local phase fraction
μ	Dynamic viscosity
ν	Kinematic viscosity
ρ	Density
P	Pressure
$\mathbf{g}$	Gravitational acceleration
$\mathbf{M}$	Momentum source term
Ω	Simulation domain
$\partial\Omega$	Simulation domain boundary
τ	Stress tensor
$\bar{\cdot}$	Time average
$\bar{\bar{\cdot}}$	Phasic average
$\widehat{\cdot}$	Mass-weighted mean phasic average
C_D	Drag coefficient
h	Mesh element length
t	Time
k	Polynomial approximation degree
N	Number of elements
Δt	Time step
$\epsilon(\cdot)$	Symmetric gradient
$[\![\cdot]\!]$	Jump operator
$\{\!\{\cdot\}\!\}$	Average operator

Chapter 1

Introduction

1.1 Research Motivation

A multiphase fluid is two or more distinct components or *phases* composed of either fluids or solids but exhibits properties of a fluid. Proper averaging techniques [39] can be used to derive a set of equations describing multiphase flow, which in principle correctly describes the dynamics of any multiphase system, subject to very general assumptions. Constitutive equations are needed to close this set of equations. Many multiphase flow forms exist, such as flow in a fluidized bed [25], bubbly flow in boilers [30, 40, 53, 58], and gas-particle flow in combustion reactors. Constitutive laws describing interactions and material properties of the various phases involved differ for particular cases.

Poor mass conservation in numerical methods is the culprit for non-physical behavior shown in mixed methods for incompressible flows. Lack of pressure robustness is due to the relaxation of the divergence constraint for incompressible flows, found in classical mixed methods to construct discretely inf-sup stable discretization schemes [3]. Pressure robust schemes have the advantage of being numerically stable regarding large pressure gradients.

Discontinuous Galerkin (DG) methods are one way to implement pressure-robust numerical schemes. The differences between the degrees-of-freedom (DOFs) in the continuous Galerkin (CG) and DG methods are illustrated in Figure 1.1. In the CG finite element method, the facet DOFs are shared between elements, as shown. For most DG methods, the DOFs do not live on the facets and instead live on the interior of each element. There are no DOFs shared between elements. While there are differences regarding the type of DOFs in both methods, the total number of DOFs is higher in the DG method, which results in a linear system that is computationally more expensive to solve. One of the primary benefits of DG methods is their local conservation properties, as shown in [12, 13, 23].

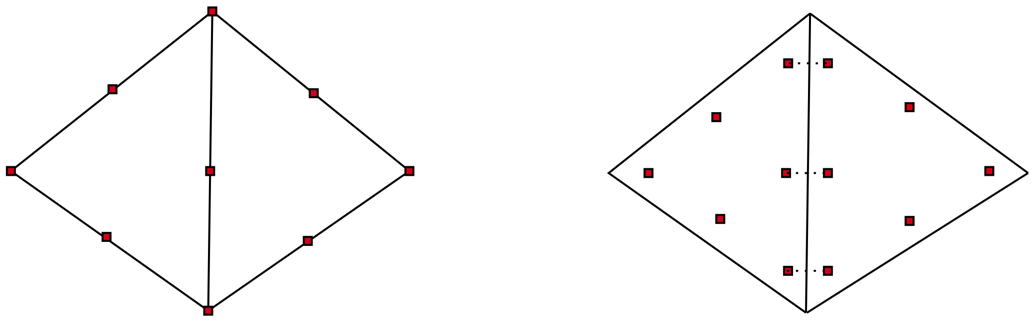

Figure 1.1: CG and DG DOFs for polynomials of total degree 2 on triangular elements. The red squares denote the element DOFs, and the dash lines indicate the communication between elements for the DG elements.

By allowing the polynomial approximation to be discontinuous across facets such as in Figure 1.2, the approximation at element boundaries is undefined. A flux across element boundaries is defined to solve this problem. Each element is only able to communicate through neighboring elements via the boundary data provided. The numerical flux is necessary to propagating information across the domain. By carefully choosing our numerical flux, we can encode invariants of the system we are modeling, such as the wave propagation speed or other information depending on the partial differential equation (PDE). An incorrect choice of numerical flux can lead to spurious oscillations or numerical instability.

One of the simplest numerical fluxes averages the values at the facets of two neighboring elements, known as central flux [42]. While central flux is undoubtedly easy to implement, it is rarely used because it can create instabilities for nonlinear problems. A more realistic numerical flux exploits the wave-like behavior of the solution. Upwinding flux uses information biased in the direction of the sign of the characteristic speeds, propagating information in the direction of motion [10].

Multiphase flow also presents a set of conservation equations which is the main motivation for using a locally conservative DG method. DG methods can support high order local approximations [22, 28, 49, 52]. Since each element can be viewed as a separate entity while only requiring boundary data from its neighbors, the degree of approximation can vary over each element and in time over the simulation while still providing mathematical rigor [2].

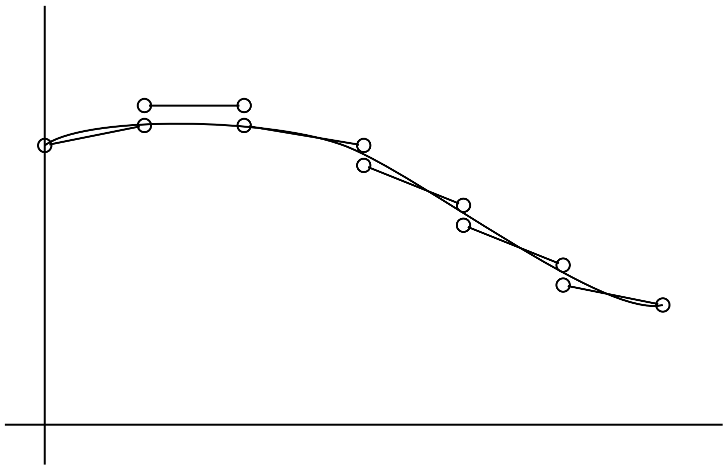

Figure 1.2: Discontinuous approximation of a continuous function. Here the piecewise approximation is undefined at the element boundaries.

Vortex-dominated flows exhibit large pressure gradients [78, 48], which require pressure-robustness to enforce the divergence constraint. They also outperform non-pressure-robust schemes such as classical Taylor–Hood element schemes at high Reynolds numbers. While there do exist continuous finite element pressure-robust schemes, $H(\mathrm{div})$-conforming DG methods allow us to incorporate a flux that can correspond to the wave speed of the solution and be particularly robust to shocks in the solution [17]. Pressure-robust schemes offer higher convergence rates while preserving the velocity of the solution under changes in viscosity. $H(\mathrm{div})$-conforming DG methods allow for numerical precision, mainly when the pressure errors are significant, as shown in Section 3.

1.2 Objectives

The overall objective of this research is to demonstrate the advantages of H-div conforming discontinuous Galerkin methods applied to two-phase flow problems. The following objectives are included to support this:

- To investigate the well-posedness of the two-fluid model for two-phase flow problems.
- To investigate the advantages of $H(\mathrm{div})$-conforming discontinuous Galerkin finite element methods such as pressure-robustness and stability with higher-order approximation polynomials.

- Development of an $H(\text{div})$-conforming discontinuous Galerkin method to solve the two-fluid model.

1.3 Structure of Thesis

The thesis is organized into six chapters: Chapter 2 — the discontinuous Galerkin Method, Chapter 3 — $H(\text{div})$-conforming finite element method for incompressible Navier-Stokes flow, Chapter 4 — multiphase flow, Chapter 5 — $H(\text{div})$-conforming finite element method for two-fluid incompressible flow systems and Chapter 6 — conclusions and reccommendations for future work.

Chapter 2 describes the discontinuous Galerkin method required for the thesis. Basic notation for the thesis is introduced, and a fully discretized discontinuous Galerkin weak formulation is derived.

Chapter 3 elaborates on the ideas from Chapter 2, presenting the idea of an $H(\text{div})$-conforming discontinuous Galerkin method and the concept of pressure-robustness. The method mentioned above is applied to the incompressible Navier-Stokes problem with results from numerical simulations.

Chapter 4 provides an overview of the Euler-Euler two-fluid model and addresses the numerical complexities of the model and the well-posedness.

Chapter 5 derives an $H(\text{div})$-conforming weak formulation of the two-fluid model. Simulation results are presented with a discussion of improvements for the numerical method.

Chapter 6 summarizes the conclusions from this work and provides recommendations for future work.

Chapter 2

The Discontinuous Galerkin Method

This chapter will introduce the DG finite element method for a vector advection-diffusion problem. We start by introducing relevant notation, after which we derive the DG discretization.

DG methods were first introduced in [60] and later developed in [14] as a robust finite element method allowing for a practical framework for the development of high-order accurate methods using unstructured grids. They are based on nonconforming finite element spaces consisting of piecewise polynomials that are discontinuous across elements. The governing equations are reformulated into their weak formulations, where the condition that the solution must satisfy the differential equation at every point in the domain is relaxed. Instead, the PDE will be weakly satisfied. The main reasons for us to consider the DG method is that it is well suited for large-scale time-dependent computations in which high accuracy is required, it is well suited for parallel computing [4, 9, 68] and, unlike, for example, the CG method, the DG method is locally conservative [12, 13, 23]. Because of this, DG is often used to solve hyperbolic conservation equations [11, 28, 29, 32, 47, 56].

2.1 Notation and Operators

Some notation is required before presenting the DG method. Consider a domain $\Omega \subset \mathbb{R}^d$, with $d = 2, 3$ and let $\mathcal{T}$ be a partition of the domain Ω into non-overlapping elements. Interior facets of $\mathcal{T}$ are denoted by $\mathcal{F}_I$, whereas boundary facets of $\mathcal{T}$ are denoted $\mathcal{F}_B$. Let $h_K = diam(K) \in \mathcal{T}$ be the diameter of each element. On the boundary of a cell, ∂K, the outward unit normal vector is denoted by $\mathbf{n}$. In this thesis, all vectors $\mathbf{r}$ will be bolded, and all tensors $\underline{\mathbf{s}}$ will be underlined.

On the interior facets, two important operators need to be defined for the DG method. The average $\{\!\{\cdot\}\!\}$ and jump $[\![\cdot]\!]$ operators are defined as follows: for a scalar q,

$$\{\!\{q\}\!\} := \frac{1}{2}\left(q^+ + q^-\right), \quad [\![q\mathbf{n}]\!] := q^+\mathbf{n}^+ + q^-\mathbf{n}^-, \tag{2.1}$$

where $\mathbf{n}^+ = -\mathbf{n}^-$. On boundary facets we set

$$\{\!\{q\}\!\} := q, \quad [\![q\mathbf{n}]\!] := q\mathbf{n}. \tag{2.2}$$

The average and jump operators for a vector $\mathbf{r}$ on interior facets are defined as

$$\{\!\{\mathbf{r}\}\!\} := \frac{1}{2}\left(\mathbf{r}^+ + \mathbf{r}^-\right), \quad [\![\mathbf{r}\cdot\mathbf{n}]\!] := \mathbf{r}^+\cdot\mathbf{n}^+ + \mathbf{r}^-\cdot\mathbf{n}^-, \tag{2.3}$$

$$[\![\mathbf{r}\otimes\mathbf{n}]\!] := \mathbf{r}^+\otimes\mathbf{n}^+ + \mathbf{r}^-\otimes\mathbf{n}^-, \tag{2.4}$$

while on boundary facets they are defined as

$$\{\!\{\mathbf{r}\}\!\} := \mathbf{r}, \quad [\![\mathbf{r}\cdot\mathbf{n}]\!] := \mathbf{r}\cdot\mathbf{n}, \quad [\![\mathbf{r}\otimes\mathbf{n}]\!] := \mathbf{r}\otimes\mathbf{n}, \tag{2.5}$$

where $\mathbf{r}\otimes\mathbf{n}$ denotes the outer product of two vectors. Finally the average and jump operators for a tensor $\underline{\mathbf{s}}$ on interior facets are defined as

$$\{\!\{\underline{\mathbf{s}}\}\!\} := \frac{1}{2}\left(\underline{\mathbf{s}}^+ + \underline{\mathbf{s}}^-\right), \quad [\![\underline{\mathbf{s}}\cdot\mathbf{n}]\!] := \underline{\mathbf{s}}^+\cdot\mathbf{n}^+ + \underline{\mathbf{s}}^-\cdot\mathbf{n}^-, \tag{2.6}$$

while on boundary facets they are defined as

$$\{\!\{\underline{\mathbf{s}}\}\!\} := \underline{\mathbf{s}}, \quad [\![\underline{\mathbf{s}}\cdot\mathbf{n}]\!] := \underline{\mathbf{s}}\cdot\mathbf{n}, \tag{2.7}$$

where $\underline{\mathbf{s}}\cdot\mathbf{n}$ denotes the non-symmetric dot product of a tensor with a vector.

To define the DG method we define also the following finite element spaces:

$$\mathbf{V}_h^k = \left\{\mathbf{v}_h \in [L^2(\Omega)]^d : \forall K \in \mathcal{T}(\Omega), \mathbf{v}_h\,|_K \in [\mathbb{P}^k(K)]^d\right\}, \tag{2.8a}$$
$$\underline{\mathbf{W}_h^k} = \left\{\underline{\boldsymbol{\tau}_h} \in [L^2(\Omega)]^{d\times d} : \forall K \in \mathcal{T}(\Omega), \underline{\boldsymbol{\tau}_h}\,|_K \in [\mathbb{P}^k(K)]^{d\times d}\right\}, \tag{2.8b}$$

where $\mathbb{P}^k(K)$ is the set of polynomials of total degree less than or equal to k. Note that these are spaces of piecewise polynomials that are discontinuous across element boundaries.

2.2 Discontinous Galerkin Method for the advection-diffusion problem

We will now introduce the discontinuous Galerkin method for an advection-diffusion problem. This problem is as follows: Consider the domain $\Omega \subset \mathbb{R}^d, d = 2, 3$, and let the time interval of interest be given by $I = (0, t_n]$. Let $\epsilon \in \mathbb{R}^+$ be a scalar diffusion coefficient. The advection-diffusion problem for the vector velocity field $\mathbf{u} : \Omega \times I \to \mathbb{R}^d$ is given by

$$\partial_t \mathbf{u} + \nabla \cdot \underline{\mathbf{f}(\mathbf{u})} - \epsilon \nabla^2 \mathbf{u} = 0 \quad \text{in } \Omega \times I, \tag{2.9a}$$

$$\mathbf{u} = \mathbf{g} \quad \text{on } \Gamma_D \times I, \tag{2.9b}$$

$$(\underline{\mathbf{f}(\mathbf{u})} - \epsilon \nabla \mathbf{u}) \cdot \mathbf{n} = \mathbf{h}^{in} \quad \text{on } \Gamma_N^{in} \times I. \tag{2.9c}$$

$$-(\epsilon \nabla \mathbf{u}) \cdot \mathbf{n} = \mathbf{h}^{out} \quad \text{on } \Gamma_N^{out} \times I, \tag{2.9d}$$

where the boundary of Ω has been partitioned into a Dirichlet (Γ_D), Neumann inflow (Γ_N^{in}), and Neumann outflow (Γ_N^{out}) boundary: $\partial\Omega = \Gamma_D \cup \Gamma_N^{in} \cup \Gamma_N^{out}$ and $\Gamma_D \cap \Gamma_N^{in} \cap \Gamma_N^{out} = \emptyset$. Throughout we assume $\Gamma_D \neq \emptyset$. On Γ_D, $\mathbf{g} : \Gamma_D \times I \to \mathbb{R}^d$ is the given Dirichlet boundary data, on Γ_N^{in}, $\mathbf{h}^{in} : \Gamma_N^{in} \times I \to \mathbb{R}^d$ is the given Neumann inflow boundary data, on Γ_N^{out}, $\mathbf{h}^{out} : \Gamma_N^{out} \times I \to \mathbb{R}^d$ is the given Neumann outflow boundary data and where $\underline{\mathbf{f}(\mathbf{u})}$, a tensor, is the flux function dependent on the velocity $\mathbf{u}$. Here we have both the advection term represented by $\nabla \cdot \underline{\mathbf{f}(\mathbf{u})}$ and the diffusion term $\epsilon \nabla^2 \mathbf{u}$, where the operator $\nabla^2(\cdot) = \nabla \cdot (\nabla(\cdot)) = \Delta(\cdot)$ is the Laplacian operator.

2.2.1 Weak Formulation

We will now describe the weak formulation for the advection-diffusion problem. Let $\underline{\boldsymbol{\sigma}} : \Omega \to \mathbb{R}^{d \times d}$ be an auxiliary variable, then we may write equation (2.9a) as

$$\underline{\boldsymbol{\sigma}} = \nabla \mathbf{u} \quad \text{in } \Omega \times I, \tag{2.10a}$$

$$\partial_t \mathbf{u} + \nabla \cdot \underline{\mathbf{f}(\mathbf{u})} - \epsilon \nabla \cdot \underline{\boldsymbol{\sigma}} = 0 \quad \text{in } \Omega \times I, \tag{2.10b}$$

which has reduced the PDE to a first order system. Mixed methods can be found in [38, 55, 64]. We will denote approximation variables with a subscript h. We will use the finite element spaces introduced in section 2.1. The auxiliary variable $\underline{\boldsymbol{\sigma}}$ is introduced here purely to derive a discretization for (2.9). In what follows below, we will eliminate $\underline{\boldsymbol{\sigma}}$ to find a discretization for (2.9) for the unknown $\mathbf{u}$ only. Multiplying (2.10a) by a test function $\underline{\boldsymbol{\tau}_h} \in \underline{\mathbf{W}_h^k}$ and (2.10b) by a test function $\mathbf{v}_h \in \mathbf{V}_h^k$, as well as integrating and summing over all elements of the triangulation yields the equations for the approximate solution $(\underline{\boldsymbol{\sigma}_h}, \mathbf{u}_h) \in \underline{\mathbf{W}_h^k} \times \mathbf{V}_h^k$:

$$\sum_{K\in\mathcal{T}} \int_K \underline{\boldsymbol{\sigma}_h} : \underline{\boldsymbol{\tau}_h} \mathrm{d}x = \sum_{K\in\mathcal{T}} \int_K \nabla \mathbf{u}_h : \underline{\boldsymbol{\tau}_h} \mathrm{d}x, \tag{2.11a}$$

$$\sum_{K\in\mathcal{T}} \int_K \mathbf{v}_h \cdot \partial_t \mathbf{u}_h \mathrm{d}x + \sum_{K\in\mathcal{T}} \int_K \mathbf{v}_h \cdot (\nabla \cdot \underline{\mathbf{f}(\mathbf{u}_h)}) \mathrm{d}x - \sum_{K\in\mathcal{T}} \int_K \epsilon \mathbf{v}_h \cdot (\nabla \cdot \underline{\boldsymbol{\sigma}_h}) \mathrm{d}x = 0. \tag{2.11b}$$

Integration by parts can be applied to both equations in (2.11) yielding

$$\sum_{K\in\mathcal{T}} \int_K \underline{\boldsymbol{\sigma}_h} : \underline{\boldsymbol{\tau}_h} \mathrm{d}x = -\sum_{K\in\mathcal{T}} \int_K \mathbf{u}_h \cdot (\nabla \cdot \underline{\boldsymbol{\tau}_h}) \mathrm{d}x + \sum_{K\in\mathcal{T}} \int_{\partial K} \overline{\mathbf{u}_h} \otimes \mathbf{n} : \underline{\boldsymbol{\tau}_h} \mathrm{d}s, \tag{2.12a}$$

$$\begin{aligned} &\sum_{K\in\mathcal{T}} \int_K \left(\mathbf{v}_h \cdot \partial_t \mathbf{u}_h - \underline{\mathbf{f}(\mathbf{u}_h)} : \nabla \mathbf{v}_h + \epsilon \underline{\boldsymbol{\sigma}_h} : \nabla \mathbf{v}_h \right) \mathrm{d}x + \sum_{K\in\mathcal{T}} \int_{\partial K} \underline{\mathbf{H}} \cdot \mathbf{v}_h \mathrm{d}s \\ &- \sum_{K\in\mathcal{T}} \int_{\partial K} \epsilon \overline{\underline{\boldsymbol{\sigma}_h}} : (\mathbf{v}_h \otimes \mathbf{n}) \mathrm{d}s = 0, \end{aligned} \tag{2.12b}$$

where $\overline{\mathbf{u}_h}$, $\underline{\mathbf{H}}$ and $\overline{\underline{\boldsymbol{\sigma}_h}}$ are approximations to, respectively, $\mathbf{u}_h$, $\underline{\mathbf{f}(\mathbf{u}_h)} \cdot \mathbf{n}$, and $\underline{\boldsymbol{\sigma}_h}$ on element boundaries. These approximations are necessary because $\mathbf{u}_h$, $\underline{\mathbf{f}(\mathbf{u}_h)} \cdot \mathbf{n}$, and $\underline{\boldsymbol{\sigma}_h}$ are not uniquely defined on element boundaries.

For $\underline{\mathbf{H}}$, we use the well-known local Lax–Friedrichs flux (LLF). Consider two adjacent elements K^+ and K^- and let F be a facet shared by both elements. Let $\mathbf{u}_h^{K+}$ be the restriction of $\mathbf{u}_h$ to K^+ and let $\mathbf{u}_h^{K-}$ be the restriction of $\mathbf{u}_h$ to K^-. Further, let $\mathbf{u}_h^+$ be the restriction of $\mathbf{u}_h^{K+}$ to the facet F and $\mathbf{u}_h^-$ be the restriction of $\mathbf{u}_h^{K-}$ to the facet F. The LLF numerical flux [57, 62] is then defined as:

$$\underline{\mathbf{H}\left(\mathbf{u}_h^+, \mathbf{u}_h^-, \mathbf{n}^+\right)} = \frac{1}{2}\left(\underline{\mathbf{f}\left(\mathbf{u}_h^+\right)} + \underline{\mathbf{f}\left(\mathbf{u}_h^-\right)} \right) \cdot \mathbf{n}^+ + \frac{1}{2}\lambda \left(\mathbf{u}_h^+ - \mathbf{u}_h^-\right) \tag{2.13}$$

where $\mathbf{n}^+$ is the unit normal vector on F pointing outwards from K^+, and where λ is the largest eigenvalue (in absolute value) of the Jacobian of $\underline{\mathbf{f}\left(\mathbf{u}_h\right)} \cdot \mathbf{n}$. We remark that the LLF introduces upwinding in our discretization and is therefore well-suited for advection dominated problems. Indeed, $\frac{1}{2}\lambda(\mathbf{u}_h^+ - \mathbf{u}_h^-)$ is a diffusive term that stabilizes the numerical method. Furthermore, note that LLF is consistent, i.e., $\underline{\mathbf{H}(\mathbf{u}, \mathbf{u}, \mathbf{n})} = \underline{\mathbf{f}(\mathbf{u})} \cdot \mathbf{n}$ for any smooth function $\mathbf{u}$.

We will now define $\overline{\mathbf{u}_h}$ and $\overline{\underline{\boldsymbol{\sigma}_h}}$ and explain how we eliminate $\underline{\boldsymbol{\sigma}_h}$. We first remark that $\overline{\mathbf{u}_h}$ and $\overline{\underline{\boldsymbol{\sigma}_h}}$ will be defined to be single-valued on element boundaries. We can then write the element boundary integrals in (2.12) in terms of facet integrals:

$$\sum_{K\in\mathcal{T}}\int_{\partial K}\overline{\mathbf{u}_h}\otimes\mathbf{n}:\underline{\boldsymbol{\tau}_h}\mathrm{d}s=\sum_{F\in\mathcal{F}_I}\int_F\overline{\mathbf{u}_h}\cdot[\![\underline{\boldsymbol{\tau}_h}\cdot\mathbf{n}]\!]\mathrm{d}s+\sum_{F\in\mathcal{F}_B}\int_F\overline{\mathbf{u}_h}\otimes\mathbf{n}:\underline{\boldsymbol{\tau}_h}\mathrm{d}s, \tag{2.14a}$$

$$\sum_{K\in\mathcal{T}}\int_{\partial K}\underline{\mathbf{H}}\cdot\mathbf{v}_h\mathrm{d}s=\sum_{F\in\mathcal{F}_I}\int_F\underline{\mathbf{H}}\cdot(\mathbf{v}_h^+-\mathbf{v}_h^-)\mathrm{d}s+\sum_{F\in\mathcal{F}_B}\int_F\underline{\mathbf{H}}\cdot\mathbf{v}_h\mathrm{d}s, \tag{2.14b}$$

$$\sum_{K\in\mathcal{T}}\int_{\partial K}\mathbf{v}_h\otimes\mathbf{n}:\epsilon\overline{\underline{\boldsymbol{\sigma}_h}}\mathrm{d}s=\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{v}_h\otimes\mathbf{n}]\!]:\epsilon\overline{\underline{\boldsymbol{\sigma}_h}}\mathrm{d}s+\sum_{F\in\mathcal{F}_B}\int_F\mathbf{v}_h\otimes\mathbf{n}:\epsilon\overline{\underline{\boldsymbol{\sigma}_h}}\mathrm{d}s. \tag{2.14c}$$

Consider now the first integral on the right-hand side of (2.12a). Integration by parts and writing element boundary integrals in terms of facet integrals results in

$$\begin{aligned}-\sum_{K\in\mathcal{T}}\int_K\mathbf{u}_h\cdot(\nabla\cdot\underline{\boldsymbol{\tau}_h})\mathrm{d}x&=\sum_{K\in\mathcal{T}}\int_K\nabla\mathbf{u}_h:\underline{\boldsymbol{\tau}_h}\mathrm{d}x-\sum_{F\in\mathcal{F}_I}\int_F\{\!\{\mathbf{u}_h\}\!\}\cdot[\![\underline{\boldsymbol{\tau}_h}\cdot\mathbf{n}]\!]\mathrm{d}s\\&\quad-\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{u}_h\otimes\mathbf{n}]\!]:\{\!\{\underline{\boldsymbol{\tau}_h}\}\!\}\mathrm{d}s-\sum_{F\in\mathcal{F}_B}\int_F\mathbf{u}_h\otimes\mathbf{n}:\underline{\boldsymbol{\tau}_h}\mathrm{d}s.\end{aligned} \tag{2.15}$$

Choosing our test function $\underline{\boldsymbol{\tau}_h}=\nabla\mathbf{v}_h\in\underline{\mathbf{W}}_h^k$ and combining equations (2.12a), (2.14a) and (2.15) we have

$$\begin{aligned}\sum_{K\in\mathcal{T}}\int_K\underline{\boldsymbol{\sigma}_h}:\nabla\mathbf{v}_h\mathrm{d}x&=\sum_{K\in\mathcal{T}}\int_K\nabla\mathbf{u}_h:\nabla\mathbf{v}_h\mathrm{d}x-\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{u}_h\otimes\mathbf{n}]\!]:\{\!\{\nabla\mathbf{v}_h\}\!\}\mathrm{d}s,\\&\quad+\sum_{F\in\mathcal{F}_I}\int_F(\overline{\mathbf{u}_h}-\{\!\{\mathbf{u}_h\}\!\})\cdot[\![\nabla\mathbf{v}_h\cdot\mathbf{n}]\!]\mathrm{d}s+\sum_{F\in\mathcal{F}_B}\int_F(\overline{\mathbf{u}_h}-\mathbf{u}_h)\otimes\mathbf{n}:\nabla\mathbf{v}_h\mathrm{d}s.\end{aligned} \tag{2.16}$$

We will now define the numerical flux $\overline{\mathbf{u}_h}$ as follows:

$$\overline{\mathbf{u}_h}=\begin{cases}\{\!\{\mathbf{u}_h\}\!\} & \text{on } F\in\mathcal{F}_I,\\ \mathbf{g} & \text{on } F\in\mathcal{F}_D,\\ \mathbf{u}_h & \text{on } F\in\mathcal{F}_N,\end{cases} \tag{2.17}$$

where $\mathcal{F}_I$ are the interior boundaries and $\mathcal{F}_B=\mathcal{F}_D\cup\mathcal{F}_N$ is the union of the Dirichlet and Neumann boundaries. Introducing (2.17) into (2.16) yields,

$$\begin{aligned}\sum_{K\in\mathcal{T}}\int_K\underline{\boldsymbol{\sigma}_h}:\nabla\mathbf{v}_h\mathrm{d}x=&\sum_{K\in\mathcal{T}}\int_K\nabla\mathbf{u}_h:\nabla\mathbf{v}_h\mathrm{d}x-\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{u}_h\otimes\mathbf{n}]\!]:\{\!\{\nabla\mathbf{v}_h\}\!\}\quad\mathrm{d}s\\&-\sum_{F\in\mathcal{F}_D}\int_F(\mathbf{u}_h-\mathbf{g})\otimes\mathbf{n}:\nabla\mathbf{v}_h\mathrm{d}s.\end{aligned} \tag{2.18}$$

Combining (2.18) now with (2.12b), (2.14b) and (2.14c) we find

$$
\begin{aligned}
0 = &\sum_{K\in\mathcal{T}}\int_K \Big(\mathbf{v}_h\cdot\partial_t\mathbf{u}_h - \underline{\mathbf{f}(\mathbf{u}_h)} : \nabla\mathbf{v}_h + \epsilon\nabla\mathbf{u}_h : \nabla\mathbf{v}_h\Big)\,\mathrm{d}x \\
&+ \sum_{F\in\mathcal{F}_I}\int_F \underline{\mathbf{H}}\cdot(\mathbf{v}_h^+ - \mathbf{v}_h^-)\mathrm{d}s + \sum_{F\in\mathcal{F}_B}\int_F \underline{\mathbf{H}}\cdot\mathbf{v}_h\mathrm{d}s \\
&- \sum_{F\in\mathcal{F}_I}\int_F [\![\mathbf{u}_h\otimes\mathbf{n}]\!] : \{\!\{\nabla\mathbf{v}_h\}\!\}\mathrm{d}s - \sum_{F\in\mathcal{F}_D}\int_F (\mathbf{u}_h-\mathbf{g})\otimes\mathbf{n} : \nabla\mathbf{v}_h\mathrm{d}s \\
&- \sum_{F\in\mathcal{F}_I}\int_F [\![\mathbf{v}_h\otimes\mathbf{n}]\!] : \epsilon\underline{\overline{\boldsymbol{\sigma}_h}}\mathrm{d}s - \sum_{F\in\mathcal{F}_B}\int_F \mathbf{v}_h\otimes\mathbf{n} : \epsilon\underline{\overline{\boldsymbol{\sigma}_h}}\mathrm{d}s.
\end{aligned}
\tag{2.19}
$$

For $\underline{\overline{\boldsymbol{\sigma}_h}}$ we now choose the standard interior penalty (IP) flux [1, 26, 35]:

$$
\underline{\overline{\boldsymbol{\sigma}_h}} = \begin{cases} \{\!\{\nabla\mathbf{u}_h\}\!\} - \frac{\beta}{h}[\![\mathbf{u}_h\otimes\mathbf{n}]\!] & \text{on } F\in\mathcal{F}_I, \\ \nabla\mathbf{u}_h - \frac{\beta}{h}[\![(\mathbf{u}_h-\mathbf{g})\otimes\mathbf{n}]\!] & \text{on } F\in\mathcal{F}_D, \\ \nabla\mathbf{u}_h & \text{on } F\in\mathcal{F}_N, \end{cases}
\tag{2.20}
$$

where β is a penalty parameter to ensure stability, which in this thesis is chosen to be $\beta = 10p^2$, with p being the degree of the approximating polynomials. Now considering the diffusive term on interior facets as well as the gradient flux defined in (2.20), we have

$$
-\sum_{F\in\mathcal{F}_I}\int_F [\![\mathbf{v}_h\otimes\mathbf{n}]\!] : \epsilon\underline{\overline{\boldsymbol{\sigma}_h}}\mathrm{d}s = -\sum_{F\in\mathcal{F}_I}\int_F [\![\mathbf{v}_h\otimes\mathbf{n}]\!] : \left(\epsilon\{\!\{\nabla\mathbf{u}_h\}\!\} - \frac{\epsilon\beta}{h}\left(\mathbf{u}_h^+ - \mathbf{u}_h^-\right)\otimes\mathbf{n}\right)\mathrm{d}s,
\tag{2.21}
$$

while on Dirichlet boundary facet integrals,

$$
-\sum_{F\in\mathcal{F}_D}\int_F \mathbf{v}_h\otimes\mathbf{n} : \epsilon\underline{\overline{\boldsymbol{\sigma}_h}}\mathrm{d}s = -\sum_{F\in\mathcal{F}_D}\int_F \mathbf{v}_h\otimes\mathbf{n} : \left(\epsilon\nabla\mathbf{u}_h - \frac{\epsilon\beta}{h}\left(\mathbf{u}_h-\mathbf{g}\right)\otimes\mathbf{n}\right)\mathrm{d}s,
\tag{2.22}
$$

and on Neumann boundary facet integrals,

$$
-\sum_{F\in\mathcal{F}_N}\int_F \mathbf{v}_h\otimes\mathbf{n} : \epsilon\underline{\overline{\boldsymbol{\sigma}_h}}\mathrm{d}s = -\sum_{F\in\mathcal{F}_N}\int_F \mathbf{v}_h\otimes\mathbf{n} : \left(\epsilon\nabla\mathbf{u}_h\right)\mathrm{d}s.
\tag{2.23}
$$

Combining these expressions with (2.19) and using the definition of the LLF flux (2.13) we find:

$$
\begin{aligned}
0 = & \sum_{K\in\mathscr{T}} \int_K \Big(\mathbf{v}_h \cdot \partial_t \mathbf{u}_h - \underline{\mathbf{f}(\mathbf{u}_h)} : \nabla \mathbf{v}_h + \epsilon \nabla \mathbf{u}_h : \nabla \mathbf{v}_h \Big) \mathrm{d}x \\
& - \sum_{F\in\mathscr{F}_I} \int_F \epsilon [\![\mathbf{u}_h \otimes \mathbf{n}]\!] : \{\!\{ \nabla \mathbf{v}_h \}\!\} \mathrm{d}s - \sum_{F\in\mathscr{F}_I} \int_F \epsilon [\![\mathbf{v}_h \otimes \mathbf{n}]\!] : \{\!\{ \nabla \mathbf{u}_h \}\!\} \mathrm{d}s \\
& - \sum_{F\in\mathscr{F}_D} \int_F \epsilon \left(\mathbf{u}_h - \mathbf{g} \right) \otimes \mathbf{n} : \nabla \mathbf{v}_h \mathrm{d}s - \sum_{F\in\mathscr{F}_D} \int_F \epsilon \mathbf{v}_h \otimes \mathbf{n} : \nabla \mathbf{u}_h \mathrm{d}s \\
& + \sum_{F\in\mathscr{F}_I} \int_F \frac{\epsilon\beta}{h} [\![\mathbf{u}_h \otimes \mathbf{n}]\!] : [\![\mathbf{v}_h \otimes \mathbf{n}]\!] \mathrm{d}s + \sum_{F\in\mathscr{F}_D} \int_F \frac{\epsilon\beta}{h} \left(\mathbf{u}_h - \mathbf{g} \right) \cdot \mathbf{v}_h \mathrm{d}s \\
& + \sum_{F\in\mathscr{F}_I} \int_F \left(\mathbf{v}_h^+ - \mathbf{v}_h^- \right) \left(\frac{1}{2} [\![\underline{\mathbf{f}(\mathbf{u}_h)} \cdot \mathbf{n}]\!] + \frac{1}{2} \lambda \left(\mathbf{u}_h^+ - \mathbf{u}_h^- \right) \right) \mathrm{d}s \\
& + \sum_{F\in\mathscr{F}_D} \int_F \mathbf{v}_h \cdot \left(\frac{1}{2} \Big(\underline{\mathbf{f}(\mathbf{u}_h)} + \underline{\mathbf{f}(\mathbf{g})} \Big) \cdot \mathbf{n} + \frac{1}{2} \lambda \left(\mathbf{u}_h - \mathbf{g} \right) \right) \mathrm{d}s \\
& + \sum_{F\in\mathscr{F}_N} \int_F \mathbf{v}_h \otimes \mathbf{n} : \Big(\underline{\mathbf{f}(\mathbf{u}_h)} - \epsilon \nabla \mathbf{u}_h \Big) \mathrm{d}s.
\end{aligned}
\tag{2.24}
$$

Substituting the Neumann boundary conditions (2.9c) and (2.9d) for the last integral of equation (2.24), we obtain the following discontinuous Galerkin weak formulation for the advection-diffusion problem (2.9):

$$
\begin{aligned}
0 = & \sum_{K\in\mathcal{T}} \int_K \left(\mathbf{v}_h \cdot \partial_t \mathbf{u}_h - \underline{\mathbf{f}(\mathbf{u}_h)} : \nabla \mathbf{v}_h + \epsilon \nabla \mathbf{u}_h : \nabla \mathbf{v}_h\right) \mathrm{d}x \\
& - \sum_{F\in\mathcal{F}_I} \int_F \epsilon [\![\mathbf{u}_h \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{v}_h\}\!\} \mathrm{d}s - \sum_{F\in\mathcal{F}_I} \int_F \epsilon [\![\mathbf{v}_h \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{u}_h\}\!\} \mathrm{d}s \\
& - \sum_{F\in\mathcal{F}_D} \int_F \epsilon \left(\mathbf{u}_h - \mathbf{g}\right) \otimes \mathbf{n} : \nabla \mathbf{v}_h \mathrm{d}s - \sum_{F\in\mathcal{F}_D} \int_F \epsilon \mathbf{v}_h \otimes \mathbf{n} : \nabla \mathbf{u}_h \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_I} \int_F \frac{\epsilon\beta}{h} [\![\mathbf{u}_h \otimes \mathbf{n}]\!] : [\![\mathbf{v}_h \otimes \mathbf{n}]\!] \mathrm{d}s + \sum_{F\in\mathcal{F}_D} \int_F \frac{\epsilon\beta}{h} \left(\mathbf{u}_h - \mathbf{g}\right) \cdot \mathbf{v}_h \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_I} \int_F \left(\mathbf{v}_h^+ - \mathbf{v}_h^-\right) \left(\frac{1}{2} \{\!\{\underline{\mathbf{f}(\mathbf{u}_h)}\}\!\} \cdot \mathbf{n}^+ + \frac{1}{2} \lambda \left(\mathbf{u}_h^+ - \mathbf{u}_h^-\right)\right) \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_D} \int_F \mathbf{v}_h \cdot \left(\frac{1}{2} \left(\underline{\mathbf{f}(\mathbf{u}_h)} + \underline{\mathbf{f}(\mathbf{g})}\right) \cdot \mathbf{n} + \frac{1}{2} \lambda \left(\mathbf{u}_h - \mathbf{g}\right)\right) \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_N^{in}} \int_F \mathbf{v}_h \cdot \mathbf{h}^{in} \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_N^{out}} \int_F \mathbf{v}_h \cdot \left(\underline{\mathbf{f}(\mathbf{u}_h)} \cdot \mathbf{n} + \mathbf{h}^{out}\right) \mathrm{d}s.
\end{aligned}
\tag{2.25}
$$

We have now obtained the semi-discrete form of (2.9). To obtain a fully discrete formulation of (2.9) we still need to discretize in time. For this, we can use any time-stepping method, for example, Euler's method or the trapezoidal rule. In the case of Euler's method, the fully discrete problem is finding $\mathbf{u}_h^{n+1} \in \mathbf{V}_h^k$ such that:

$$
\begin{aligned}
0 = & \sum_{K\in\mathcal{T}} \int_K \frac{1}{\Delta t}\left(\mathbf{u}_h^{n+1} - \mathbf{u}_h^n\right)\cdot \mathbf{v}_h \mathrm{d}x \\
& + \sum_{K\in\mathcal{T}} \int_K \left(-\underline{f(\mathbf{u}_h^n)} : \nabla \mathbf{v}_h + \epsilon \nabla \mathbf{u}_h^n : \nabla \mathbf{v}_h\right)\mathrm{d}x \\
& - \sum_{F\in\mathcal{F}_I} \int_F \epsilon [\![\mathbf{u}_h^n \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{v}_h\}\!\}\mathrm{d}s - \sum_{F\in\mathcal{F}_I} \int_F \epsilon [\![\mathbf{v}_h \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{u}_h^n\}\!\}\mathrm{d}s \\
& - \sum_{F\in\mathcal{F}_D} \int_F \epsilon\left(\mathbf{u}_h^n - \mathbf{g}^n\right)\otimes \mathbf{n} : \nabla \mathbf{v}_h \mathrm{d}s - \sum_{F\in\mathcal{F}_D} \int_F \epsilon \mathbf{v}_h \otimes \mathbf{n} : \nabla \mathbf{u}_h^n \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_I} \int_F \frac{\epsilon\beta}{h} [\![\mathbf{u}_h^n \otimes \mathbf{n}]\!] : [\![\mathbf{v}_h \otimes \mathbf{n}]\!]\mathrm{d}s + \sum_{F\in\mathcal{F}_D} \int_F \frac{\epsilon\beta}{h}\left(\mathbf{u}_h^n - \mathbf{g}^n\right)\cdot \mathbf{v}_h \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_I} \int_F \left(\mathbf{v}_h^+ - \mathbf{v}_h^-\right)\cdot\left(\frac{1}{2}\{\!\{\underline{\mathbf{f}(\mathbf{u}_h^n)}\}\!\}\cdot \mathbf{n}^+ + \frac{1}{2}\lambda^n\left((\mathbf{u}_h^n)^+ - (\mathbf{u}_h^n)^-\right)\right)\mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_D} \int_F \mathbf{v}_h \cdot\left(\frac{1}{2}\left(\underline{\mathbf{f}(\mathbf{u}_h^n)} + \underline{\mathbf{f}(\mathbf{g}^n)}\right)\cdot \mathbf{n} + \frac{1}{2}\lambda^n\left(\mathbf{u}_h^n - \mathbf{g}^n\right)\right)\mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_N^{in}} \int_F \mathbf{v}_h \cdot (\mathbf{h}^{in})^n \mathrm{d}s \\
& + \sum_{F\in\mathcal{F}_N^{out}} \int_F \mathbf{v}_h \cdot\left(\underline{\mathbf{f}(\mathbf{u}_h^n)}\cdot \mathbf{n} + (\mathbf{h}^{out})^n\right)\mathrm{d}s.
\end{aligned} \tag{2.26}
$$

Chapter 3

H(div)-conforming Finite Element Method for Incompressible Navier–Stokes Flow

In this chapter, we consider finite element methods for the incompressible Navier–Stokes equations. Let $\nu \in \mathbb{R}^+$ be the kinematic viscosity and let $\mathbf{f} : \Omega \times I \to \mathbb{R}^d$ be a given forcing term. All other notation is defined similarly in section 2.2. The incompressible Navier–Stokes problem for the velocity field $\mathbf{u} : \Omega \times I \to \mathbb{R}^d$ and kinematic pressure field $p : \Omega \times I \to \mathbb{R}$ are given by

$$\partial_t \mathbf{u} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) - \nu \Delta \mathbf{u} + \nabla p = \mathbf{f} \qquad \text{in } \Omega \times I, \tag{3.1a}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega \times I, \tag{3.1b}$$

$$\mathbf{u} = \mathbf{g} \qquad \text{on } \Gamma_D \times I, \tag{3.1c}$$

$$(\mathbf{u} \otimes \mathbf{u} - \nu \nabla \mathbf{u} + p\mathbb{I}) \cdot \mathbf{n} - \max(\mathbf{u} \cdot \mathbf{n}, 0)\mathbf{u} = \mathbf{h} \qquad \text{on } \Gamma_N \times I. \tag{3.1d}$$

The nonlinear term $\nabla \cdot (\mathbf{u} \otimes \mathbf{u})$ and the term $\nu \Delta \mathbf{u}$ in (3.1a) represent convective and viscous forces of the fluid respectively. The forcing function $\mathbf{f}$, represents forces such as gravity, buoyancy, and centrifugal forces. The incompressibility constraint is $\nabla \cdot \mathbf{u} = 0$, also known as the conservation of the mass. If $\Gamma_N = \emptyset$, i.e., $\partial\Omega = \Gamma_D$, then the Dirichlet boundary data $\mathbf{g}$ must satisfy the compatibility condition

$$\int_{\partial\Omega} \mathbf{g} \cdot \mathbf{n} \mathrm{d}x = 0, \tag{3.2}$$

and the pressure mean is set to zero.

3.1 $H(\text{div})$-conforming Finite Element Spaces

The Navier–Stokes equation contains the constraint $\nabla \cdot \mathbf{u} = 0$, where $\mathbf{u}$ is the flow velocity of the system. This constraint is not enforced exactly in all numerical schemes, for example, Taylor–Hood elements [8]. One of the main motivations of this thesis is to investigate a numerical scheme that satisfies this constraint exactly. We will first consider an $H(\text{div})$ finite element space defined on a domain Ω. The $H(\text{div})$ space is defined as:

$$H(\text{div}, \Omega) := \left\{ \mathbf{u} \in L^2(\Omega)^2 \mid \nabla \cdot \mathbf{u} \in L^2(\Omega) \right\}, \tag{3.3}$$

which is the space of square integrable vector functions with square integrable divergence [17]. Examples of $H(\text{div})$-conforming finite element spaces include Raviart–Thomas (RT) [63] and Brezzi–Douglas–Marini (BDM) [7]. The $H(\text{div}, \Omega)$ conforming BDM space of the lowest order is defined in [6]. BDM spaces of order k are denoted by BDM_h^k with approximating polynomials of total degree less than or equal to k. By choosing our test and trial functions to belong to an $H(\text{div})$ conforming finite element space and choosing our pressure test and trial functions to belong to an L^2 function space, we enforce the constraint of $\mathbf{u}$ being divergence free in our weak formulation as shown in Section 3.3, and as a consequence $H(\text{div})$-conforming numerical methods are pointwise divergence free within numerical precision. Without this constraint, laws beyond mass conservation could be violated in the numerical approximation, such as energy conservation [61].

As shown in [7], when using an $H(\text{div})$ conforming finite element space, piecewise vector functions are continuous with respect to the vector normal component across adjacent facets. That is, if $\mathbf{u} \in V_h \subset H(\text{div}, \Omega)$, with V_h the BDM or RT space, then $\mathbf{u}_a \cdot \mathbf{n}_a = \mathbf{u}_b \cdot \mathbf{n}_b$ on facets, where the subscripts a and b denote adjacent elements. Since $\mathbf{n}_a = -\mathbf{n}_b$, more formally, we can say that $[\![\mathbf{u} \cdot \mathbf{n}]\!] = 0$ on facets which means that the dot product of $\mathbf{u}$ with the normal vector $\mathbf{n}$ is continuous across facets. Examples of $H(\text{div})$ methods can be found in [16, 34].

3.2 Pressure-Robustness

When using mixed finite elements such as Taylor–Hood elements for the incompressible Stokes equations to solve for a velocity field $\mathbf{u}$ and a pressure p, the velocity error between the exact velocity $\mathbf{u}$ and the discrete velocity $\mathbf{u}_h$ is *pressure-dependent* [41]:

$$\|\nabla(\mathbf{u} - \mathbf{u}_h)\|_{L^2} \leq C_1 \inf_{\mathbf{w}_h \in V(\mathcal{T})} \|\nabla(\mathbf{u} - \mathbf{w}_h)\|_{L^2} + \frac{C_2}{\nu} \inf_{q_h \in Q(\mathcal{T})} \|p - q_h\|_{L^2}, \tag{3.4}$$

where $V(\mathcal{T})$ and $Q(\mathcal{T})$ denote the discrete velocity and discrete pressure trial spaces, $\|\cdot\|_{L^2}$ denotes the L^2 norm, ν is the kinematic viscosity and C_1 and C_2 are constants. One of the immediate concerns which is typical of classical mixed methods that are not divergence-free is that the error in the approximate velocity field is dependent on the pressure field and the inverse of the viscosity.

An alternative class of numerical methods are *pressure-robust* methods. These methods have the advantage of behaving robustly regarding strong pressure gradients when the kinematic viscosity is large. Notable methods to construct such schemes include H^1 conforming and divergence-free mixed methods [31] and inf-sup stable $H(\text{div})$-conforming DG methods [13]. In this chapter, we consider this latter class of DG methods.

3.3 $H(\text{div})$-conforming Discontinuous Galerkin Method

For the velocity approximation, $\mathbf{u}_h$, we consider BDM function spaces defined in [7]. Let $\mathbf{u}_h, \mathbf{v}_h \in BDM_h^k$, and for the pressure approximation, let $p_h, q_h \in Q_h^{k-1}$, where

$$Q_h^{k-1} = \left\{ q_h \in L^2(\Omega) : \forall K \in \mathcal{T}(\Omega), q_h \mid_K \in \mathbb{P}^{k-1}(K) \right\}. \tag{3.5}$$

The weak formulation for the divergence equation can be obtained by multiplying (3.1b) by a test function $q_h \in Q_h^{k-1}$ and integrating over all elements of the domain,

$$\sum_{K \in \mathcal{T}} \int_K q_h \nabla \cdot \mathbf{u}_h \mathrm{d}x = 0. \tag{3.6}$$

The pair, $BDM_h^k \backslash Q_h^{k-1}$ forms an inf-sup stable finite element pair that furthermore has the property that $\nabla \cdot BDM_h^k = Q_h^{k-1}$. Since $\mathbf{u}_h \in BDM_h^k$, we have that $\nabla \cdot \mathbf{u}_h \in Q_h^{k-1}$. Since our weak formulation must be satisfied for all $q_h \in Q_h^{k-1}$ choosing $q_h = \nabla \cdot \mathbf{u}_h$ we have

$$\sum_{K \in \mathcal{T}} \int_K (\nabla \cdot \mathbf{u}_h)^2 \mathrm{d}x = 0 \implies \nabla \cdot \mathbf{u}_h = 0. \tag{3.7}$$

Thus our numerical approximation $\mathbf{u}_h$ is divergence-free. Following similar steps in section 2.2.1 where we found the DG weak formulation for the advection-diffusion equation, the

DG weak formulation for (3.1a) is given by:

$$\begin{aligned}\sum_{K\in\mathcal{T}}\int_K \mathbf{f}\cdot\mathbf{v}_h\mathrm{d}x &= \sum_{K\in\mathcal{T}}\int_K \left(\mathbf{v}_h\cdot\partial_t\mathbf{u}_h-(\mathbf{u}_h\otimes\mathbf{u}_h):\nabla\mathbf{v}_h+\nu\nabla\mathbf{u}_h:\nabla\mathbf{v}_h-p_h\nabla\cdot\mathbf{v}_h\right)\mathrm{d}x\\ &+\sum_{F\in\mathcal{F}_I}\int_F (\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}})\cdot(\mathbf{v}_h^+-\mathbf{v}_h^-)\mathrm{d}s+\sum_{F\in\mathcal{F}_B}\int_F(\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}})\cdot\mathbf{v}_h\mathrm{d}s\\ &-\sum_{F\in\mathcal{F}_I}\int_F [\![\mathbf{u}_h\otimes\mathbf{n}]\!]:\{\!\{\nabla\mathbf{v}_h\}\!\}\mathrm{d}s-\sum_{F\in\mathcal{F}_D}\int_F(\mathbf{u}_h-\mathbf{g})\otimes\mathbf{n}:\nabla\mathbf{v}_h\mathrm{d}s\\ &-\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{v}_h\otimes\mathbf{n}]\!]:\nu\underline{\boldsymbol{\sigma}_h}\mathrm{d}s-\sum_{F\in\mathcal{F}_B}\int_F\mathbf{v}_h\otimes\mathbf{n}:\nu\underline{\boldsymbol{\sigma}_h}\mathrm{d}s\\ &-\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{v}_h\otimes\mathbf{n}]\!]:\overline{p_h\mathbb{I}}\mathrm{d}s-\sum_{F\in\mathcal{F}_B}\int_F\mathbf{v}_h\otimes\mathbf{n}:\overline{p_h\mathbb{I}}\mathrm{d}s,\end{aligned}\tag{3.8}$$

where $\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}},\underline{\boldsymbol{\sigma}_h}$ and $\overline{p_h\mathbb{I}}$ are approximations to, respectively, $(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n},\underline{\boldsymbol{\sigma}_h}$ and $p_h\mathbb{I}$ and $\mathbf{u}_h,\ \mathbf{v_h}\in H(\mathrm{div})$. We will now choose our numerical fluxes and simplify (3.8). First we will consider the integrals involving the flux of the pressure:

$$\begin{aligned}&\sum_{F\in\mathcal{F}_I}\int_F[\![\mathbf{v}_h\otimes\mathbf{n}]\!]:\overline{p_h\mathbb{I}}\mathrm{d}s+\sum_{F\in\mathcal{F}_D}\int_F\mathbf{v}_h\otimes\mathbf{n}:\overline{p_h\mathbb{I}}\mathrm{d}s+\sum_{F\in\mathcal{F}_N}\int_F\mathbf{v}_h\otimes\mathbf{n}:\overline{p_h\mathbb{I}}\mathrm{d}s\\ &=\sum_{F\in\mathcal{F}_I}\int_F\left(\mathbf{v}_h^+-\mathbf{v}_h^-\right)\cdot\mathbf{n}^+\overline{p_h}\mathrm{d}s+\sum_{F\in\mathcal{F}_D}\int_F\mathbf{v}_h\cdot\mathbf{n}\overline{p_h}\mathrm{d}s+\sum_{F\in\mathcal{F}_N}\int_F\mathbf{v}_h\cdot\mathbf{n}\overline{p_h}\mathrm{d}s\\ &=\sum_{F\in\mathcal{F}_N}\int_F\mathbf{v}_h\cdot\mathbf{n}p_h\mathrm{d}s,\end{aligned}\tag{3.9}$$

where we set $\overline{p_h}=p_h$ on $F\in\mathcal{F}_N$ and where the last equality is due to the single valuedness of $\overline{p_h}$ and $\mathbf{v}_h\cdot\mathbf{n}$ on interior facets, and since $\mathbf{v_h}\cdot\mathbf{n}=0$ on Dirichlet boundary facets. For $\underline{\boldsymbol{\sigma}_h}$ we will again choose the IP flux as in equation (2.20). For $\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}}$ we will choose the usual upwind flux which is given by:

$$\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}}=\{\!\{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}\}\!\}+\frac{1}{2}|\mathbf{u}\cdot\mathbf{n}|(\mathbf{u}_h^+-\mathbf{u}_h^-).\tag{3.10}$$

Using the above defined fluxes for $\overline{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}},\underline{\boldsymbol{\sigma}_h}$ and $\overline{p_h\mathbb{I}}$ as well as the Neumann boundary condition (3.1d), we obtain the following discontinuous Galerkin weak formulation for the momentum equation (3.1a):

$$
\begin{aligned}
&\sum_{K\in\mathcal{T}}\int_K \mathbf{f}\cdot\mathbf{v}_h \mathrm{d}x = \\
&\sum_{K\in\mathcal{T}}\int_K \left(\mathbf{v}_h\cdot\partial_t\mathbf{u}_h - \mathbf{u}_h\otimes\mathbf{u}_h : \nabla\mathbf{v}_h + \nu\nabla\mathbf{u}_h : \nabla\mathbf{v}_h - p_h\nabla\cdot\mathbf{v}_h\right)\mathrm{d}x \\
&-\sum_{F\in\mathcal{F}_I}\int_F \nu[\![\mathbf{u}_h\otimes\mathbf{n}]\!] : \{\!\{\nabla\mathbf{v}_h\}\!\}\mathrm{d}s - \sum_{F\in\mathcal{F}_I}\int_F \nu[\![\mathbf{v}_h\otimes\mathbf{n}]\!] : \{\!\{\nabla\mathbf{u}_h\}\!\}\mathrm{d}s \\
&-\sum_{F\in\mathcal{F}_D}\int_F \nu\left(\mathbf{u}_h-\mathbf{g}\right)\otimes\mathbf{n} : \nabla\mathbf{v}_h\mathrm{d}s - \sum_{F\in\mathcal{F}_D}\int_F \nu\mathbf{v}_h\otimes\mathbf{n} : \nabla\mathbf{u}_h\mathrm{d}s \\
&+\sum_{F\in\mathcal{F}_I}\int_F \frac{\nu\beta}{h}[\![\mathbf{u}_h\otimes\mathbf{n}]\!] : [\![\mathbf{v}_h\otimes\mathbf{n}]\!]\mathrm{d}s + \sum_{F\in\mathcal{F}_D}\int_F \frac{\nu\beta}{h}\left(\mathbf{u}_h-\mathbf{g}\right)\cdot\mathbf{v}_h\mathrm{d}s \\
&+\sum_{F\in\mathcal{F}_I}\int_F \left(\mathbf{v}_h^+ - \mathbf{v}_h^-\right)\cdot\left(\{\!\{(\mathbf{u}_h\otimes\mathbf{u}_h)\cdot\mathbf{n}\}\!\} + \frac{1}{2}|\mathbf{u}_h\cdot\mathbf{n}|\left(\mathbf{u}_h^+ - \mathbf{u}_h^-\right)\right)\mathrm{d}s \\
&+\sum_{F\in\mathcal{F}_D}\int_F \mathbf{v}_h\cdot\left((\mathbf{u}_h\otimes\mathbf{u}_h + \mathbf{g}\otimes\mathbf{g})\cdot\mathbf{n} + \frac{1}{2}|\mathbf{u}_h\cdot\mathbf{n}|\left(\mathbf{u}_h-\mathbf{g}\right)\right)\mathrm{d}s \\
&+\sum_{F\in\mathcal{F}_N}\int_F \mathbf{v}_h\cdot\left(\mathbf{h} + \max(\mathbf{u}_h\cdot\mathbf{n},0)\mathbf{u}_h\right)\mathrm{d}s.
\end{aligned}
\tag{3.11}
$$

The semi-discrete $H(\mathrm{div})$ discretization is given by (3.6) and (3.11). To obtain a fully-discrete discretization we use the unconditionally stable, second-order accurate in time, trapezoidal rule to discretize the equations in time and in which the convective velocity is approximated by a linear combination of $\mathbf{u}$ at the previous time step: $\mathbf{u}^\star = \frac{3}{2}\mathbf{u}^n - \frac{1}{2}\mathbf{u}^{n-1}$. Let $y^{n+\frac{1}{2}} = \frac{1}{2}y^n + \frac{1}{2}y^{n+1}$. Then, the fully discrete discontinuous Galerkin weak formulation is given by: Find $(\mathbf{u}_h, p_h) \in BDM_h^k \times Q_h^{k-1}$ such that for all $(\mathbf{v}_h, q_h) \in BDM_h^k \times Q_h^{k-1}$ the following holds:

$$
\begin{aligned}
0 = &\sum_{K\in\mathcal{T}} \int_K \frac{1}{\Delta t}\left(\mathbf{u}_h^{n+1} - \mathbf{u}_h^n\right)\cdot \mathbf{v}_h \mathrm{d}x \\
&+ \sum_{K\in\mathcal{T}} \int_K \left(-\mathbf{u}_h^{n+\frac{1}{2}} \otimes \mathbf{u}_h^{\star} : \nabla \mathbf{v}_h + \nu \nabla \mathbf{u}_h^{n+\frac{1}{2}} : \nabla \mathbf{v}_h - p_h^{n+\frac{1}{2}} \nabla\cdot \mathbf{v}_h\right)\mathrm{d}x \\
&- \sum_{F\in\mathcal{F}_I} \int_F \nu [\![\mathbf{u}_h^{n+\frac{1}{2}} \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{v}_h\}\!\}\mathrm{d}s - \sum_{F\in\mathcal{F}_I} \int_F \nu [\![\mathbf{v}_h \otimes \mathbf{n}]\!] : \{\!\{\nabla \mathbf{u}_h^{n+\frac{1}{2}}\}\!\}\mathrm{d}s \\
&- \sum_{F\in\mathcal{F}_D} \int_F \nu \left(\mathbf{u}_h^{n+\frac{1}{2}} - \mathbf{g}\right)\otimes \mathbf{n} : \nabla \mathbf{v}_h \mathrm{d}s - \sum_{F\in\mathcal{F}_D} \int_F \nu \mathbf{v}_h \otimes \mathbf{n} : \nabla \mathbf{u}_h^{n+\frac{1}{2}} \mathrm{d}s \\
&+ \sum_{F\in\mathcal{F}_I} \int_F \frac{\nu\beta}{h} [\![\mathbf{u}_h^{n+\frac{1}{2}} \otimes \mathbf{n}]\!] : [\![\mathbf{v}_h \otimes \mathbf{n}]\!]\mathrm{d}s + \sum_{F\in\mathcal{F}_D} \int_F \frac{\nu\beta}{h}\left(\mathbf{u}_h^{n+\frac{1}{2}} - \mathbf{g}\right)\cdot \mathbf{v}_h \mathrm{d}s \\
&+ \sum_{F\in\mathcal{F}_I} \int_F \left(\mathbf{v}_h^+ - \mathbf{v}_h^-\right)\cdot\left(\{\!\{(\mathbf{u}_h^{n+\frac{1}{2}} \otimes \mathbf{u}_h^{\star})\cdot \mathbf{n}\}\!\} + \frac{1}{2}|\mathbf{u}_h^{\star}\cdot \mathbf{n}|\left((\mathbf{u}_h^{n+\frac{1}{2}})^+ - (\mathbf{u}_h^{n+\frac{1}{2}})^-\right)\right)\mathrm{d}s \\
&+ \sum_{F\in\mathcal{F}_D} \int_F \mathbf{v}_h \cdot \left((\mathbf{u}_h^{n+\frac{1}{2}} \otimes \mathbf{u}_h^{\star} - \mathbf{g}^{n+\frac{1}{2}} \otimes \mathbf{g}^{n+\frac{1}{2}})\cdot \mathbf{n}^+ + \frac{1}{2}|\mathbf{u}_h^{n+\frac{1}{2}}\cdot \mathbf{n}|\left(\mathbf{u}_h^{\star} - \mathbf{g}^{n+\frac{1}{2}}\right)\right)\mathrm{d}s \\
&+ \sum_{F\in\mathcal{F}_N} \int_F \mathbf{v}_h \cdot \left(\mathbf{h} + \max(\mathbf{u}_h^{\star}\cdot \mathbf{n}, 0)\mathbf{u}_h^{n+\frac{1}{2}}\right)\mathrm{d}s - \sum_{K\in\mathcal{T}} \int_K \mathbf{f}^{n+\frac{1}{2}} \cdot \mathbf{v}_h \mathrm{d}x,
\end{aligned}
\tag{3.12}
$$

$$
\sum_{K\in\mathcal{T}} \int_K q_h \nabla\cdot \mathbf{u}_h^{n+\frac{1}{2}} \mathrm{d}x = 0. \tag{3.13}
$$

3.4 Numerical Examples

We will now present some numerical examples to demonstrate some of the $H(\mathrm{div})$ discretization properties. All examples in this section have been implemented in NGSolve [66].

3.4.1 Example 1: Unit Square

For the first test case, we consider our domain $\Omega := [0, 1]^2$ to be the unit square. Dirichlet boundary conditions are prescribed along the left and top edges, while Neumann boundary conditions are prescribed along the right and bottom edges. As exact solutions we choose $\mathbf{u} = (u_x, u_y)$ and p where

$$\begin{aligned} u_x &= \sin(\pi x - t)\sin(\pi y - t), \\ u_y &= \cos(\pi x - t)\cos(\pi y - t), \\ p &= \sin(\pi x)\cos(\pi y). \end{aligned} \tag{3.14}$$

In this test problem, we set the Dirichlet boundary data $\mathbf{g}$ equal to the exact solution $\mathbf{u}$ in (3.1c). Likewise the Neumann boundary data $\mathbf{h}$ in (3.1d) and source term $\mathbf{f}$ in (3.1a) are calculated from the exact solution of $\mathbf{u}$ and p. The NGSolve code is provide in Appendix A.1.

The *computational order of convergence* [15] of each of these quantities is calculated as follows. Suppose that $e(N_1)$ and $e(N_2)$ is any quantity for two consecutive triangulations with respectively N_1 and N_2 number of triangles. Then the computational rate of convergence is given by

$$-2\frac{\log(e(N_1)/e(N_2))}{\log(N_1/N_2)}. \tag{3.15}$$

In Table 3.1, we present the numerical results for the smooth solution. For this test case we choose $\nu = 10^{-5}$. A visualization of the initial pressure and velocity field is shown in Figure 3.1. A time step of $\Delta t = 10^{-6}$ is chosen sufficiently small to not affect the convergence rates with respect to the mesh size. Table 3.1 also shows that approximation polynomials of order k and $k-1$ for the velocity and pressure respectively produce a convergence rate of $k+1$ for the velocity error and a rate of convergence of k for the pressure error. As for our approximation velocity divergence, we see that the magnitude is slightly increasing as mesh size N increases with a magnitude of machine precision. Thus, we can conclude that our discrete formulation produces an approximate divergence-free velocity field with optimal convergence rates for the velocity and pressure.

In Table 3.2, we display the behaviour of the error for the smooth solution when $\nu = 10^5$. We once again observe that for all meshes and $k \geq 1$, similar convergence rates for the velocity field and pressure. The approximate velocity field is pointwise divergence-free. By comparing Tables 3.1 and 3.2 we observe that an increase in viscosity increases the error in the pressure but not the velocity field, concluding the BDM finite element numerical scheme is pressure-robust with respect to mesh size and polynomial degree. Similar simulations have been done using Taylor–Hood elements in Tables 3.3 and 3.4, where the convergence of the pressure as well as the error in the divergence of the velocity is worse. Furthermore, we see that the velocity error is $100 - 1000$ times larger when $\nu = 10^{-5}$ compared to $\nu = 10^5$. The results in Tables 3.1 - 3.4 clearly demonstrate the superiority of the pressure-robust BDM/DG scheme over the non-pressure-robust Taylor–Hood scheme.

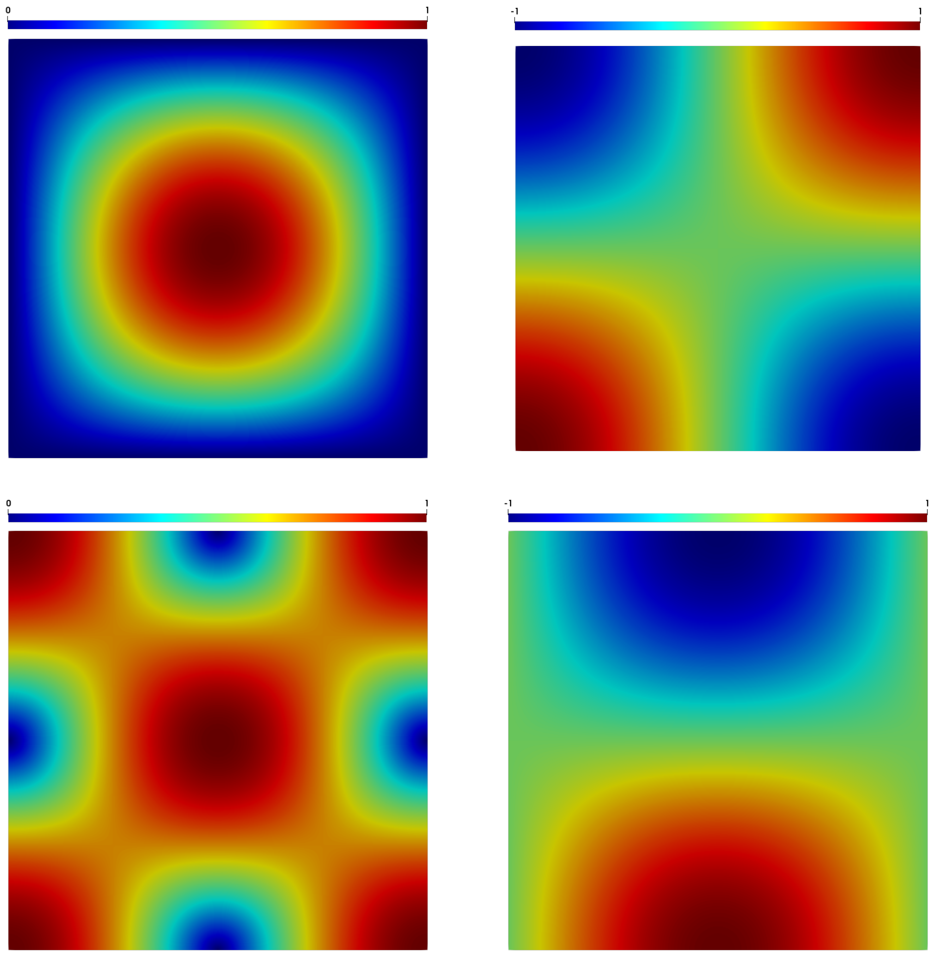

Figure 3.1: The horizontal (*top left*) and vertical (*top right*) components of the approximate velocity, velocity magnitude (*bottom left*) and pressure (*bottom right*) obtained using $BDM_h^4 \backslash Q_h^3$ on a mesh with $N = 2048$ at $t = 0$.

Table 3.1: BDM elements history of convergence using of the error $\|u_h - u\|_{L^2}$, $\|p_h - p\|_{L^2}$ and $\|\nabla \cdot u_h\|_{L^2}$. Uniform mesh refinement, smooth solution, $\nu = 10^{-5}$.

$BDM_h^1 \backslash Q_h^0$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	1.3E − 01	–	1.9E − 01	–	1.6E − 12
32	9.1E − 02	0.5	1.4E − 01	0.5	3.4E − 12
128	2.5E − 02	1.9	6.6E − 02	1.0	6.1E − 12
512	6.5E − 03	1.9	3.3E − 02	1.0	1.2E − 11
2048	1.7E − 03	2.0	1.6E − 02	1.0	2.4E − 11
$BDM_h^2 \backslash Q_h^1$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	9.6E − 02	–	1.1E − 01	–	2.5E − 12
32	9.7E − 03	3.3	2.0E − 02	2.5	3.6E − 12
128	1.3E − 03	2.9	5.0E − 03	2.0	7.7E − 12
512	1.7E − 04	3.0	1.2E − 03	2.0	1.5E − 11
2048	2.2E − 05	3.0	3.1E − 04	2.0	3.1E − 11
$BDM_h^3 \backslash Q_h^2$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	3.2E − 03	–	5.1E − 03	–	2.7E − 12
32	7.7E − 04	2.0	2.2E − 03	1.2	5.0E − 12
128	4.6E − 05	4.0	2.7E − 04	3.0	9.2E − 12
512	2.8E − 06	4.0	3.4E − 05	3.0	1.9E − 11
2048	1.8E − 07	4.0	4.3E − 06	3.0	3.8E − 11
$BDM_h^4 \backslash Q_h^3$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	2.6E − 03	–	7.9E − 04	–	3.2E − 12
32	6.1E − 05	5.4	2.7E − 05	4.9	5.7E − 12
128	2.0E − 06	4.9	1.4E − 06	4.3	1.1E − 12
512	6.5E − 08	5.0	8.1E − 08	4.1	2.3E − 11
2048	2.0E − 09	5.0	5.4E − 09	3.9	4.4E − 11

Table 3.2: BDM elements history of convergence of the error $\|u_h - u\|_{L^2}$, $\|p_h - p\|_{L^2}$ and $\|\nabla \cdot u_h\|_{L^2}$. Uniform mesh refinement, smooth solution, $\nu = 10^5$.

$BDM_h^1\backslash Q_h^0$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$9.6\text{E} - 02$	–	$4.3\text{E} + 05$	–	$1.8\text{E} - 12$
32	$9.1\text{E} - 02$	0.1	$1.7\text{E} + 06$	-2.0	$3.0\text{E} - 12$
128	$2.5\text{E} - 02$	1.9	$7.2\text{E} + 05$	1.2	$6.1\text{E} - 12$
512	$6.5\text{E} - 03$	2.0	$3.3\text{E} + 05$	1.1	$1.2\text{E} - 11$
2048	$1.6\text{E} - 03$	2.0	$1.6\text{E} + 05$	1.1	$2.4\text{E} - 11$
$BDM_h^2\backslash Q_h^1$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$9.5\text{E} - 02$	–	$2.6\text{E} + 06$	–	$2.2\text{E} - 12$
32	$9.7\text{E} - 03$	3.3	$4.8\text{E} + 05$	2.4	$4.0\text{E} - 12$
128	$1.3\text{E} - 03$	2.9	$1.1\text{E} + 05$	2.1	$7.6\text{E} - 12$
512	$1.7\text{E} - 04$	3.0	$2.7\text{E} + 04$	2.1	$1.5\text{E} - 11$
2048	$2.2\text{E} - 05$	3.0	$6.4\text{E} + 03$	2.1	$3.1\text{E} - 11$
$BDM_h^3\backslash Q_h^2$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$3.2\text{E} - 03$	–	$1.0\text{E} + 05$	–	$2.4\text{E} - 12$
32	$7.7\text{E} - 04$	2.0	$6.1\text{E} + 04$	0.8	$4.9\text{E} - 12$
128	$4.6\text{E} - 05$	4.0	$7.0\text{E} + 03$	3.1	$9.4\text{E} - 12$
512	$2.8\text{E} - 06$	4.0	$8.3\text{E} + 02$	3.1	$1.8\text{E} - 11$
2048	$1.8\text{E} - 07$	4.0	$1.0\text{E} + 02$	3.0	$3.8\text{E} - 11$
$BDM_h^4\backslash Q_h^3$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$2.6\text{E} - 03$	–	$7.9\text{E} - 04$	–	$3.2\text{E} - 12$
32	$6.1\text{E} - 05$	5.4	$2.7\text{E} - 05$	4.9	$5.7\text{E} - 12$
128	$2.0\text{E} - 06$	4.9	$1.4\text{E} - 06$	4.3	$1.1\text{E} - 12$
512	$6.5\text{E} - 08$	5.0	$8.1\text{E} - 08$	4.1	$2.3\text{E} - 11$
2048	$2.0\text{E} - 09$	5.0	$5.4\text{E} - 09$	3.9	$4.4\text{E} - 11$

Table 3.3: Taylor–Hood elements history of convergence using of the error $\|u_h - u\|_{L^2}$, $\|p_h - p\|_{L^2}$ and $\|\nabla \cdot u_h\|_{L^2}$. Uniform mesh refinement, smooth solution, $\nu = 10^{-5}$.

$P^2 \backslash P^1$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$9.5\mathrm{E}+01$	–	$1.5\mathrm{E}-01$	–	$9.3\mathrm{E}+02$
32	$1.4\mathrm{E}+01$	2.8	$3.7\mathrm{E}-02$	2.1	$2.3\mathrm{E}+02$
128	$1.6\mathrm{E}+00$	3.1	$9.1\mathrm{E}-03$	2.0	$5.5\mathrm{E}+01$
512	$1.8\mathrm{E}-01$	3.1	$2.2\mathrm{E}-03$	2.0	$1.3\mathrm{E}+01$
2048	$2.3\mathrm{E}-02$	3.0	$5.6\mathrm{E}-04$	2.0	$3.4\mathrm{E}+00$
$P^3 \backslash P^2$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$3.8\mathrm{E}+00$	–	$1.0\mathrm{E}-02$	–	$6.6\mathrm{E}+01$
32	$7.0\mathrm{E}-01$	2.5	$3.4\mathrm{E}-03$	1.6	$2.2\mathrm{E}+01$
128	$6.0\mathrm{E}-02$	3.5	$4.8\mathrm{E}-04$	2.8	$3.5\mathrm{E}+00$
512	$4.4\mathrm{E}-03$	3.8	$6.3\mathrm{E}-05$	2.9	$4.8\mathrm{E}-01$
2048	$2.9\mathrm{E}-04$	3.9	$8.1\mathrm{E}-06$	3.0	$6.3\mathrm{E}-02$
$P^4 \backslash P^3$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	$9.2\mathrm{E}-01$	–	$2.8\mathrm{E}-03$	–	$2.7\mathrm{E}+01$
32	$2.5\mathrm{E}-02$	5.2	$1.7\mathrm{E}-04$	4.1	$1.4\mathrm{E}+00$
128	$7.3\mathrm{E}-04$	5.1	$1.1\mathrm{E}-05$	4.0	$7.9\mathrm{E}-02$
512	$2.2\mathrm{E}-05$	5.0	$6.5\mathrm{E}-07$	4.0	$4.8\mathrm{E}-03$
2048	$6.8\mathrm{E}-07$	5.0	$4.1\mathrm{E}-08$	4.0	$3.0\mathrm{E}-04$

Table 3.4: Taylor–Hood elements history of convergence using of the error $\|u_h - u\|_{L^2}$, $\|p_h - p\|_{L^2}$ and $\|\nabla \cdot u_h\|_{L^2}$. Uniform mesh refinement, smooth solution, $\nu = 10^5$.

$P^2\backslash P^1$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	5.0E − 02	–	2.9E − 01	–	6.1E − 01
32	6.3E − 03	3.0	4.8E − 02	2.6	1.1E − 01
128	8.7E − 04	2.8	1.1E − 02	2.1	2.9E − 02
512	1.1E − 04	2.9	2.5E − 03	2.1	7.1E − 03
2048	1.4E − 05	3.0	6.0E − 04	2.0	1.8E − 03
$P^3\backslash P^2$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	2.5E − 03	–	2.4E − 02	–	3.0E − 02
32	4.1E − 04	2.6	4.3E − 03	2.4	1.2E − 02
128	2.5E − 05	4.0	5.6E − 04	3.0	1.4E − 03
512	1.6E − 06	4.0	7.0E − 05	3.0	1.7E − 04
2048	9.7E − 08	4.0	8.6E − 06	3.0	2.0E − 05
$P^4\backslash P^3$					
N	$\|u_h - u\|_{L^2}$	order	$\|p_h - p\|_{L^2}$	order	$\|\nabla \cdot u_h\|_{L^2}$
8	1.9E − 03	–	8.2E − 03	–	2.1E − 02
32	4.6E − 05	5.4	3.1E − 04	4.7	9.7E − 04
128	1.5E − 06	4.9	1.8E − 05	4.1	6.2E − 05
512	4.7E − 08	5.0	1.1E − 06	4.1	3.8E − 06
2048	1.7E − 09	4.8	6.4E − 08	4.0	3.0E − 07

3.4.2 Example 2: Flow Past a Cylinder

In this test case we consider laminar flow past a circular cylinder, see [59, 69]. As computational domain we take $\Omega = [0, 2] \times [0, 0.41]$ and mesh size of $N = 5856$ triangles. We compute the solution over the time interval $I = [0, 2.2]$ for which we set the time step to $\Delta t = 0.001$. Here, the obstacle chosen is a circle with radius $r = 0.05$ with a center located at $(0.2, 0.2)$, which can be seen in Figure 3.2. The Dirichlet inflow BC on the left edge is given by a parabolic inlet condition $\mathbf{u} = (u_x, u_y) = (6y(0.41 - y)/(0.41)^2, 0)$. Here, the mean inflow velocity $\bar{\mathbf{u}} = 1$, the viscosity in (3.1a) is set to $\nu = 10^{-3}$, and with a characteristic length which is the diameter of the obstruction $L = 0.1$, results in a Reynolds number $Re = \bar{\mathbf{u}}L/\nu = 100$. No-slip boundaries are prescribed on the edge of the circle as well as the top and bottom walls, and on the right edge we impose a Neumann BC with $\mathbf{h} = 0$. We consider a cubic polynomial approximation for the velocity and a quadratic polynomial approximation for the pressure. The NGSolve code is provided in Appendix A.2.

To save computation time, we first solve the Stokes flow equations, also using an $H(\text{div})$-conforming method, which is used as an initial condition for the Navier–Stokes simulation. In Figure 3.3 we have the solution to Stokes equation for laminar flow. At $t = 0.5$, we begin to see the wake structure form behind the cylinder in Figure 3.4. At $t = 2$, such a flow can be seen at a time instance where the characteristic vortex shedding of a periodic Kármán vortex street has formed, see Figure 3.5.

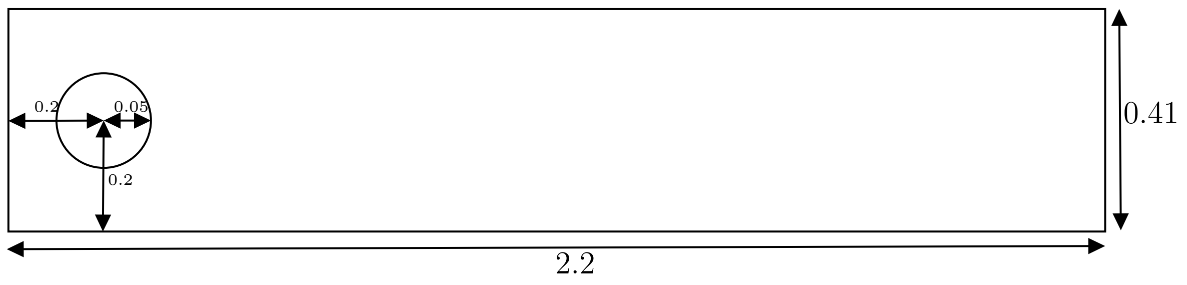

Figure 3.2: Simulation domain for single-phase flow inside a channel around a cylinder.

Table 3.5: Numerical quantities.

f	$\max c_D$	$\min c_D$	$\max c_L$	$\min c_L$	St
2.97	3.122	3.094	1.019	-1.006	0.297

3.4.3 Drag, Lift, and the Strouhal Number

In order to quantify the validity of the model for this example, the maximal and minimal drag and lift coefficients, c_D, c_L that act on the cylinder were calculated, defined as

$$c_D := \frac{1}{\bar{\mathbf{u}}^2 r} \int_{\Gamma_\circ} \left(\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} \right) \cdot e_x ds, \quad c_L := \frac{1}{\bar{\mathbf{u}}^2 r} \int_{\Gamma_\circ} \left(\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{u} \right) \cdot e_y ds. \tag{3.16}$$

Here, e_x, e_y are the unit vectors in the x and y direction, $r = 0.005$ is the radius of the cylinder, $\bar{\mathbf{u}}$ is the mean inflow velocity and $\Gamma_\circ$ denotes the surface of the cylinder. Additionally, the Strouhal number St, which is a dimensionless number describing oscillating flow mechanisms, was also measured, which is defined as

$$\mathrm{St} = \frac{2rf}{\bar{\mathbf{u}}}, \tag{3.17}$$

where f is the frequency and $1/f$ is the length of a cycle, from when c_L is smallest at time t_0 and ends at time instance $t_1 = t_0 + 1/f$ when c_L is smallest again. Table 3.5 contains the frequency of a cycle, the maximum and minimum drag and lift coefficients, and the Strouhal number. The drag and lift coefficients are compared to [37, 46], and the frequency and Strouhal number are compared in [65] which are found to be in agreement.

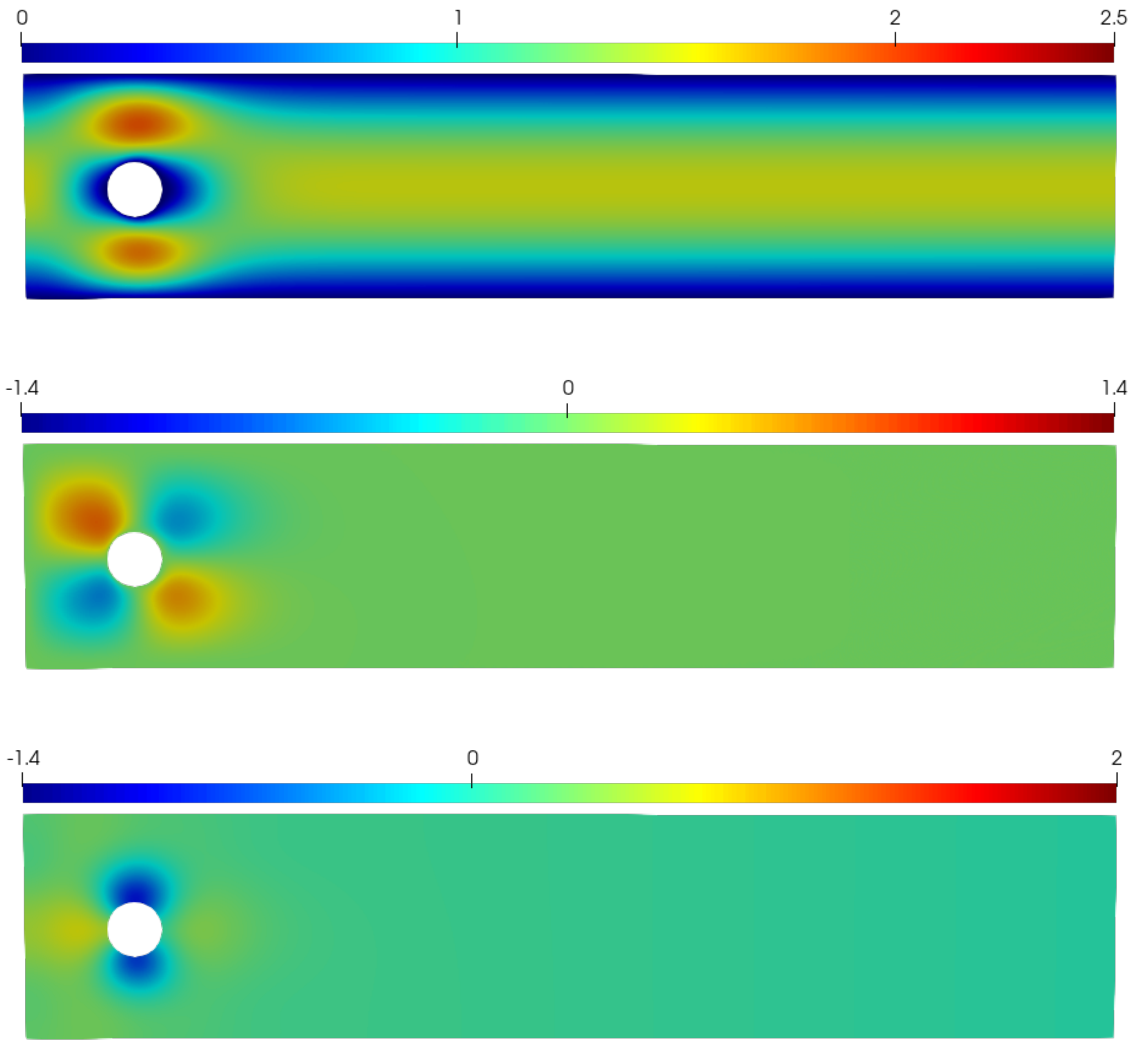

Figure 3.3: The horizontal (*top*) and vertical (*middle*) components of the approximate velocity field and pressure (*bottom*) at $t = 0$.

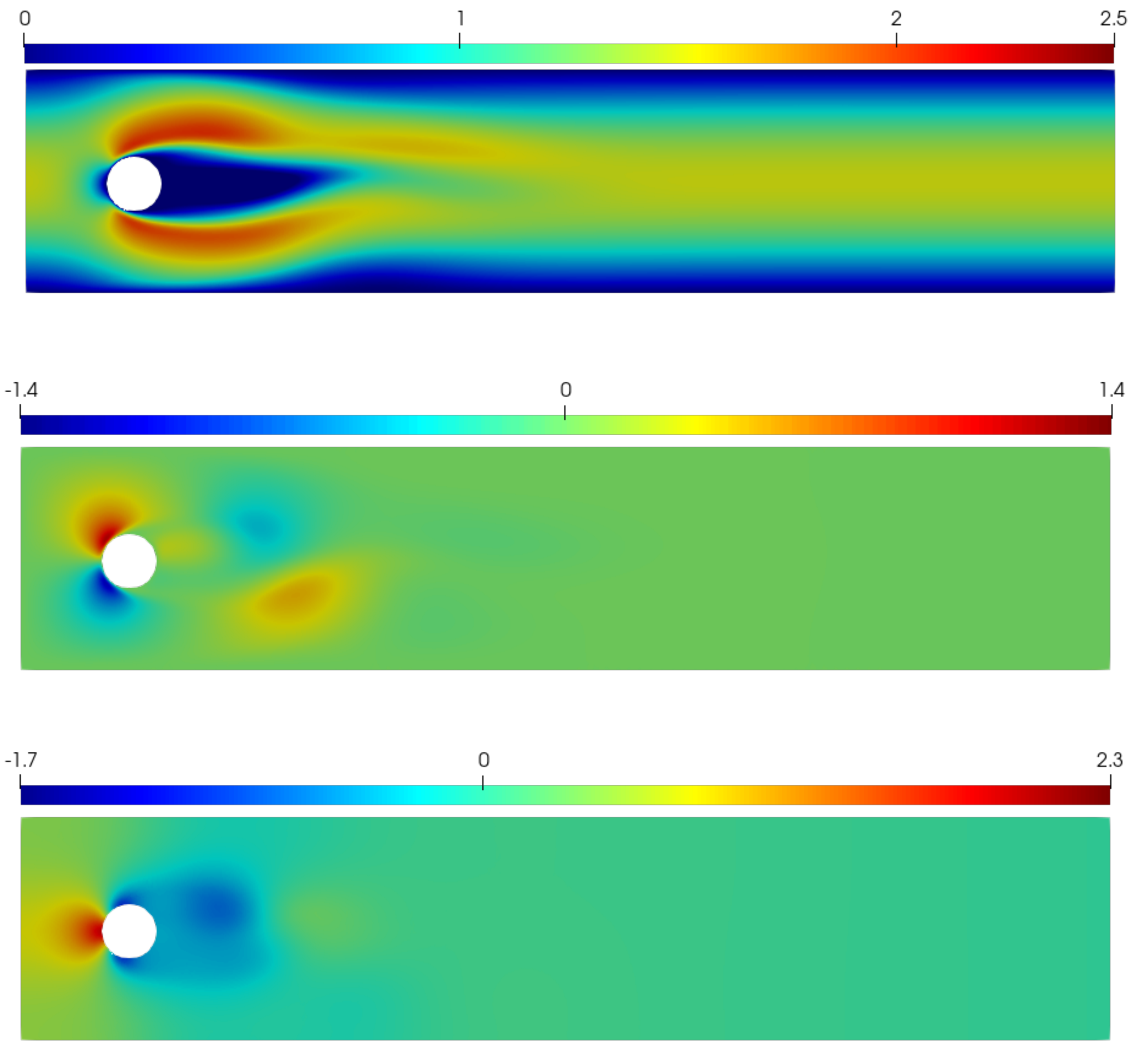

Figure 3.4: The horizontal (*top*) and vertical (*middle*) components of the approximate velocity field and pressure (*bottom*) at $t = 0.5$.

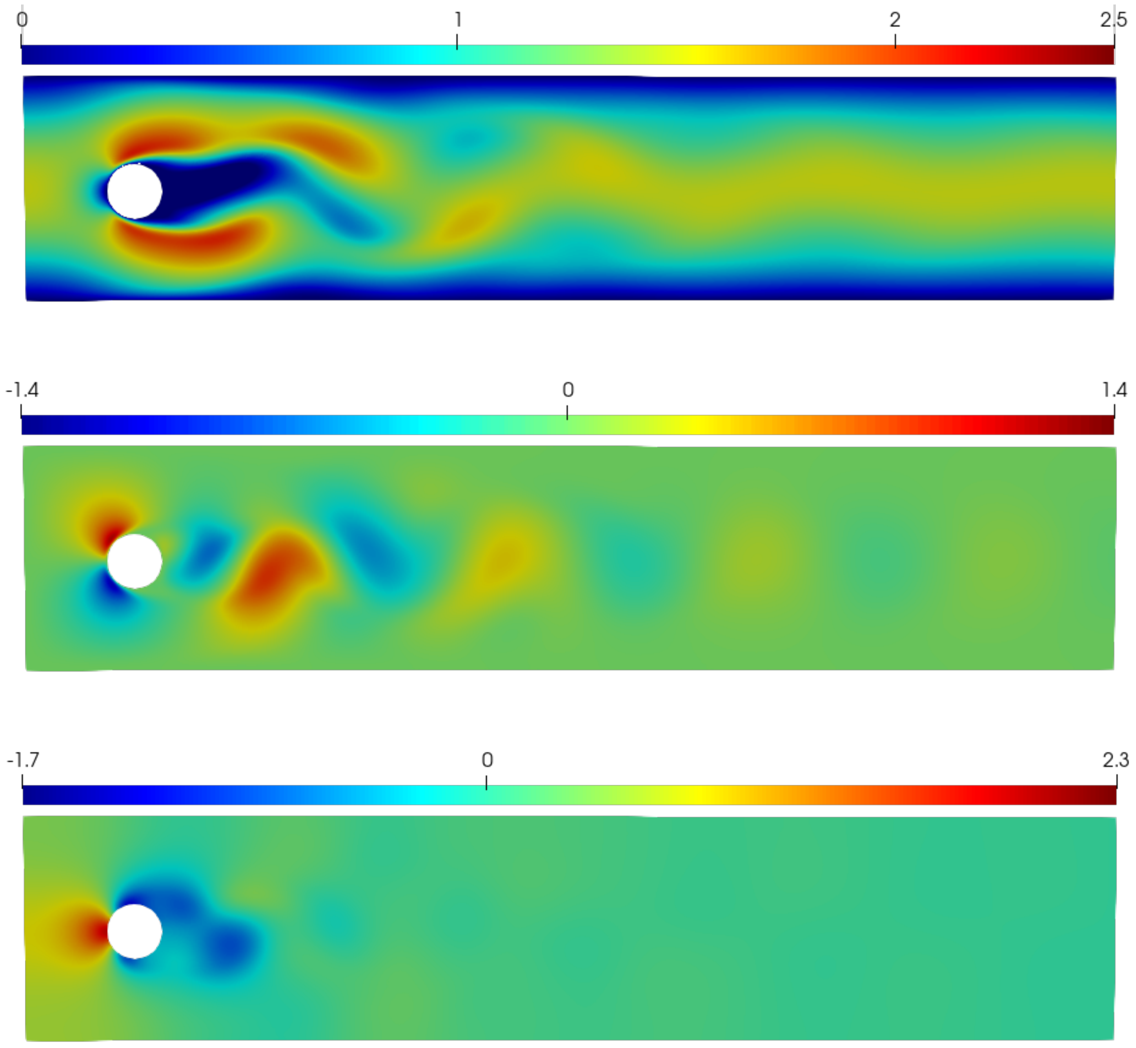

Figure 3.5: The horizontal (*top*) and vertical (*middle*) components of the approximate velocity field and pressure (*bottom*) at $t = 2$.

Chapter 4

Multiphase Flow

4.1 Background

Multiphase flow is the study of the interactions of two or more fluids of different phases or densities, often requiring the coupling of multiple equations, dramatically increasing computational costs. While the momentum and continuity equations are used as a foundation of the system, there are still questions over how to properly couple these equations using momentum exchange terms while accurately modeling the physical phenomena.

There are two basic ideas behind modeling multiphase flows. One example is trajectory models [5], where the motion of the disperse phase is modeled as individual particles or a small collection is modeled as a singular entity. These models are beneficial when modeling granular flows; however, computational costs rise as the number of individual particles increases.

Another way of modeling multiphase flows is using the Eulerian two-fluid approach, where all phases are treated formally as fluids that obey standard single-phase equations of motion, with appropriate boundary conditions specified at phase boundaries. The *macroscopic* flow equations are derived from these mesoscopic equations using an averaging procedure of some kind. There are several averaging methods such as time averaging, volume averaging, and ensemble averaging.

Two-fluid models assume each fluid as a continuous phase, coupling the basic momentum, energy, and continuity conservation equations using terms such as mass transfer or dispersion. In terms of bubbly flows, the bubbles represent a dispersed phase approximated as a continuous phase. They can be modeled using the momentum and continuity equations, but the model is ill-posed without coupling the two fluids. A solution is well-posed if a unique solution depends continuously on the initial and boundary conditions. [21]

A parameter varying from zero to one is used to describe the fluid phase fraction for all points in space, known as the phase fraction of the fluid. There have been numerous methods to change the well-posedness of the model to obtain stability, but this raises questions regarding the physical fidelity, such as boundedness of the phase fractions [75]. Some methods to ensure well-posedness over some or all ranges of volume fraction include:

- Inclusion of the interfacial pressure in the governing equations [33, 45, 77].
- Addition of a momentum flux [70].
- Including Virtual mass contribution [33, 44].
- Additional momentum transfer terms [76, 77].
- Dispersion terms dependent on the gradient of the dispersed phase fraction [54].

There is division among which methods are valid for an accurate description of the phenomena, and further research is needed to examine these effects.

The advantage of the two-fluid model is its generality. In theory, it can be applied to any multiphase system, regardless of the number and properties of the phases. A drawback of the two-fluid model approach is that it often leads to a highly complex set of flow equations and closure relations, proving difficult to solve numerically.

We will denote each phase variable F as F_q, $q \in \{c, d\}$ where c and d are the continuous and dispersed phase for some variable F. In modeling gas-liquid flows using the two-fluid model, each phase is considered a continuous fluid requiring two sets of conservation equations coupled together through interphase momentum transfer terms.

The two-fluid model can account for either dispersed-continuous or continuous-continuous phase interactions. For a dispersed-continuous phase interaction, the phase fraction may take any value between 0 and 1. Examples are when the dispersed phase is a particle (solids), droplets (liquids), or bubbles (gas) dissolved in a continuous fluid. When the dispersed phase fraction $\alpha_d = 0$, that implies that none of the dispersed phase is occupying those cells, as shown in Figure 4.1, and as the phase fraction gradually increases, the concentration of the dispersed phase in those cells gradually increases. For a continuous-continuous phase interaction, where we have a sharp interface between phases, we expect on one side of the interface the phase fraction to be $\alpha_c = 0$ where none of the phase is present, and on the other side of the interface the phase fraction to be $\alpha_c = 1$ where the phase is completely present. A gradient in the phase fraction is restricted only to the interface. We will only consider dispersed-continuous phase interactions in this thesis.

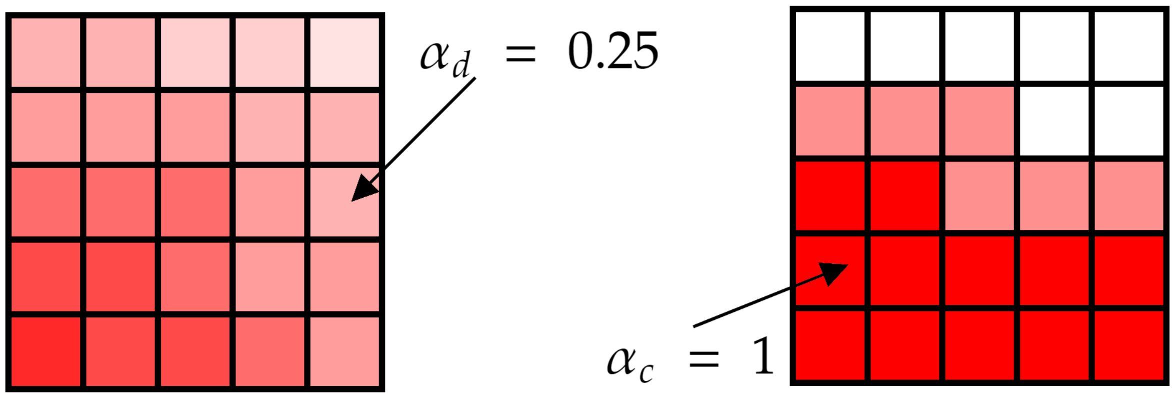

Figure 4.1: Mesh cell phase fraction values for dispersed-continuous (*left*) and continuous-continuous (*right*) phase interactions.

4.1.1 Averaging notation

To solve for the macroscopic flow behavior of the two-fluid equation, we will introduce the following averaging notation. The Eulerian time-averaged quantities, denoted by $\overline{(\cdot)}$ is defined [39]:

$$\overline{F_q}\left(\mathbf{x}_0, t_0\right) := \lim_{\delta \to 0} \frac{1}{\Delta t} \int_{[\Delta t]_q} F_q\left(\mathbf{x}_0, t\right) dt, \tag{4.1}$$

for some general function F_q. Here we have a Δt a fixed time interval, δ the interfacial thickness, $(\mathbf{x}_0, t_0)$ a reference point and time respectively, and $[\Delta t]_q = [\Delta t]_c + [\Delta t]_d$ is the sum of the time intervals of the continuous and dispersed phase. Eulerian averaging is often most common, where dependent variables change with respect to the independent variables of time and space. The integral operator smooths spatially local or temporally instantaneous variations within the domain of integration.

The phasic average $\overline{\overline{(\cdot)}}$ is defined as

$$\overline{\overline{F_q}} := \frac{\overline{F_q}}{\alpha_q}, \tag{4.2}$$

which represents the average in time of the phase.

Quantities such as volume, momentum and energy can be written in terms of the variable per unit mass ψ, i.e. $F_q = \rho_q \psi_q$, where ρ_q is the local instant fluid density for the phase q, and ψ_q is some quantity per unit mass. Thus, the mean value of ψ_q should be weighted by its density:

$$\widehat{F_q} := \frac{\overline{\rho_q F_q}}{\bar{\rho}_q}, \tag{4.3}$$

which is the mass weighted mean value $\widehat{(\cdot)}$.

4.1.2 Mass Conservation

The general expression for the conservation of mass for a phase q is given as follows from applying the averaging techniques from 4.1.1 applied to the functions associated with a particular phase [39]:

$$\frac{\partial \left(\alpha_q \overline{\overline{\rho_q}}\right)}{\partial t} + \nabla \cdot \left(\alpha_q \overline{\overline{\rho_q}} \widehat{\mathbf{v}}_q\right) = 0, \tag{4.4}$$

where α_q is the time-averaged local phase fraction of a phase q, $\overline{\overline{\rho_q}}$ is the time-averaged phasic average density and $\widehat{\mathbf{v}}_q$ is the time-averaged mass-weighted mean phase velocity. We have also assumed there is no interphase mass transfer. In the case where each phase is incompressible, the mean density $\overline{\overline{\rho_q}}$ is constant and we obtain from (4.4),

$$\frac{\partial \left(\alpha_q\right)}{\partial t} + \nabla \cdot \left(\alpha_q \widehat{\mathbf{v}}_q\right) = 0, \tag{4.5}$$

which is the mass conservation equation used in Chapter 5 for the $H(\mathrm{div})$-conforming method.

4.1.3 Momentum Equation

The general expression for the conservation of momentum for a phase q is given as follows [39]:

$$\begin{aligned} \frac{\partial \left(\alpha_q \overline{\overline{\rho_q}} \widehat{\mathbf{v}}_q\right)}{\partial t} + \nabla \cdot \left(\alpha_q \overline{\overline{\rho_q}} \widehat{\mathbf{v}}_q \otimes \widehat{\mathbf{v}}_q\right) = -\nabla \left(\alpha_q \overline{\overline{P_q}}\right) + \nabla \cdot \left(\alpha_q \overline{\overline{\underline{\boldsymbol{\tau}_q}}}\right) + \alpha_q \overline{\overline{\rho_q}} \widehat{\mathbf{g}}_q + \mathbf{M}_q \\ + \overline{\overline{P_{q,i}}} \nabla \alpha_q - \nabla \alpha_q \cdot \overline{\overline{\underline{\boldsymbol{\tau}_{q,i}}}}, \end{aligned} \tag{4.6}$$

where $\overline{\overline{P_q}}$ is the time-averaged phasic pressure, $\overline{\overline{\underline{\boldsymbol{\tau}_q}}}$ is the time averaged viscous stress tensor, $\widehat{\mathbf{g}}_q$ is the time-averaged mass weighted mean phase gravitational acceleration, $\overline{\overline{P_{q,i}}}\nabla\alpha_q$ and $\nabla\alpha_q \cdot \overline{\overline{\underline{\boldsymbol{\tau}_{q,i}}}}$ are the interfacial stresses, and $\mathbf{M}_q$ is the interphase momentum source term.

This work focuses on dispersed flow, where the interfacial shear stress term is assumed to be negligible. Additionally, the interfacial pressure and shear stress of the continuous and dispered phases can be assume to be equal, i.e., $\overline{\overline{P_{c,i}}} \approx \overline{\overline{P_{d,i}}} = \overline{\overline{P_{int}}}$ and $\overline{\overline{\underline{\boldsymbol{\tau}_{c,i}}}} \approx \overline{\overline{\underline{\boldsymbol{\tau}_{d,i}}}}$ [39]. We can also approximate the pressure of the dispersed phase by the interfacial pressure, $\overline{\overline{P_d}} \approx \overline{\overline{P_{int}}}$ [39]. The conservation of momentum for a dispersed flow system becomes:

$$\begin{aligned}\frac{\partial\left(\alpha_c\overline{\overline{\rho_c}}\widehat{\mathbf{v}}_c\right)}{\partial t} + \nabla\cdot\left(\alpha_c\overline{\overline{\rho_c}}\widehat{\mathbf{v}}_c\otimes\widehat{\mathbf{v}}_c\right) = &-\alpha_c\nabla\overline{\overline{P_c}} + \nabla\cdot\left(\alpha_c\overline{\overline{\underline{\boldsymbol{\tau}_c}}}\right) + \alpha_c\overline{\overline{\rho_c}}\widehat{\mathbf{g}} + \mathbf{M}_c \\ &+ \left(\overline{\overline{P_{\text{int}}}} - \overline{\overline{P_c}}\right)\nabla\alpha_c,\end{aligned} \tag{4.7a}$$

$$\frac{\partial\left(\alpha_d\overline{\overline{\rho_d}}\widehat{\mathbf{v}}_d\right)}{\partial t} + \nabla\cdot\left(\alpha_d\overline{\overline{\rho_d}}\widehat{\mathbf{v}}_d\otimes\widehat{\mathbf{v}}_d\right) = -\alpha_d\nabla\overline{\overline{P_{int}}} + \nabla\cdot\left(\alpha_d\overline{\overline{\underline{\boldsymbol{\tau}_d}}}\right) + \alpha_d\overline{\overline{\rho_d}}\widehat{\mathbf{g}} + \mathbf{M}_d. \tag{4.7b}$$

We omit the time-averaged, time-averaged phasic average and the time-average mass-weighted notation for all subsequent equations.

4.1.4 Interphase Momentum Transfer

The interphase momentum transfer terms $\mathbf{M_q}$, is the sum of multiple contributions of momentum transfer such as drag, lift, virtual mass, wall lubrication force, interfacial pressure, *etc.* For the continuous phase we have

$$\mathbf{M}_c = \mathbf{M}_{c,drag}. \tag{4.8}$$

Between the continuous and dispersed phases, the momentum exchange should sum to zero, thus

$$\mathbf{M}_c = -\mathbf{M}_d. \tag{4.9}$$

The drag force term acts in the opposite direction of the relative motion of the of the bubbles and is the sum of the form and skin drag forces on the fluid. This is caused from pressure imbalances and shear forces at the interface [39]. For spherical bubbles, The interphase momentum transfer of phase c becomes [39]:

$$\mathbf{M}_{c,drag} = \frac{3}{4}\rho_c\alpha_d\frac{C_D}{d_d}\left\|\mathbf{v}_r\right\|\mathbf{v}_r, \tag{4.10}$$

where d_d is the bubble diameter, C_D the drag coefficient, and $\mathbf{v}_r$ is the relative velocity between the dispersed and continuous phases, i.e, $\mathbf{v}_r = \mathbf{v}_d - \mathbf{v}_c$. The drag coefficient C_D can be approximated using the Schiller-Naumann drag expression [67]:

$$C_D = \max\left(\frac{24}{Re}(1 + 0.15Re^{0.687}), 0.44\right), \tag{4.11}$$

where $Re = \rho_l \|\mathbf{v}_r\| d_b / \mu_l$ is the Reynolds number. Using equation (4.9), the drag force contribution of the dispersed phase is

$$\mathbf{M}_{d,drag} = -\frac{3}{4}\rho_c \alpha_d \frac{C_D}{d_d} \|\mathbf{v}_r\| \, \mathbf{v}_r. \tag{4.12}$$

Lift is a result of shear forces and asymmetric pressure distribution around the dispersed phase, which is perpendicular to the direction of flow [19]. The expression for the momentum transfer to the continuous fluid c due to lift is

$$\mathbf{M}_{c,lift} = C_L \rho_c \alpha_d \mathbf{v}_r \times (\nabla \times \mathbf{v}_c), \tag{4.13}$$

where C_L is the lift coefficient. Drag and lift forces are depicted in Figure 4.2.

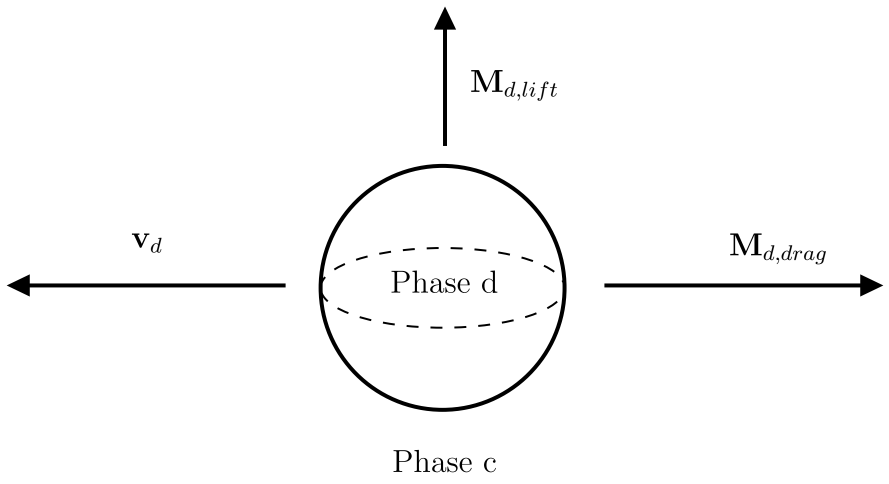

Figure 4.2: The drag $\mathbf{M}_{d,drag}$ and lift $\mathbf{M}_{d,lift}$ forces for a spherical bubble of phase d moving with velocity $\mathbf{v}_d$ in phase c.

4.1.5 Phase Fraction Boundedness

We will consider a volume fraction for each phase, also known as a phase fraction, denoted by α_q, where the subscript will denote each phase. While in continuous-continuous phase interactions are restricted a phase fraction of either 0 or 1, except in the interface region, in the case of continuous-dispersed phase interactions, the phase fraction requires the inequality constraint

$$0 \leq \alpha_q \leq 1, \tag{4.14}$$

We also have an equality constraint: the sum of the phase fractions must equal one at every point in the domain Ω:

$$\sum_{q=1} \alpha_q = 1. \tag{4.15}$$

It is crucial to enforce the boundedness of the phase fraction to preserve the physical fidelity of the model. Since the phase fraction of the continuous phase is $\alpha_c = 1 - \alpha_d$, only one of the conservation of mass equation needs to be solved (typically the disperse phase), which satisfies (4.15). For (4.14), boundedness is not inherently satisfied. One approach to maintaining phase fraction boundedness consists of thresholding [51, 36]. Depending on the formulation of the momentum equation used, thresholding can be necessary to avoid issues with division by zero and increase stability, where values below the threshold are set to a relatively small value, with the disadvantage that it can alter the profile of the phase fraction [73].

4.2 Hyperbolicity and Well-posedness

The well-posedness of the model can affect the stability and accuracy of the solution. For a model to be well-posed, three criteria must be met[50]:

- A solution exists.
- The solution is unique.
- The solution depends continuously on boundary data and parameters.

Let us consider hyperbolic partial differential equations. The most common hyperbolic general model is as follows: for n spatial dimensions $(x^i, 1 \leq i \leq n)$ we have

$$\mathbf{U}_t + \underline{\mathbf{J}}^i \mathbf{U}_i = \mathbf{S}(\mathbf{U}), \tag{4.16}$$

for time $t \geq 0$ with initial data

$$\mathbf{U}(t=0, x) = \mathbf{f}(x), \tag{4.17}$$

where $\mathbf{U}$, $\mathbf{f}$ are real vectors with m components, $\mathbf{U}_i = \partial \mathbf{U}/\partial x^i$, and $\underline{\mathbf{J}^i}$ is an $m \times m$ matrix. Examples include equations for shallow water, Burgers' equation, and the Euler's equations of gas dynamics. If the characteristic matrix of (4.16) has all real eigenvalues, the system is said to be *weakly hyperbolic*. If the eigenvalues are all real and distinct, then the system has *strong hyperbolicity* [18]. Strong hyperbolicity is equivalent to well-posedness [43, 71].

The single-pressure two-fluid model lacks hyperbolicity, and hence the equations of the model are ill-posed. Shown in [20], the system exhibits complex eigenvalues. In the following section, we will show that the eigenvalues of a simplified two-fluid model are only real when the dispersed and continuous phase velocities are equal.

4.2.1 Ill-posed Two-fluid Model Eigenvalue Example

Consider the following two-fluid model, neglecting viscous stress terms:

$$\frac{\partial \left(\rho_{\mathrm{c}} \alpha_{\mathrm{c}}\right)}{\partial t} + \frac{\partial \left(\rho_c \mathbf{u}_c \alpha_{\mathrm{c}}\right)}{\partial x} = 0, \tag{4.18a}$$

$$\frac{\partial \left(\rho_{\mathrm{d}} \alpha_{\mathrm{d}}\right)}{\partial t} + \frac{\partial \left(\rho_d \mathbf{u}_d \alpha_{\mathrm{d}}\right)}{\partial x} = 0, \tag{4.18b}$$

$$\frac{\partial \left(\rho_c \mathbf{u}_c \alpha_{\mathrm{c}}\right)}{\partial t} + \frac{\partial \left(\rho_c \mathbf{u}_c \mathbf{u}_c \alpha_{\mathrm{c}}\right)}{\partial x} = -\alpha_c \frac{\partial P_c}{\partial x} + \rho_c \alpha_{\mathrm{c}} \mathbf{g} + \mathbf{M}_c, \tag{4.18c}$$

$$\frac{\partial \left(\rho_d \mathbf{u}_d \alpha_{\mathrm{d}}\right)}{\partial t} + \frac{\partial \left(\rho_d \mathbf{u}_d \mathbf{u}_d \alpha_{\mathrm{d}}\right)}{\partial x} = -\alpha_d \frac{\partial P_c}{\partial x} + \rho_d \alpha_{\mathrm{d}} \mathbf{g} + \mathbf{M}_d. \tag{4.18d}$$

For this model, the governing equations cannot be written in fully conservative form and instead can be written as quasi-linear in terms of primitive variables $\mathbf{W} = (\alpha_{\mathrm{c}}, \mathbf{u}_c, \mathbf{u}_d, P)^{\mathrm{T}}$ such that

$$\mathbf{A}(\mathbf{W}) \frac{\partial \mathbf{W}}{\partial t} + \mathbf{B}(\mathbf{W}) \frac{\partial \mathbf{W}}{\partial x} + \mathbf{S}(\mathbf{W}) = \mathbf{0}, \tag{4.19}$$

with matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & \alpha_c\rho_c & 0 & 0 \\ 0 & 0 & \alpha_d\rho_d & 0 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \mathbf{u}_c & \alpha_\mathrm{c} & 0 & 0 \\ -\mathbf{u}_d & 0 & \alpha_\mathrm{d} & 0 \\ 0 & \alpha_c\rho_c\mathbf{u}_c & 0 & \alpha_c \\ 0 & 0 & \alpha_d\rho_d\mathbf{u}_d & \alpha_d \end{pmatrix},$$
$$\mathbf{S} = \begin{pmatrix} 0 \\ 0 \\ -\rho_c\alpha_\mathrm{c}\mathbf{g} - \mathbf{M}_c \\ -\rho_d\alpha_\mathrm{d}\mathbf{g} - \mathbf{M}_d \end{pmatrix}. \tag{4.20}$$

Here matrices **A** and **B** have been simplified by divided by their respective phase fractions, as well as the substitution $\frac{\partial\alpha_c}{\partial t} = -\frac{\partial\alpha_\mathrm{d}}{\partial t}$. The eigenvalues of this system are found from the generalized eigenvalue problem

$$\det(\mathbf{B} - \lambda\mathbf{A}) = 0. \tag{4.21}$$

The number of eigenvalues is lower than the dimension of the matrix, due to **A** being rank deficient (rank(**A**) = 3). The characteristic polynomial of the generalized eigenvalue problem is given by:

$$\rho_c\alpha_\mathrm{d}\left(\mathbf{u}_c - \lambda\right)^2 + \rho_d\alpha_\mathrm{c}\left(\mathbf{u}_d - \lambda\right)^2 = 0, \tag{4.22}$$

with two roots given by

$$\lambda_{1,2} = \frac{\alpha_d\rho_c u_c + \alpha_c\rho_d u_d \pm \sqrt{-\alpha_c\alpha_d\rho_d\rho_d(u_c - u_d)^2}}{\alpha_d\rho_c + \alpha_c\rho_d}. \tag{4.23}$$

$$\lambda_{1,2} = \frac{(\rho u)^* \pm \xi}{\rho^*}, \tag{4.24}$$

simplified with the use of an averaging operator $(.)^*$:

$$(.)^* := \frac{(.)_c}{\alpha_\mathrm{c}} + \frac{(.)_d}{\alpha_\mathrm{d}}, \tag{4.25}$$

and

$$\xi = \sqrt{-\frac{\rho_d\rho_c}{\alpha_\mathrm{d}\alpha_\mathrm{c}}\left(\mathbf{u}_d - \mathbf{u}_c\right)^2}. \tag{4.26}$$

The two eigenvalues $\lambda_{1,2}$ are real provided that $\mathbf{u}_c = \mathbf{u}_d$. If the relative velocity is non-zero, the eigenvalues are complex-valued. So for values when the continuous velocity is not equal to the dispersed velocity, the model is ill-posed. To find the third and fourth eigenvalues requires the inverse eigenvalue problem, $\mathbf{A}\mathbf{v} = \mu\mathbf{B}\mathbf{v}$, with determinant equation

$$\det(\mu\mathbf{B} - \mathbf{A}) = 0, \tag{4.27}$$

and characteristic polynomial

$$\mu^2 \left(\rho_c\alpha_{\mathrm{d}} \left(\mu\mathbf{u}_c - 1\right)^2 + \rho_d\alpha_{\mathrm{c}} \left(\mu\mathbf{u}_d - 1\right)^2\right) = 0. \tag{4.28}$$

Since $\mu_{1,2} = 1/\lambda_{1,2}$, then $\mu_{3,4} = 0$ implies $\lambda_{3,4}$ are infinite.

An ill-posed two-fluid model can contain nonphysical instabilities as well as excessive numerical diffusion. Structured numerical methods are then required to preserve the integrity of the solution. Cockburn and Shu [14] demonstrated the effectiveness of the discontinuous Galerkin method on hyperbolic problems, thus possibly being a favorable scheme to be applied to the two-fluid model.

Chapter 5

H(div)-conforming Finite Element Method for Two-Fluid Incompressible Flow Systems

From section 4, the governing equations for the two-fluid model are as follows:

$$\frac{\partial\left(\alpha_q\right)}{\partial t}+\nabla\cdot\left(\alpha_q\mathbf{v}_q\right)=0, \tag{5.1a}$$

$$\begin{aligned}\frac{\partial\left(\alpha_c\rho_c\mathbf{v}_c\right)}{\partial t}+\nabla\cdot\left(\alpha_c\rho_c\mathbf{v}_c\otimes\mathbf{v}_c\right)=&-\alpha_c\nabla P_c+\nabla\cdot\left(\alpha_c\underline{\boldsymbol{\tau}_c}\right)+\alpha_c\rho_c\mathbf{g}+\mathbf{M}_c\\&+\left(P_{\text{int}}-P_c\right)\nabla\alpha_c,\end{aligned} \tag{5.1b}$$

$$\frac{\partial\left(\alpha_d\rho_d\mathbf{v}_d\right)}{\partial t}+\nabla\cdot\left(\alpha_d\rho_d\mathbf{v}_d\otimes\mathbf{v}_d\right)=-\alpha_d\nabla P_{int}+\nabla\cdot\left(\alpha_d\underline{\boldsymbol{\tau}_d}\right)+\alpha_d\rho_d\mathbf{g}+\mathbf{M}_d, \tag{5.1c}$$

once again, noting that the averaged notation has been omitted. In this work, only the interphase momentum transfer due to drag is considered. For simplifying this work, we will assume that the bulk and interfacial pressures are equal, i.e., $P_c = P_d = P_{int}$. Omitting this assumption can improve the approximation's numerical accuracy and increase the time interval where the dispersed phase fraction is well-posed [72]. The fluid-fluid system of interest is a gas-liquid system where liquid, l, is the continuous phase and gas, g is the dispersed phase, which will be replaced in the subscripts of the equations from now on.

Substituting equations (4.10) and (4.12) into (5.1) yields:

$$\begin{aligned}\frac{\partial\left(\alpha_g\rho_g\mathbf{v}_g\right)}{\partial t}+\nabla\cdot\left(\alpha_g\rho_g\mathbf{v}_g\otimes\mathbf{v}_g\right)=&-\alpha_g\nabla P+\nabla\cdot\left(2\mu_g\alpha_g\epsilon(\mathbf{v}_g)\right)+\alpha_g\rho_g\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left\|\mathbf{v}_r\right\|\mathbf{v}_r,\end{aligned}\tag{5.2a}$$

$$\begin{aligned}\frac{\partial\left(\alpha_l\rho_l\mathbf{v}_l\right)}{\partial t}+\nabla\cdot\left(\alpha_l\rho_l\mathbf{v}_l\otimes\mathbf{v}_l\right)=&-\alpha_l\nabla P+\nabla\cdot\left(2\mu_l\alpha_l\epsilon(\mathbf{v}_l)\right)+\alpha_l\rho_l\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left\|\mathbf{v}_r\right\|\mathbf{v}_r.\end{aligned}\tag{5.2b}$$

Where $\epsilon(\cdot)=\frac{1}{2}(\nabla(\cdot)+\nabla(\cdot)^T)$ is the symmetric gradient and $\rho_l,\rho_g,\mu_l,\mu_g$ are assumed constant. The pressure quantity has dropped its subscript as it is a *shared* quantity between the two momentum equations. For compactness, $\epsilon(\cdot)$ is retained in the subsequent equations. Using the fact that $\frac{\partial\alpha_l}{\partial t}=-\frac{\partial\alpha_g}{\partial t}$ and that both phase densities are constant in equations (5.1a), the liquid and gas phase mass conservation equations can be combined:

$$\nabla\cdot\left(\alpha_l\mathbf{v}_l+\alpha_g\mathbf{v}_g\right)=0.\tag{5.3}$$

This equation is analagous to the incompressibility constraint in section 3.1.

5.1 Mixture Velocity

The mixture velocity is the sum of the continuous and dispersed phase weighted velocities, namely $\mathbf{V}=\alpha_l\mathbf{v}_l+\alpha_g\mathbf{v}_g$. Since $\mathbf{v}_l=(\mathbf{V}-\alpha_g\mathbf{v}_g)/\alpha_l$, equations (5.2b), (5.2a) and (5.3) can be rewritten as the following system of partial differential equations:

$$\begin{aligned}&\frac{\partial\left(\alpha_g\rho_g\mathbf{v}_g\right)}{\partial t}+\nabla\cdot\left(\alpha_g\rho_g\mathbf{v}_g\otimes\mathbf{v}_g\right)=-\alpha_g\nabla P+\nabla\cdot\left(2\mu_g\alpha_g\epsilon(\mathbf{v_g})\right)+\alpha_g\rho_g\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right),\end{aligned}\tag{5.4a}$$

$$\begin{aligned}&\frac{\partial\left(\rho_l\mathbf{V}-\alpha_g\rho_l\mathbf{v_g}\right)}{\partial t}+\nabla\cdot\left(\frac{\rho_l}{\alpha_l}(\mathbf{V}-\alpha_g\mathbf{v_g})(\mathbf{V}-\alpha_g\mathbf{v_g})\right)\\&=-\alpha_l\nabla P+\nabla\cdot\left(2\mu_l\alpha_l\left(\epsilon\left(\frac{\mathbf{V}}{\alpha_l}\right)-\epsilon\left(\frac{\alpha_g\mathbf{v}_g}{\alpha_l}\right)\right)\right)+\alpha_l\rho_l\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right),\end{aligned}\tag{5.4b}$$

$$\nabla \cdot \mathbf{V} = 0. \tag{5.4c}$$

Equations (5.4) make up the two-fluid Eulerian-Eulerian model using the mixture velocity. By including the mixture velocity, this allows us to satisfy equation $\nabla \cdot \mathbf{V} = 0$ exactly using an $H(div)$-conforming method.

5.2 Weak Formulation for the Momentum Equations

Derivation of the momentum equations is similar to the $H(\text{div})$-conforming weak formulation of the incompressible Navier–Stokes equations (see Section 3). Our function space for the velocity will be BDM_h^k, and our pressure space is chosen as Q_h^{k-1}, see (3.5). Our weak formulation for $\nabla \cdot \mathbf{V}$ becomes

$$\sum_{K \in \mathcal{T}} \int_K q \nabla \cdot \mathbf{V} \mathrm{d}x = 0. \tag{5.5}$$

Using the LLF flux from (B.9) and the fact that $\epsilon(\mathbf{u}) : \nabla(\mathbf{v}) = \epsilon(\mathbf{u}) : \epsilon(\mathbf{v})$ (see 4.9), we can derive the momentum equation weak formulation for the gas phase:

$$
\begin{aligned}
&\sum_{K\in\mathcal{T}} \int_K \alpha_g \rho_g \mathbf{g}\cdot\mathbf{u}\mathrm{d}x \\
&\quad= \sum_{K\in\mathcal{T}} \int_K \left(\mathbf{u}\cdot\partial_t(\alpha_g\rho_g\mathbf{v}_g) - (\alpha_g\rho_g\mathbf{v}_g)\otimes\mathbf{v}_g : \nabla\mathbf{u} + 2\mu_g\alpha_g\epsilon(\mathbf{v}_g) : \epsilon(\mathbf{u}) - P\nabla\cdot(\alpha_g\mathbf{u})\right)\mathrm{d}x \\
&\quad- \sum_{K\in\mathcal{T}} \int_K \frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g - \frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g - \frac{\mathbf{V}}{\alpha_l}\right)\cdot\mathbf{u}\mathrm{d}x \\
&\quad- \sum_{K\in\mathcal{T}} \int_K \nabla\cdot(\alpha_g)\otimes\mathbf{v}_g : \epsilon(\mathbf{u})\mathrm{d}x \\
&\quad- \sum_{F\in\mathcal{F}_I} \int_F 2\mu_g \left[\!\left[\mathbf{v}_g\otimes\mathbf{n}\right]\!\right] : \{\!\{\alpha_g\epsilon(\mathbf{u})\}\!\}\mathrm{d}s - \sum_{F\in\mathcal{F}_I} \int_F 2\mu_g \left[\!\left[\mathbf{u}\otimes\mathbf{n}\right]\!\right] : \{\!\{\alpha_g\epsilon(\mathbf{v}_g)\}\!\}\mathrm{d}s \\
&\quad- \sum_{F\in\mathcal{F}_D} \int_F 2\mu_g\left(\mathbf{v}_g - \boldsymbol{\phi}_g\right)\otimes\mathbf{n} : \alpha_g\epsilon(\mathbf{u})\mathrm{d}s - \sum_{F\in\mathcal{F}_D} \int_F 2\mu_g\mathbf{u}\otimes\mathbf{n} : \alpha_g\epsilon(\mathbf{v}_g)\mathrm{d}s \\
&\quad+ \sum_{F\in\mathcal{F}_I} \int_F \frac{2\mu_g\beta}{h}\left[\!\left[\mathbf{u}\otimes\mathbf{n}\right]\!\right] : \left[\!\left[\mathbf{v}_g\otimes\mathbf{n}\right]\!\right]\mathrm{d}s + \sum_{F\in\mathcal{F}_D} \int_F \frac{2\mu_g\beta}{h}\mathbf{u}\cdot\left(\mathbf{v}_g - \boldsymbol{\phi}_g\right)\mathrm{d}s \\
&\quad+ \sum_{F\in\mathcal{F}_I} \int_F \left(\rho_g(\mathbf{u}^+ - \mathbf{u}^-)\cdot\left(\{\!\{\alpha_g\mathbf{v}_g\otimes\mathbf{v}_g\}\!\}\cdot\mathbf{n}^+ + \frac{1}{2}|\mathbf{v}_g\cdot\mathbf{n}|\left((\mathbf{v}_g)^+ - (\mathbf{v}_g)^-\right)\right)\mathrm{d}s\right. \\
&\quad+ \sum_{F\in\mathcal{F}_D} \int_F \rho_g\mathbf{u}\cdot\left(\frac{1}{2}\left(\alpha_g\mathbf{v}_g\otimes\mathbf{v}_g + \alpha_g\boldsymbol{\phi}_{\boldsymbol{g}}\otimes\boldsymbol{\phi}_{\boldsymbol{g}}\right)\cdot\mathbf{n} + \frac{1}{2}|\mathbf{v}_g\cdot\mathbf{n}|\left(\mathbf{v}_g - \boldsymbol{\phi}_{\boldsymbol{g}}\right)\right)\mathrm{d}s \\
&\quad+ \sum_{F\in\mathcal{F}_N} \int_F \mathbf{u}\cdot(\boldsymbol{\psi}_{\boldsymbol{g}} + \alpha_g\max(\mathbf{v}_g\cdot\mathbf{n}, 0)\mathbf{v}_g)\mathrm{d}s.
\end{aligned}
\tag{5.6a}
$$

Lastly, using the LLF flux for the continuous phase in (B.8), we have the mixture velocity weak formulation:

$$
\begin{aligned}
&\sum_{K\in\mathcal{T}}\int_K \alpha_l\rho_l\mathbf{g}\cdot\mathbf{w}\mathrm{d}x \\
&\quad=\sum_{K\in\mathcal{T}}\int_K \left(\mathbf{w}\cdot\partial_t(\alpha_l\rho_l\mathbf{v}_l)-(\alpha_l\rho_l\mathbf{v}_l)\otimes\mathbf{v}_l:\nabla\mathbf{w}+2\mu_l\alpha_l\epsilon(\mathbf{v}_l):\epsilon(\mathbf{w})-P\nabla\cdot(\alpha_l\mathbf{w})\right)\mathrm{d}x \\
&\quad-\sum_{K\in\mathcal{T}}\int_K \frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right)\cdot\mathbf{w}\mathrm{d}x \\
&\quad-\sum_{K\in\mathcal{T}}\int_K \nabla\cdot(\alpha_l)\otimes\mathbf{v}_l:\epsilon(\mathbf{w})\mathrm{d}x \\
&\quad-\sum_{F\in\mathcal{F}_I}\int_F 2\mu_l\left[\!\left[\mathbf{v}_l\otimes\mathbf{n}\right]\!\right]:\{\!\{\alpha_l\epsilon(\mathbf{w})\}\!\}\mathrm{d}s-\sum_{F\in\mathcal{F}_I}\int_F 2\mu_l\left[\!\left[\mathbf{w}\otimes\mathbf{n}\right]\!\right]:\{\!\{\alpha_l\epsilon(\mathbf{v}_l)\}\!\}\mathrm{d}s \\
&\quad-\sum_{F\in\mathcal{F}_D}\int_F 2\mu_l\left(\mathbf{v}_l-\boldsymbol{\phi}_l\right)\otimes\mathbf{n}:\alpha_l\epsilon(\mathbf{w})\mathrm{d}s-\sum_{F\in\mathcal{F}_D}\int_F 2\mu_l\mathbf{w}\otimes\mathbf{n}:\alpha_l\epsilon(\mathbf{v}_l)\mathrm{d}s \\
&\quad+\sum_{F\in\mathcal{F}_I}\int_F \frac{2\mu_l\beta}{h}\left[\!\left[\mathbf{w}\otimes\mathbf{n}\right]\!\right]:\left[\!\left[\mathbf{v}_l\otimes\mathbf{n}\right]\!\right]\mathrm{d}s+\sum_{F\in\mathcal{F}_D}\int_F \frac{2\mu_l\beta}{h}\mathbf{w}\cdot\left(\mathbf{v}_l-\boldsymbol{\phi}_l\right)\mathrm{d}s \\
&\quad+\sum_{F\in\mathcal{F}_I}\int_F \left(\rho_l(\mathbf{w}^+-\mathbf{w}^-)\cdot\left(\{\!\{\alpha_l\mathbf{v}_l\otimes\mathbf{v}_l\}\!\}\cdot\mathbf{n}^++\frac{1}{2}|\mathbf{v}_l\cdot\mathbf{n}|\left((\mathbf{v}_l)^+-(\mathbf{v}_l)^-\right)\right)\mathrm{d}s\right. \\
&\quad+\sum_{F\in\mathcal{F}_D}\int_F \rho_l\mathbf{w}\cdot\left(\frac{1}{2}\left(\alpha_l\mathbf{v}_l\otimes\mathbf{v}_l+\alpha_l\boldsymbol{\phi_l}\otimes\boldsymbol{\phi_l}\right)\cdot\mathbf{n}+\frac{1}{2}|\mathbf{v}_l\cdot\mathbf{n}|\left(\mathbf{v}_l-\boldsymbol{\phi_l}\right)\right)\mathrm{d}s \\
&\quad+\sum_{F\in\mathcal{F}_N}\int_F \mathbf{w}\cdot(\boldsymbol{\psi_l}+\max(\mathbf{v}_l\cdot\mathbf{n},0)\mathbf{v}_l)\mathrm{d}s,
\end{aligned}
\tag{5.6b}
$$

where $\mathbf{v}_l=(\mathbf{V}-\alpha_g\mathbf{v}_g)/\alpha_l$.

5.3 Weak formulation for the Phasic Mass Conservation Equations

The two-fluid flow momentum equations (5.4) will be coupled to the dispersed phase mass conservation equation; thus, we will discuss its discontinuous Galerkin weak form next.

The dispersed phase mass conservation equation is given by

$$\partial_t(\alpha_g) + \nabla\cdot(\alpha_g \mathbf{v}_g) = 0 \qquad \text{in } \Omega\times I, \tag{5.7a}$$

$$\alpha_g = \psi \qquad \text{on } \Gamma_D\times I, \tag{5.7b}$$

$$\mathbf{v}_g\alpha_g - \max(\mathbf{v}_g\cdot\mathbf{n},0)\alpha_g = \phi \qquad \text{on } \Gamma_N\times I, \tag{5.7c}$$

where $\alpha_g : \Omega\times I \to \mathbb{R}$ is the gas (dispersed) phase fraction and ψ and ϕ given boundary data. Consider the following finite element space for the phase fraction:

$$Z_h := \{z_h \in L^2(\mathcal{T}), z_h \in P_k(K)\forall K\in\mathcal{T}\}. \tag{5.8}$$

The DG weak formulation of (5.7) is a scalar version of the DG weak formulation for the Navier–Stokes problem (with $p_h = 0, \nu = 0$). Starting with (3.11), the DG weak formulation of (5.7) is given by

$$\begin{aligned} 0 = &\sum_{K\in\mathcal{T}}\int_K z^h\partial_t(\alpha_g^h)\mathrm{d}x \\ &-\sum_{K\in\mathcal{T}}\int_K \alpha_g^h\mathbf{v}_g^h\cdot\nabla z^h\mathrm{d}x \\ &+\sum_{F\in\mathcal{F}_I}\int_F (z^{h+}-z^{h-})\left(\mathbf{v}_g^h\cdot\mathbf{n}\{\!\{\alpha_g^h\}\!\} + \frac{1}{2}\left|\mathbf{v}_g^h\cdot\mathbf{n}\right|(\alpha_g^{h+}-\alpha_g^{h-})\right)\mathrm{d}s \\ &+\sum_{F\in\mathcal{F}_D}\int_F z^h\left(\frac{1}{2}\mathbf{v}_g^h\cdot\mathbf{n}(\alpha_g^h+\psi) + \frac{1}{2}\left|\mathbf{v}_g^h\cdot\mathbf{n}\right|(\alpha_g^h-\psi)\right)\mathrm{d}s \\ &+\sum_{F\in\mathcal{F}_N}\int_F z^h(\phi+\max(\mathbf{v}_g^h\cdot n,0)\alpha_g^h)\cdot\mathbf{n}\mathrm{d}s, \end{aligned} \tag{5.9}$$

where $a_g^h, z_h \in Z_h$. The liquid phase fraction can be solved by

$$\alpha_c^h = 1-\alpha_d^h. \tag{5.10}$$

5.4 Coupling of Two-Fluid Flow Momentum and Phase Fraction Equations: Time Stepping

In this section, we consider the time-stepping algorithm when coupling the two-fluid flow problem (5.4) to the phasic mass conservation equation (5.7). In particular, consider the problem:

$$\begin{aligned}&\frac{\partial\left(\alpha_g\rho_g\mathbf{v}_g\right)}{\partial t}+\nabla\cdot\left(\alpha_g\rho_g\mathbf{v}_g\otimes\mathbf{v}_g\right)=-\alpha_l\nabla P+\nabla\cdot\left(2\mu_g\alpha_g\epsilon(\mathbf{v_g})\right)+\alpha_g\rho_g\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right),\end{aligned}\tag{5.11a}$$

$$\begin{aligned}&\frac{\partial\left(\rho_l\mathbf{V}-\alpha_g\rho_l\mathbf{v_g}\right)}{\partial t}+\nabla\cdot\left(\frac{\rho_l}{\alpha_l}(\mathbf{V}-\alpha_d\mathbf{v_g})(\mathbf{V}-\alpha_d\mathbf{v_g})\right)\\=&-\alpha_l\nabla P+\nabla\cdot\left(2\mu_l\alpha_l\left(\epsilon\left(\frac{\mathbf{V}}{\alpha_l}\right)-\epsilon\left(\frac{\alpha_g\mathbf{v}_g}{\alpha_l}\right)\right)\right)+\alpha_l\rho_l\mathbf{g}\\&+\frac{3}{4}\alpha_g\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right|\left(\left(1+\frac{\alpha_g}{\alpha_l}\right)\mathbf{v}_g-\frac{\mathbf{V}}{\alpha_l}\right),\end{aligned}\tag{5.11b}$$

$$\nabla\cdot(\mathbf{V})=0,\tag{5.11c}$$

$$\partial_t\left(\alpha_g\right)+\nabla\cdot\left(\alpha_g\mathbf{v}_g\right)=0,\tag{5.11d}$$

$$\alpha_l=1-\alpha_g.\tag{5.11e}$$

Following [24], we consider the following time stepping approach. Let $\mathbf{V}^{n+1}=\alpha_g^n\mathbf{v}_g^{n+1}+\alpha_l^n\mathbf{v}_l^{n+1}$, $\mathbf{V}^n=\alpha_g^n\mathbf{v}_g^n+\alpha_l^n\mathbf{v}_l^n$, $\mathbf{v}_r^{n+1}=\mathbf{v}_l^{n+1}-\mathbf{v}_g^{n+1}$ and $\mathbf{v}_r^n=\mathbf{v}_l^n-\mathbf{v}_g^n$, then the time discrete formulation becomes:

$$\begin{aligned}&\frac{\alpha_g^n\rho_g\mathbf{v}_g^{n+1}-\alpha_g^n\rho_g\mathbf{v}_g^n}{\Delta t}+\nabla\cdot\left(\alpha_g^n\rho_g\mathbf{v}_g^{n+1}\otimes\mathbf{v}_g^n\right)=-\alpha_l^n\nabla P^{n+1}\\&+\nabla\cdot\left(2\mu_g\alpha_g^n\epsilon({\mathbf{v_g}}^{n+1})\right)+\alpha_g^n\rho_g\mathbf{g}\\&+\frac{3}{4}\alpha_g^n\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g^n}{\alpha_l^n}\right)\mathbf{v}_g^n-\frac{\mathbf{V}^n}{\alpha_l^n}\right|\left(\left(1+\frac{\alpha_g^n}{\alpha_l^n}\right)\mathbf{v}_g^{n+1}-\frac{\mathbf{V}^{n+1}}{\alpha_l^n}\right),\end{aligned}\tag{5.12a}$$

$$\begin{aligned}&\frac{(\rho_l\mathbf{V}^{n+1}-\alpha_g^n\rho_l{\mathbf{v_g}}^{n+1})-(\rho_l\mathbf{V}^n-\alpha_g^n\rho_l{\mathbf{v_g}}^n)}{\Delta t}\\&+\nabla\cdot\left(\frac{\rho_l}{\alpha_l^n}(\mathbf{V}^{n+1}-\alpha_d^n{\mathbf{v_g}}^{n+1})(\mathbf{V}^n-\alpha_d^n{\mathbf{v_g}}^n)\right)\\=&-\alpha_l^n\nabla P^{n+1}+\nabla\cdot\left(2\mu_l\alpha_l^n\left(\epsilon\left(\frac{\mathbf{V}^{n+1}}{\alpha_l^n}\right)-\epsilon\left(\frac{\alpha_g^n\mathbf{v}_g^{n+1}}{\alpha_l^n}\right)\right)\right)+\alpha_l^n\rho_l\mathbf{g}\\&+\frac{3}{4}\alpha_g^n\rho_l\frac{C_D}{d_b}\left|\left(1+\frac{\alpha_g^n}{\alpha_l^n}\right)\mathbf{v}_g^n-\frac{\mathbf{V}^n}{\alpha_l^n}\right|\left(\left(1+\frac{\alpha_g^n}{\alpha_l^n}\right)\mathbf{v}_g^{n+1}-\frac{\mathbf{V}^{n+1}}{\alpha_l^n}\right),\end{aligned}\tag{5.12b}$$

$$\nabla\cdot(\mathbf{V}^{n+1})=0,\tag{5.12c}$$

$$\frac{\alpha_g^{n+1} - \alpha_g^n}{\Delta t} + \nabla \cdot \left(\alpha_g^{n+1} \mathbf{v}_g^n\right) = 0, \tag{5.12d}$$

$$\alpha_l^{n+1} = 1 - \alpha_g^{n+1}. \tag{5.12e}$$

Note that by having linearized the non-linear term in (5.12b), (5.12a) and (5.12d) we have uncoupled the problem: we first compute α_g^{n+1} and α_l^{n+1} from (5.12d) and (5.12e), after which we compute $\mathbf{v_g}^{n+1}, \mathbf{v_l}^{n+1}$ and P^{n+1} by solving (5.12b)-(5.12c). To fully discretize (5.12), we use the weak formulations given by (5.6a), (5.6b) and (5.9).

5.5 Simulation Conditions

As a simple first case example to demonstrate the validity of the code, we will consider simple channel flow through a two-dimensional pipe. The simulation domain is a two-dimensional channel with both the gas and liquid phase injected from the left. Momentum transfer due to lift and virtual mass is generally used as fine-tuning parameters and are thus neglected in this work. As a computational domain we take $\Omega = [0, 2] \times [0, 0.41]$ and mesh size $N = 2048$ triangles. We compute the solution over the time interval $I = [0, 2]$ for which we set the time step to $\Delta t = 0.01$. The computational domain can be seen in Figure 5.1. No-slip boundaries are prescribed on the top and bottom wall, and on the right edge, we impose a Neumann BC with $\boldsymbol{\psi_g}, \boldsymbol{\psi_l} = 0$. We consider a cubic polynomial approximation for the velocities and phase fractions and a quadratic polynomial approximation for the pressure.

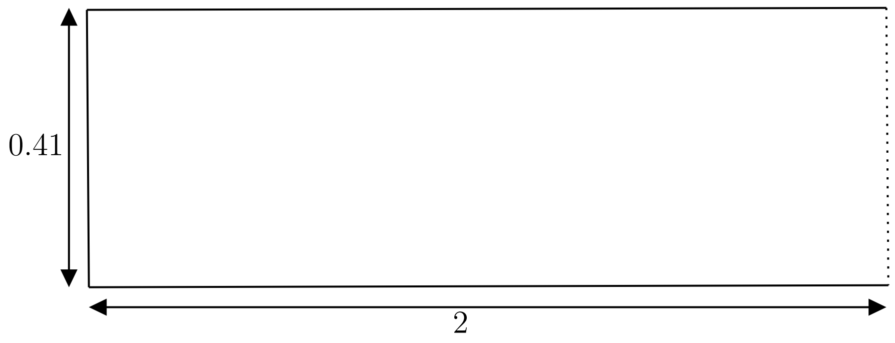

Figure 5.1: Simulation domain for two-phase flow. A dashed line indicates a Neumann boundary.

In this simulation, both the continuous and dispersed phase velocity are equal, as are the fundamental properties of each fluid, to demonstrate the basic validity of the numerical scheme. For this test case the gravitational force is $\mathbf{g} = (0, 0)$.

Property	Value
Gas density (kg/m^3)	10
Liquid density (kg/m^3)	1000
Gas viscosity (Pa s)	2×10^{-5}
Liquid viscosity (Pas)	5×10^{-3}
Bubble diameter (m)	10^{-3}
Drag constant	$\max\left[\frac{24}{Re}\left(1+0.15Re^{0.687}\right), 0.44\right], Re = \frac{\rho_l \lVert \mathbf{v}_r \rVert d_b}{\mu_l}$

Table 5.1: Physical properties.

Initial Condition
$\alpha_g(\mathbf{x}, 0) = 0.025$
$\mathbf{v}_g(\mathbf{x}, 0) = \mathbf{v}_l(\mathbf{x}, 0) = 1.5(4y)(0.41 - y)/(0.41)^2$
$P(\mathbf{x}, 0) = 0$

Table 5.2: Initial and boundary conditions.

For this numerical test the number of elements $N = 2048$, with $\Delta t = 0.01$. All other physical properties of the system are given in Table 5.1, with initial and boundary conditions given in Table 5.2. The NGSolve code is provided in Appendix A.3. As shown in Figure 5.2, although the horizontal and vertical velocity of the gas phase is what we would expect, we start to see small instabilities in the gas phase fraction along the walls. This effect increases at time $t = 1$ in Figure 5.3. The divergence constraint, $\nabla \cdot \mathbf{V} = 0$, was approximated with machine precision for this numerical test, i.e., $\nabla \cdot \mathbf{V} \approx 10^{-11}$.

We can conclude that while this method is stable and accurate for rudimentary test cases, further work needs to be done to validate this method. In general, for cases when $\mathbf{v}_l$ is not equal to $\mathbf{v}_g$, or when the inlet velocity is much faster, the method is unstable, which could be caused by limiting the dispersed phase via thresholding [74].

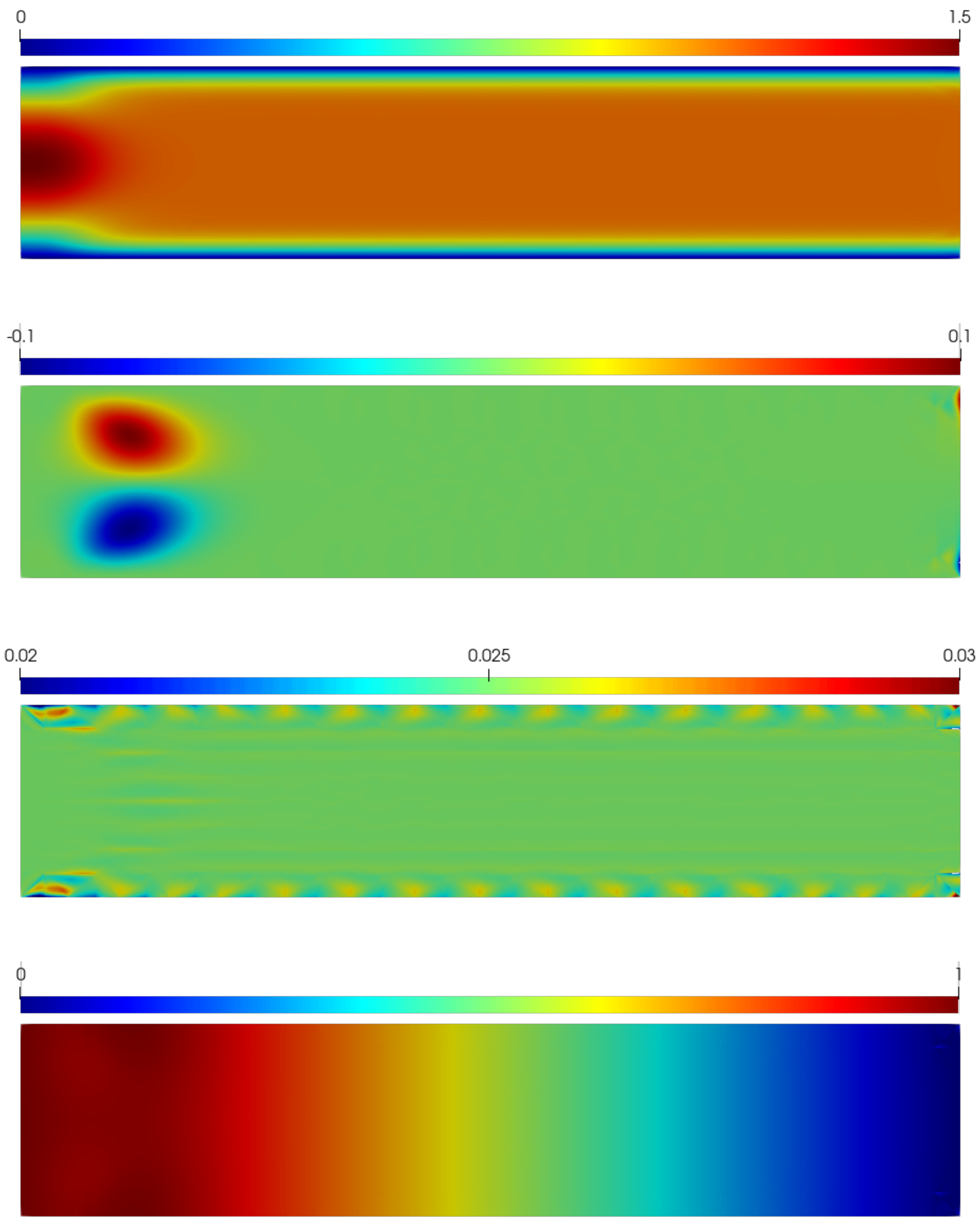

Figure 5.2: In order: the horizontal gas phase velocity, vertical gas phase velocity, gas-phase fraction, and pressure at t = 0.1.

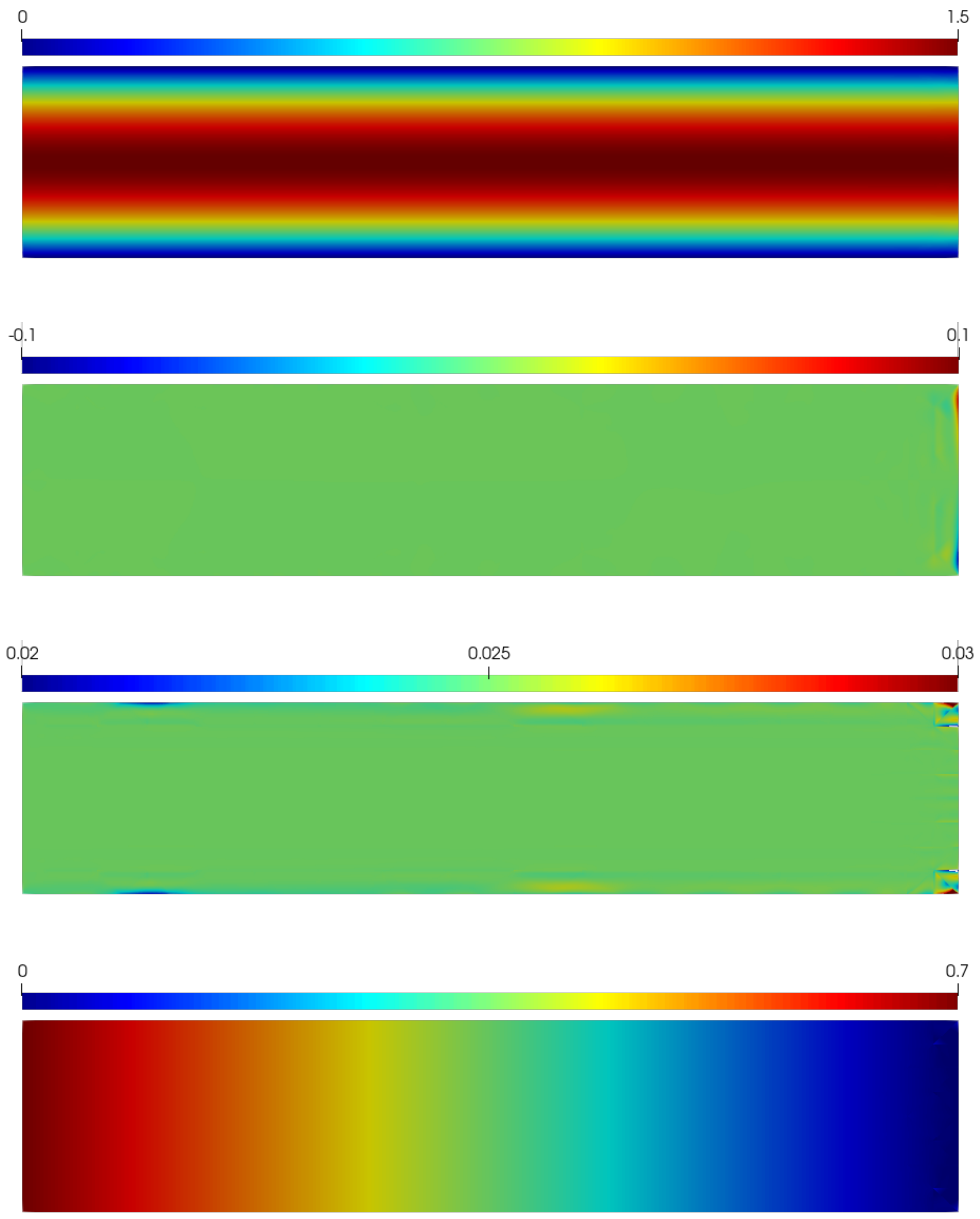

Figure 5.3: In order: the horizontal gas phase velocity, vertical gas phase velocity, gas phase fraction and pressure at t = 1.

Chapter 6

Conclusions and Recommendations for Future Work

6.1 Conclusions

An $H(\mathrm{div})$-conforming discontinuous Galerkin method for the incompressible Navier–Stokes equations is introduced. Simulations of an $H(\mathrm{div})$-conforming method for the incompressible Navier–Stokes equations to display the pressure-robustness of the model and stability with high degree approximation polynomials for the velocity and pressure. An $H(\mathrm{div})$-conforming discontinuous Galerkin method for two-phase bubbly flow using mixture velocity for the two-fluid Euler-Euler model has been developed. Simulations for two-phase channel flow are presented to verify the $H(\mathrm{div})$-conforming method.

6.2 Recommendations for Future Work

The recommendations for future work focuses on improvements to the $H(\mathrm{div})$-conforming method physical fidelity of the model, which include:

- One of the drawbacks of the discontinuous Galerkin method is that it introduces considerably more unknowns than standard continuous Galerkin or finite volume methods. Discretizations using discontinuous Galerkin finite element methods are less sparse and introduce many couplings between unknowns. One approach to resolve this problem is using hybrid discontinuous Galerkin finite element methods, which results in a smaller linear system resulting from the discretization.

- In this work, only the momentum exchange due to the drag force is considered. The addition of momentum exchange terms can affect the numerical simulation and impact the physical fidelity of the solution.

- Panicker and Passalacqua [54] proved the inclusion of a dispersion term dependent on the coefficient and the gradient of the gas volume fraction can ensure the hyperbolicity of the equations preventing non-physical instabilities.

- Shown in [73], a diffuse-interface method can impose a solid-fluid boundary with the structure of the interface impacting the boundary conditions, which could be accommodated for discontinuous Galerkin methods.

References

[1] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, 2002.

[2] A. Baggag, H. Atkins, and D. Keyes. Parallel implementation of the discontinuous Galerkin method. 1999.

[3] K.-J. Bathe. The inf–sup condition and its evaluation for mixed finite element methods. *Computers & structures*, 79(2):243–252, 2001.

[4] M. Bernacki, L. Fezoui, S. Lanteri, and S. Piperno. Parallel discontinuous Galerkin unstructured mesh solvers for the calculation of three-dimensional wave propagation problems. *Applied mathematical modelling*, 30(8):744–763, 2006.

[5] C. E. Brennen. *Fundamentals of multiphase flow*. Cambridge University Press, 2005.

[6] F. Brezzi, J. Douglas, and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numerische Mathematik*, 47(2):217–235, 1985.

[7] F. Brezzi and M. Fortin. Mixed and hybrid finite element methods. *Springer Science & Business Media*, 2012.

[8] M. A. Case, V. J. Ervin, A. Linke, and L. G. Rebholz. A connection between Scott–Vogelius and grad-div stabilized Taylor–Hood fe approximations of the Navier–Stokes equations. *SIAM Journal on Numerical Analysis*, 49(4):1461–1481, 2011.

[9] N. Chalmers, G. Agbaglah, M. Chrust, and C. Mavriplis. A parallel hp-adaptive high order discontinuous Galerkin method for the incompressible Navier–Stokes equations. *Journal of Computational Physics: X*, 2:100023, 2019.

[10] L. Chen and L. Schaefer. Godunov-type upwind flux schemes of the two-dimensional finite volume discrete Boltzmann method. *Computers & Mathematics with Applications*, 75(9):3105–3126, 2018.

[11] B. Cockburn. Discontinuous Galerkin methods for convection-dominated problems. *High-Order Methods for Computational Physics*, 9:69–224, 1999.

[12] B. Cockburn, G. Kanschat, and D. Schötzau. A locally conservative ldg method for the incompressible Navier–Stokes equations. *Mathematics of Computation*, 74(251):1067–1095, 2005.

[13] B. Cockburn, G. Kanschat, and D. Schötzau. A note on discontinuous Galerkin divergence-free solutions of the Navier–Stokes equations. *Journal of Scientific Computing*, 31(1-2):61–73, 2007.

[14] B. Cockburn and C.-W. Shu. Tvb Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. ii. general framework. *Mathematics of Computation*, 52(186):411–435, 1989.

[15] B. Cockburn and W. Zhang. An a posteriori error estimate for the variable-degree Raviart–Thomas method. *Mathematics of Computation*, 83(287):1063–1082, 2014.

[16] D. D. Siqueira, P. R. B. Devloo, and S. M. Gomes. A new procedure for the construction of hierarchical high order hdiv and hcurl finite element spaces. *Journal of Computational and Applied Mathematics*, 240:204–214, 2013.

[17] H. D. Sterck, T. A. Manteuffel, S. F. McCormick, and L. Olson. Numerical conservation properties of h (div)-conforming least-squares finite element methods for the Burgers equation. *SIAM Journal on Scientific Computing*, 26(5):1573–1597, 2005.

[18] T. N. Dinh, R. R. Nourgaliev, and T. G. Theofanous. Understanding the ill-posed two-fluid model. *Proceedings of the 10th International Topical Meeting on Nuclear Reactor Thermal-Hydraulics (NURETH'03)*, 2003.

[19] D. A. Drew and R. T. Lahey Jr. The virtual mass and lift force on a sphere in rotating and straining inviscid flow. *International Journal of Multiphase Flow*, 13(1):113–121, 1987.

[20] D. A. Drew and S. L. Passman. Theory of multicomponent fluids. *Applied Mathematical Sciences*, 1998.

[21] D. A. Drew and S. L. Passman. Theory of multicomponent fluids, volume 135. *Springer Science & Business Media*, 2006.

[22] M. Dumbser and M. Käser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—ii. the three-dimensional isotropic case. *Geophysical Journal International*, 167(1):319–336, 2006.

[23] T. Ellis, L. Demkowicz, and J. Chan. Locally conservative discontinuous Petrov–Galerkin finite elements for fluid problems. *Computers & Mathematics with Applications*, 68(11):1530–1549, 2014.

[24] H. Elman, M. Mihajlović, and D. Silvester. Fast iterative solvers for buoyancy driven flow problems. *Journal of Computational Physics*, 230(10):3900–3914, 2011.

[25] H. Enwald, E. Peirano, and A.-E. Almstedt. Eulerian two-phase flow theory applied to fluidization. *International Journal of Multiphase Flow*, 22:21–66, 1996.

[26] Xi. Feng, M. Neilan, and S.Schnake. Interior penalty discontinuous Galerkin methods for second order linear non-divergence form elliptic pdes. *Journal of Scientific Computing*, 74(3):1651–1676, 2018.

[27] M. Giordano and V. Magi. Petrov–Galerkin finite element stabilization for two-phase flows. *International Journal for Numerical Methods in Fluids*, 51(9-10):1117–1129, 2006.

[28] F. X. Giraldo, J. S. Hesthaven, and T. Warburton. Nodal high-order discontinuous Galerkin methods for the spherical shallow water equations. *Journal of Computational Physics*, 181(2):499–525, 2002.

[29] V. Girault and P.-A. Raviart. Finite element methods for Navier–Stokes equations: theory and algorithms, volume 5. *Springer Science & Business Media*, 2012.

[30] A. Gupta and S. Roy. Euler–Euler simulation of bubbly flow in a rectangular bubble column: Experimental validation with radioactive particle tracking. *Chemical Engineering Journal*, 225:818–836, 2013.

[31] J. Guzmán and M. Neilan. Conforming and divergence-free stokes elements in three dimensions. *IMA Journal of Numerical Analysis*, 34(4):1489–1508, 2014.

[32] J. Guzmán, C.-W. Shu, and F. A. Sequeira. H (div) conforming and dg methods for incompressible Euler's equations. *IMA Journal of Numerical Analysis*, 37(4):1733–1771, 2017.

[33] T. C. Haley, D. A. Drew, and R. T. Lahey Jr. An analysis of the eigenvalues of bubbly two-phase flows. *Chemical Engineering Communications*, 106(1):93–117, 1991.

[34] Y. Han and Y. Hou. Semirobust analysis of an h (div)-conforming dg method with semi-implicit time-marching for the evolutionary incompressible Navier–stokes equations. *IMA Journal of Numerical Analysis*, 2021.

[35] R. Hartmann and P. Houston. Symmetric interior penalty dg methods for the compressible Navier–Stokes equations ii: Goal–oriented a posteriori error estimation. *International Journal of Numerical Analysis and Modeling*, 3(2):141–162, 2006.

[36] S. Hill, D. Deising, T. Acher, H. Klein, D. Bothe, and H. Marschall. Boundedness-preserving implicit correction of mesh-induced errors for vof based heat and mass transfer. *Journal of Computational Physics*, 352:285–300, 2018.

[37] T. L. Horváth and S. Rhebergen. An exactly mass conserving space-time embedded-hybridized discontinuous Galerkin method for the Navier–Stokes equations on moving domains. *Journal of Computational Physics*, 417:109577, 2020.

[38] P. Houston, D. Schötzau, and T. P. Wihler. An hp-adaptive mixed discontinuous Galerkin fem for nearly incompressible linear elasticity. *Computer Methods in Applied Mechanics and Engineering*, 195(25-28):3224–3246, 2006.

[39] M. Ishii and T. Hibiki. Thermo-fluid dynamics of two-phase flow. *Springer Science & Business Media*, 2010.

[40] H. A. Jakobsen, H. Lindborg, and C. A. Dorao. Modeling of bubble column reactors: progress and limitations. *Industrial & Engineering Chemistry Research*, 44(14):5107–5151, 2005.

[41] V. John, A. Linke, C. Merdon, M. Neilan, and L. G. Rebholz. On the divergence constraint in mixed finite element methods for incompressible flows. *SIAM Review*, 59(3):492–544, 2017.

[42] D. Kim and J. H. Kwon. A high-order accurate hybrid scheme using a central flux scheme and a weno scheme for compressible flowfield analysis. *Journal of Computational Physics*, 210(2):554–583, 2005.

[43] H.-O. Kreiss and J. Lorenz. Initial-boundary value problems and the Navier–Stokes equations. *SIAM*, 2004.

[44] R. T. Lahey Jr, L. Y. Cheng, D. A. Drew, and J. E. Flaherty. The effect of virtual mass on the numerical stability of accelerating two-phase flows. *International Journal of Multiphase Flow*, 6(4):281–294, 1980.

[45] S.-J. Lee, K.-S. Chang, and S.-J. Kim. Surface tension effect in the two-fluids equation system. *International Journal of Heat and Mass Transfer*, 41(18):2821–2826, 1998.

[46] C. Lehrenfeld and J. Schöberl. High order exactly divergence-free hybrid discontinuous Galerkin methods for unsteady incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 307:339–361, 2016.

[47] J. Li, D. Zhang, X. Meng, B. Wu, and Q. Zhang. Discontinuous Galerkin methods for nonlinear scalar conservation laws: Generalized local Lax–Friedrichs numerical fluxes. *SIAM Journal on Numerical Analysis*, 58(1):1–20, 2020.

[48] A. Linke and C. Merdon. On velocity errors due to irrotational forces in the Navier–Stokes momentum balance. *Journal of Computational Physics*, 313:654–661, 2016.

[49] J.-G. Liu and C.-W. Shu. A high-order discontinuous Galerkin method for 2d incompressible flows. *Journal of Computational Physics*, 160(2):577–596, 2000.

[50] M. Louaked, L. Hanich, and C. P. Thompson. Well-posedness of incompressible models of two-and three-phase flow. *IMA Journal of Applied Mathematics*, 68(6):595–620, 2003.

[51] A. J. Mwambela and G. A. Johansen. Multiphase flow component volume fraction measurement: experimental evaluation of entropic thresholding methods using an electrical capacitance tomography system. *Measurement Science and Technology*, 12(8):1092, 2001.

[52] C. R. Nastase and D. J. Mavriplis. High-order discontinuous Galerkin methods using an hp-multigrid approach. *Journal of Computational Physics*, 213(1):330–357, 2006.

[53] Y. Pan, M. P. Dudukovic, and M. Chang. Dynamic simulation of bubbly flow in bubble columns. *Chemical Engineering Science*, 54(13-14):2481–2489, 1999.

[54] N. Panicker, A. Passalacqua, and R. O. Fox. On the hyperbolicity of the two-fluid model for gas–liquid bubbly flows. *Applied Mathematical Modelling*, 57:432–447, 2018.

[55] L. Pesch. Discontinuous Galerkin finite element methods for the Navier–Stokes equations in entropy variable formulation, volume 68. 2007.

[56] J. Proft and B. Riviere. Discontinuous Galerkin methods for convection-diffusion equations for varying and vanishing diffusivity. *International Journal of Numerical Analysis & Modeling*, 6(4), 2009.

[57] J. Qiu, B. C. Khoo, and C.-W. Shu. A numerical study for the performance of the Runge–Kutta discontinuous Galerkin method based on different numerical fluxes. *Journal of Computational Physics*, 212(2):540–565, 2006.

[58] M. Rafique, P. Chen, and M. P. Duduković. Computational modeling of gas-liquid flow in bubble columns. *Reviews in Chemical Engineering*, 20(3-4):225–375, 2004.

[59] B. N. Rajani, A. Kandasamy, and S. Majumdar. Numerical simulation of laminar flow past a circular cylinder. *Applied Mathematical Modelling*, 33(3):1228–1247, 2009.

[60] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. *Technical Report, Los Alamos Scientific Lab., N. Mex.(USA)*, 1973.

[61] S. Rhebergen and G. N. Wells. A hybridizable discontinuous Galerkin method for the Navier–Stokes equations with pointwise divergence-free velocity field. *Journal of Scientific Computing*, 76(3):1484–1501, 2018.

[62] W. J. Rider and R. B. Lowrie. The use of classical Lax–Friedrichs Riemann solvers with discontinuous Galerkin methods. *International Journal for Numerical Methods in Fluids*, 40(3-4):479–486, 2002.

[63] J. E. Roberts and J.-M. Thomas. Mixed and hybrid methods. *Handbook of Numerical Analysis* , 2:523–639, 1991.

[64] R. R. Baier and I. Lunati. Mixed finite element – discontinuous finite volume element discretization of a general class of multicontinuum models. *Journal of Computational Physics*, 322, 07 2016.

[65] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. *Flow simulation with high-performance computers II.* Springer, 547–566, 1996.

[66] J. Schöberl. C++11 implementation of finite elements in ngsolve. *Technical Report 30, Institute for Analysis and Scientific Computing, Vienna University of Technology*, 2014.

[67] L. Schiller. A drag coefficient correlation. *Zeit. Ver. Deutsch. Ing.*, 77:318–320, 1933.

[68] B. Seny, J. Lambrechts, T. Toulorge, V. Legat, and J.-F. Remacle. An efficient parallel implementation of explicit multirate Runge–Kutta schemes for discontinuous Galerkin computations. *Journal of Computational Physics*, 256:135–160, 2014.

[69] J. C. Simo and F. Armero. Unconditional stability and long-term behavior of transient algorithms for the incompressible Navier–Stokes and Euler equations. *Computer Methods in Applied Mechanics and Engineering*, 111(1-2):111–154, 1994.

[70] J. H. Song and M. Ishii. The well-posedness of incompressible one-dimensional two-fluid model. *International Journal of Heat and Mass Transfer*, 43(12):2221–2231, 2000.

[71] J. M. Stewart. The cauchy problem and the initial boundary value problem in numerical relativity. *Classical and Quantum Gravity*, 15(9):2865, 1998.

[72] J. H. Stuhmiller. The influence of interfacial pressure forces on the character of two-phase flow model equations. *International Journal of Multiphase Flow*, 3(6):551–560, 1977.

[73] T. Treeratanaphitak. *Diffuse solid-fluid interface method for dispersed multiphase flows*. PhD thesis, University of Waterloo, 2018.

[74] T. Treeratanaphitak and N. M. Abukhdeir. Phase-bounded finite element method for two-fluid incompressible flow systems. *International Journal of Multiphase Flow*, 117:1–13, 2019.

[75] A. Vaidheeswaran. *Well-posedness and convergence of cfd two-fluid model for bubbly flows*. PhD thesis, Purdue University, 2015.

[76] A. Vaidheeswaran and M. L. de Bertodano. Stability and convergence of computational eulerian two-fluid model for a bubble plume. *Chemical Engineering Science*, 160:210–226, 2017.

[77] A. Vaidheeswaran, W. D. Fullmer, and M. L. de Bertodano. Effect of collision force on well-posedness and stability of the two-fluid model for vertical bubbly flows. *Nuclear Science and Engineering*, 184(3):353–362, 2016.

[78] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon. Validating the flash code: vortex-dominated flows. *High Energy Density Laboratory Astrophysics*, pages 341–346. Springer, 2005.

APPENDICES

Appendix A

NGSolve Code

A.1 H(div)-conforming Discontinuous Galerkin Method for Navier-Stokes: Unit Square

```python
"""

This code tests the validity of a Discontinuous Galerkin BDM elements
Incompressible Navier-Stokes solution by solving the following equation:

u_t + div(u x u) - nu*div(grad(u)) + grad(p) = f
div(u) = 0

u = g                                               on Dirichlet Boundaries
(u x u - nu*grad(u) + p*I)*n - max(u*n,0)u = h      on Neumann Boundaries

This file was created by Kyle Booker and James Lowman under the supervision of
Sander Rhebergen and Nasser Abukhdeir at the University of Waterloo in 2019.

This program is designed and operated using the NGSolve Finite Element Package.
www.ngsolve.org

"""

from ngsolve import *
from netgen.geom2d import unit_square
from netgen.geom2d import SplineGeometry
from math import pi
import time
start_time = time.time()

#-------------------------------------------------------------------------------#
#   User Settings:
#-------------------------------------------------------------------------------#

Verbose_Mode        = 1     #   0/1 == Yes/No -- Outputs solution information to terminal
Polynomial_Order    = 3     #   Int            -- Order of approximation polynomials
Initial_Mesh_Size   = 1   #   Float         -- Initial mesh Size
No_Refinments       = 6     #   Int            -- Number of times to refine the mesh
Time_Step           = 1e-10  #   Float          -- Size of the time step to take
No_Time_Solutions   = 1    #   Int           -- Number of transient solutions
nu                  = 1   #   Float         -- Kinematic viscosity

#-------------------------------------------------------------------------------#
#   Mesh Generation:
#-------------------------------------------------------------------------------#
```

```python


mesh = Mesh(unit_square.GenerateMesh(maxh=Initial_Mesh_Size))

if Verbose_Mode == 1:
    print ("\n\t Boundary Labels: ", mesh.GetBoundaries(),"\n") # Check boundary labels

#------------------------------------------------------------------------------#
#   Define Function Spaces:
#------------------------------------------------------------------------------#

"""

A BDM finite element space (HDiv) is defined on the mesh for the velocity while
an L2 space is defined for the pressure.

BDM elements have that property that u * n is continuous across
elements (i.e. u_1 * n_1 = u_2 * n_1 on element boundaries).

"""

# Velocity Space - HDiv BDM space
V    =  HDiv(mesh, order = Polynomial_Order, dgjumps = True, dirichlet="bottom|left")
# Pressure Space - one polynomial degree less than V
Q    =  L2(mesh, order = Polynomial_Order-1, dgjumps = True)
# Mixed Finite Element space
X    =  FESpace ([V, Q], dgjumps = True) # Mixed finite element space (u,p)

#------------------------------------------------------------------------------#
#   Define trial and test functions, and solution storage sunctions:
#------------------------------------------------------------------------------#

(u, p), (v, q) = X.TnT()    #   Define Trial functions (u,p) and Test functions (v,q)

# NGSolve utilizes grid functions as mutable scalar/vector/tensor variables
UN   =  GridFunction(X)      #   Grid Function for the solution space
UOld =  GridFunction(X)      #   Grid Function for the solution space at previous time step

# Temporary storage variables for previous time step data
U0  =   CoefficientFunction (UOld.components[0])     #   Previous velocity
P0  =   CoefficientFunction (UOld.components[1])     #   Previous pressure

#------------------------------------------------------------------------------#
#   Special variable definitions:
#------------------------------------------------------------------------------#

# Definition of the outward facing normal for every facet in the domain
n   =   specialcf.normal(mesh.dim) # Normal vector on an interface

# Definition of the individual cell sizes
h   =   specialcf.mesh_size

# A Nitsche penalty parameter is defined in the weak foruмation for all facets
alpha   =   10.0*Polynomial_Order**2/h

#------------------------------------------------------------------------------#
#   Helper Functions:
#------------------------------------------------------------------------------#

# NGSolve has no native "Max" function, therefore Max is defined explicitly
def Max(A,B):
    return IfPos(A-B,A,B) # If A-B>0 return A; else return B

# A custom function to calculate the L2 Norm error for a given solution
def CalcL2Error(sol):
    err_u = sqrt(Integrate((sol.components[0]-u_exact)**2, mesh))
    p_mean = sqrt(Integrate(sol.components[1]**2, mesh))
    p_exact_mean = sqrt(Integrate(p_exact**2, mesh))
    err_p = sqrt(Integrate((sol.components[1]-p_mean-p_exact+p_exact_mean)**2, mesh))
    #err_p = sqrt(Integrate((sol.components[1]-p_exact)**2, mesh))
    err_div = sqrt(Integrate(Trace(Grad(sol.components[0]))**2, mesh))
    return (err_u, err_p, err_div)


#------------------------------------------------------------------------------#
#   NGSolve-Mutable Variables:
#------------------------------------------------------------------------------#
```

```

# A special coefficient function class, Parameter, is required to update time.
#       This is required as the exact solution is dependent on time, and as such
#       requires the Dirichlet boundary conditions to be dependent on time
var_time    =   Parameter(0.0)

#-------------------------------------------------------------------------------#
#   Exact Solution:
#-------------------------------------------------------------------------------#

u_x     =   CoefficientFunction(sin(pi*x-var_time)*sin(pi*y-var_time))
u_y     =   CoefficientFunction(cos(pi*x-var_time)*cos(pi*y-var_time))
u_exact =   CoefficientFunction((u_x, u_y))

p_exact = sin(pi*x)*cos(pi*y)

#-------------------------------------------------------------------------------#
#   Generation of forcing function, f, that enables exact solution:
#-------------------------------------------------------------------------------#

# Helper variables, vector calculus
grad_u          =   CoefficientFunction((   u_x.Diff(x),    u_x.Diff(y),    \
                                            u_y.Diff(x),    u_y.Diff(y)),   \
                                        dims=(2,2))
outerUU         =   OuterProduct(u_exact,u_exact)
div_outerUU     =   CoefficientFunction((outerUU[0,0].Diff(x) + outerUU[0,1].Diff(y),outerUU[1,0].Diff(x) + outerUU
     [1,1].Diff(y)))

# Forcing function for the Stokes initial condition solution
f_xstokes       =   -nu * (u_exact[0].Diff(x).Diff(x) + u_exact[0].Diff(y).Diff(y))+ p_exact.Diff(x)
f_ystokes       =   -nu * (u_exact[1].Diff(x).Diff(x) + u_exact[1].Diff(y).Diff(y))+ p_exact.Diff(y)
f_st            =   CoefficientFunction((f_xstokes,f_ystokes))

# Multiplying pressure by the identity tensor
p_I             =   CoefficientFunction((   p_exact,    0,              \
                                            0,          p_exact),       \
                                        dims=(2,2))

# In the weak formulation of Stokes, the Neumann condition requires
# -nu grad(u) + pI projected onto the outward facing normal on the boundary.
# Generated here as h for Stokes.
h_stokes        =   (- nu * grad_u + p_I) * n

# Forcing function for Stokes
f_x             =   u_exact[0].Diff(var_time) - nu * (u_exact[0].Diff(x).Diff(x) \
                  + u_exact[0].Diff(y).Diff(y)) + div_outerUU[0] + p_exact.Diff(x)
f_y             =   u_exact[1].Diff(var_time) - nu * (u_exact[1].Diff(x).Diff(x) \
                  + u_exact[1].Diff(y).Diff(y)) + div_outerUU[1] + p_exact.Diff(y)
force_navier_stokes   = CoefficientFunction((f_x,f_y))

# Neumann boundary condition for INS
h_ins   =  (outerUU - nu*grad_u + p_I)*n - Max(u_exact*n, 0)*u_exact

#-------------------------------------------------------------------------------#
#   Setting up the time stepping variables:
#-------------------------------------------------------------------------------#

dt      = Time_Step                     #   Time step
t       = 0.0                           #   Initial time
t_final = No_Time_Solutions*Time_Step   #   Final time

#-------------------------------------------------------------------------------#
#   Mutable helper functions:
#-------------------------------------------------------------------------------#

avg_u       = 0.5*(u + u.Other())               #   Average of Velocity     {{u}}
jump_u      = u-u.Other()                       #   Jump of Velocity        [[u]]
jump_v      = v-v.Other()                       #   Jump of Basis Functions [[v]]
avggrad_u   = 0.5*(Grad(u) + Grad(u.Other()))   #   Average of Vel Grad     {{Grad(u)}}
avggrad_v   = 0.5*(Grad(v) + Grad(v.Other()))   #   Average of BFs          {{Grad(v)}}

u_time_bl   = u*v/dt                            #   Blinear du/dt           (U^N+1)*v/dt
u_time_l    = U0*v/dt                           #   Linear du/dt            (U^N)*v/dt

#-------------------------------------------------------------------------------#
#   Setup the steady-state Stokes problem, to find initial condition:
#-------------------------------------------------------------------------------#
```

```
"""
A solution to the steady-state Stokes problem is utilized as the initial
condition for the Incompressible Navier-Stokes.

-nu*div(grad(u)) + grad(p) = f_stokes
"""

# Bilinear form for Stokes
bl_st   =   BilinearForm(X)

bl_st  +=   nu * InnerProduct(Grad(u), Grad(v))                     * dx \
        +   nu * alpha * InnerProduct(jump_u, jump_v)               * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_u, OuterProduct(jump_v, n))   * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_v, OuterProduct(jump_u, n))   * dx(skeleton=True) \
        +   nu * alpha * u * v                                      * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -   nu * InnerProduct(Grad(u), OuterProduct(v, n))          * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u, n))          * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -   div(v)*p                                                * dx \
        -   div(u)*q                                                * dx \

# Linear form for Stokes
l_st    =   LinearForm(X)

l_st   +=   f_st * v                                                * dx \
        +   nu * alpha * u_exact * v                                * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u_exact, n))    * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -   v * (h_stokes)                                          * ds(skeleton=True, definedon=mesh.Boundaries("top|
     right"))

#------------------------------------------------------------------------------#
#   Solution function for the steady-state Stokes problem:
#------------------------------------------------------------------------------#

# Set verbose mode for the solver:
ngsglobals.msg_level=0#Verbose_Mode

# Function that handles solution of the Stokes problem
def SolveBVP_Stokes():
    var_time.Set(0.0)
    # Begin Task Manager function to handle automatic updating of mutable variables
    with TaskManager():

        # Print degrees of freedom
        if Verbose_Mode == 1:
            print("\n\t Number of Degrees of Freedom: ", X.ndof)

        # Update the solution gridfunction
        UN.Update()

        # Interpolate the exact solution onto the boundary facets
        UN.components[0].Set((u_exact*n)*n, definedon=mesh.Boundaries("bottom|left"))

        # Assemble the linear and bilinear matrices
        bl_st.Assemble()
        l_st.Assemble()

        # Create a residual vector
        res = l_st.vec.CreateVector()
        res.data = l_st.vec - bl_st.mat * UN.vec
        # Iteratively solve for UN
        UN.vec.data += bl_st.mat.Inverse(freedofs=X.FreeDofs(), inverse='umfpack') * res

        # Save the Stokes solution as initial condition for INS
        UOld.vec.data = UN.vec.data

        # Plotting
        Draw (UOld.components[0], mesh, "velocity")
        Draw (UOld.components[1], mesh, "pressure")
        Draw (Norm(UOld.components[0]), mesh, "|velocity|")

#------------------------------------------------------------------------------#
#   Solve the steady-state Stokes problem and print the error:
```

```
#-----------------------------------------------------------------------------#

# SolveBVP_Stokes()

if Verbose_Mode == 1:
    err_u, err_p, err_div = CalcL2Error(UN)
    print("\n")
    print("\t Error in Stokes Velocity: %1.2e" %err_u)
    print("\t Error in Stokes Pressure: %1.2e" %err_p)
    print("\t Error in Stokes Divergence: %1.2e" %err_div)
    print ("\n")

#-----------------------------------------------------------------------------#
#   Setup the Incompressible Navier-Stokes problem:
#-----------------------------------------------------------------------------#

# Navier-Stokes Bilinear Form
bl_ns    =   BilinearForm(X)

bl_ns   +=   u_time_bl                                         * dx \
        +    nu * InnerProduct(Grad(u), Grad(v))               * dx \
        +    nu * alpha * InnerProduct(jump_u, jump_v)         * dx(skeleton=True) \
        -    nu * InnerProduct(avggrad_u, OuterProduct(jump_v, n))   * dx(skeleton=True) \
        -    nu * InnerProduct(avggrad_v, OuterProduct(jump_u, n))   * dx(skeleton=True) \
        +    nu * alpha * u * v                                * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    nu * InnerProduct(Grad(u), OuterProduct(v, n))    * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    nu * InnerProduct(Grad(v), OuterProduct(u, n))    * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    div(v)*p                                          * dx \
        -    div(u)*q                                          * dx \
        -    InnerProduct(OuterProduct(u,U0), Grad(v))         * dx \
        +    jump_v * (U0 * n * avg_u + 0.5 * Norm(U0 * n) * jump_u) * dx(skeleton=True) \
        +    v * (0.5 * (U0 * n) * u + 0.5 * Norm(U0 * n) * u )      * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        +    v *( Max(U0*n, 0.0) * u)                          * ds(skeleton=True, definedon=mesh.Boundaries("top|
     right"))

# Navier-Stokes Linear Form
l_ns     =   LinearForm(X)

l_ns    +=   force_navier_stokes * v * dx + u_time_l           * dx \
        +    nu * alpha * u_exact * v                          * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    nu * InnerProduct(Grad(v), OuterProduct(u_exact, n))    * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    v * (0.5*U0 * n * u_exact - 0.5*Norm(U0 * n) * u_exact) * ds(skeleton=True, definedon=mesh.Boundaries("
     bottom|left")) \
        -    v * h_ins                                         * ds(skeleton=True, definedon=mesh.Boundaries("top|
     right"))

#-----------------------------------------------------------------------------#
#   Setup the Incompressible Navier-Stokes Preconditioner:
#-----------------------------------------------------------------------------#

c = Preconditioner(type="direct", bf=bl_ns, flags = {"inverse" : "umfpack" } )

#-----------------------------------------------------------------------------#
#   Solution function for the Incompressible Navier-Stokes problem:
#-----------------------------------------------------------------------------#

# Function that handles transient solution of the Navier-Stokes problem
store = []
def SolveBVP_NavierStokes():

    # Resolve Stokes with new mesh
    SolveBVP_Stokes()
    Redraw (blocking=False)

    t = 0

    # Solve transient INS
    step    =   0  # Iteration step counter
    with TaskManager():
        while t < t_final:
            step += 1
```

```python

            # Increase time by time step
            t += float(dt)

            # Set the parameter time to update mutable variables
            var_time.Set(t)

            if Verbose_Mode == 1:
                print ('\t Time step: %d \t\t Time: %1.1e' %(step,t))

            # Update the boundary condition interpoloation with respect to time
            UN.components[0].Set(u_exact, definedon=mesh.Boundaries("bottom|left"))

            # Assemble the linear and bi-linear matrices, and preconditioner
            bl_ns.Assemble()
            l_ns.Assemble()
            c.Update()

            # solve system
            BVP(bf=bl_ns,lf=l_ns,gf=UN,pre=c,maxsteps=3,prec=1e-10).Do()

            UOld.vec.data = UN.vec.data
            Redraw (blocking=False)

    err_u, err_p, err_div = CalcL2Error(UN)
    store.append ( (X.ndof, mesh.ne, err_u, err_p, err_div) )

vtk = VTKOutput(ma=mesh,coefs=[UN.components[0][0], UN.components[0][1], UN.components[0], UN.components[1]],names=["
     HorizontalVelocity", "VerticalVelocity", "VelocityMagnitude", "Pressure"],filename="square",subdivision=3)

for i in range(No_Refinments):

    # Refine the mesh
    if i != 1:
        mesh.Refine()

    # Update the mixed function space and grid functions if mesh has changed
    X.Update()
    UN.Update()
    UOld.Update()

    # Solve the Navier-Stoke system with the new mesh
    SolveBVP_NavierStokes()
    vtk.Do()


# Final print routine to output convergence rates for velocity, pressure, divergence
i = 1
print ("\n\n\n\n\n\n")
print ("-------------------------------------------------------------------")
print (" Cells ||  E_u \t   | rate ||  E_p     | rate ||  div")
print ("-------------------------------------------------------------------")
while i < len(store) :
    rate_u   = log(store[i-1][2]/store[i][2])/log(2.0000)
    rate_p   = log(store[i-1][3]/store[i][3])/log(2.0000)
    rate_div = log(store[i-1][4]/store[i][4])/log(2.0000)
    print("%6d ||  %1.1e | %1.1f  ||  %1.1e | %1.1f  ||  %1.1e" % \
          (store[i][1], store[i][2], rate_u, store[i][3], rate_p, store[i][4]))
    i =  i+1
print ("-------------------------------------------------------------------")
print ("\n\n\n")


"""
End of Unit Test.
"""

print("--- %s seconds ---\n\n" % (time.time() - start_time))
```

A.2 H(div)-conforming Discontinuous Galerkin Method for Navier-Stokes: Vortex Shedding

```python
"""

This code tests the validity of a Discontinuous Galerkin BDM elements
Incompressible Navier-Stokes solution by solving the following equation:

u_t + div(u x u) - nu*div(grad(u)) + grad(p) = f
div(u) = 0

u = g                                                on Dirichlet Boundaries
(u x u - nu*grad(u) + p*I)*n - max(u*n,0)u = h      on Neumann Boundaries

This file was created by Kyle Booker and James Lowman under the supervision of
Sander Rhebergen and Nasser Abukhdeir at the University of Waterloo in 2019.

This program is designed and operated using the NGSolve Finite Element Package.
www.ngsolve.org

"""

from ngsolve import *
from netgen.geom2d import unit_square
from netgen.geom2d import SplineGeometry
from math import pi
import time
start_time = time.time()

import matplotlib
import numpy as np
import matplotlib.pyplot as plt

#------------------------------------------------------------------------------#
#   User Settings:
#------------------------------------------------------------------------------#

Verbose_Mode        = 1     #   0/1 == Yes/No -- Outputs solution information to terminal
Polynomial_Order    = 2  #   Int            -- Order of approximation polynomials
Initial_Mesh_Size   = 1/4   #   Float          -- Initial mesh Size
No_Refinments       = 3     #   Int            -- Number of times to refine the mesh
Time_Step           = 0.001 #   Float          -- Size of the time step to take
No_Time_Solutions   = 1000   #   Int            -- Number of transient solutions
nu                  = 0.001   #   Float          -- Kinematic viscosity
t_final             = 30  #   Final time

#------------------------------------------------------------------------------#
#   Mesh Generation:
#------------------------------------------------------------------------------#

geo = SplineGeometry()
geo.AddRectangle( (0, 0), (2.2, 0.41), bcs = ("wall", "outlet", "wall", "inlet"))
geo.AddCircle ( (0.2, 0.2), r=0.05, leftdomain=0, rightdomain=1, bc="cyl")
mesh = Mesh( geo.GenerateMesh(maxh=0.1))

mesh.Curve(Polynomial_Order)
Draw(mesh)

if Verbose_Mode == 1:
    print ("\n\t Boundary Labels: ", mesh.GetBoundaries(),"\n") # Check boundary labels

#------------------------------------------------------------------------------#
#   Define Function Spaces:
#------------------------------------------------------------------------------#

"""

A BDM finite element space (HDiv) is defined on the mesh for the velocity while
an L2 space is defined for the pressure.

BDM elements have that property that u * n is continuous across
elements (i.e. u_1 * n_1 = u_2 * n_1 on element boundaries).

"""

# Velocity Space - HDiv BDM space
V    =  HDiv(mesh, order = Polynomial_Order, dgjumps = True, dirichlet="inlet|wall|cyl")
# Pressure Space - one polynomial degree less than V
Q    =  L2(mesh, order = Polynomial_Order-1, dgjumps = True)
# Mixed Finite Element space
```

```
X    =  FESpace ([V, Q], dgjumps = True) # Mixed finite element space (u,p)
print("DOF: ", X.ndof)
drag = []
lift = []

#-----------------------------------------------------------------------------#
#   Define trial and test functions, and solution storage sunctions:
#-----------------------------------------------------------------------------#

(u, p), (v, q) = X.TnT()    #   Define Trial functions (u,p) and Test functions (v,q)

# NGSolve utilizes grid functions as mutable scalar/vector/tensor variables
UN   =  GridFunction(X)      #   Grid Function for the solution space
UOld =  GridFunction(X)      #   Grid Function for the solution space at previous time step
Pressure = GridFunction(Q)

# Temporary storage variables for previous time step data
U0  =   CoefficientFunction (UOld.components[0])     #   Previous velocity
P0  =   CoefficientFunction (UOld.components[1])     #   Previous pressure

#-----------------------------------------------------------------------------#
#   Special variable definitions:
#-----------------------------------------------------------------------------#

# Definition of the outward facing normal for every facet in the domain
n   =   specialcf.normal(mesh.dim) # Normal vector on an interface

# Definition of the individual cell sizes
h   =   specialcf.mesh_size

# A Nitsche penalty parameter is defined in the weak forumation for all facets
alpha   =   10.0*Polynomial_Order**2/h

#-----------------------------------------------------------------------------#
#   Helper Functions:
#-----------------------------------------------------------------------------#

# NGSolve has no native "Max" function, therefore Max is defined explicitly
def Max(A,B):
    return IfPos(A-B,A,B) # If A-B>0 return A; else return B

# A custom function to calculate the L2 Norm error for a given solution
def CalcL2Error(sol):
    err_u = sqrt(Integrate((sol.components[0]-u_exact)**2, mesh))
    p_mean = sqrt(Integrate(sol.components[1]**2, mesh))
    p_exact_mean = sqrt(Integrate(p_exact**2, mesh))
    err_p = sqrt(Integrate((sol.components[1]-p_mean-p_exact+p_exact_mean)**2, mesh))
    #err_p = sqrt(Integrate((sol.components[1]-p_exact)**2, mesh))
    err_div = sqrt(Integrate(Trace(Grad(sol.components[0]))**2, mesh))
    return (err_u, err_p, err_div)


#-----------------------------------------------------------------------------#
#   NGSolve-Mutable Variables:
#-----------------------------------------------------------------------------#

# A special coefficient function class, Parameter, is required to update time.
#       This is required as the exact solution is dependent on time, and as such
#       requires the Dirichlet boundary conditions to be dependent on time
var_time    =   Parameter(0.0)

#-----------------------------------------------------------------------------#
#   Exact Solution:
#-----------------------------------------------------------------------------#

u_x     =   CoefficientFunction(1.5*4*y*(0.41-y)/(0.41*0.41))
u_y     =   CoefficientFunction(0.0)
u_exact =   CoefficientFunction((u_x, u_y))

p_exact = 0 #sin(pi*x)*cos(pi*y)

#-----------------------------------------------------------------------------#
#   Generation of forcing function, f, that enables exact solution:
#-----------------------------------------------------------------------------#

f_st            =   CoefficientFunction((0.0,0.0))

```

```
h_stokes        =   CoefficientFunction((0.0,0.0))


force_navier_stokes   = CoefficientFunction((0.0,0.0))

# Neumann boundary condition for INS
h_ins = CoefficientFunction((0.0,0.0))

#-------------------------------------------------------------------------------#
#   Setting up the time stepping variables:
#-------------------------------------------------------------------------------#

dt      = Time_Step                     #   Time step
t       = 0.0                           #   Initial time

#-------------------------------------------------------------------------------#
#   Mutable helper functions:
#-------------------------------------------------------------------------------#

avg_u       = 0.5*(u + u.Other())               #   Average of Velocity     {{u}}
jump_u      = u-u.Other()                       #   Jump of Velocity        [[u]]
jump_v      = v-v.Other()                       #   Jump of Basis Functions [[v]]
avggrad_u   = 0.5*(Grad(u) + Grad(u.Other()))   #   Average of Vel Grad     {{Grad(u)}}
avggrad_v   = 0.5*(Grad(v) + Grad(v.Other()))   #   Average of BFs          {{Grad(v)}}

u_time_bl   = u*v/dt                            #   Blinear du/dt           (U^N+1)*v/dt
u_time_l    = U0*v/dt                           #   Linear du/dt            (U^N)*v/dt

#-------------------------------------------------------------------------------#
#   Setup the steady-state Stokes problem, to find initial condition:
#-------------------------------------------------------------------------------#
"""
A solution to the steady-state Stokes problem is utilized as the initial
condition for the Incompressible Navier-Stokes.

-nu*div(grad(u)) + grad(p) = f_stokes
"""

# Bilinear form for Stokes
bl_st   =   BilinearForm(X)

bl_st  +=   nu * InnerProduct(Grad(u), Grad(v))                     * dx \
        +   nu * alpha * InnerProduct(jump_u, jump_v)               * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_u, OuterProduct(jump_v, n))   * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_v, OuterProduct(jump_u, n))   * dx(skeleton=True) \
        +   nu * alpha * u * v                                      * ds(skeleton=True, definedon=mesh.Boundaries("inlet
     |wall|cyl")) \
        -   nu * InnerProduct(Grad(u), OuterProduct(v, n))          * ds(skeleton=True, definedon=mesh.Boundaries("inlet
     |wall|cyl")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u, n))          * ds(skeleton=True, definedon=mesh.Boundaries("inlet
     |wall|cyl")) \
        -   div(v)*p                                                * dx \
        -   div(u)*q                                                * dx \

# Linear form for Stokes
l_st    =   LinearForm(X)

l_st   +=   f_st * v                                                * dx \
        +   nu * alpha * u_exact * v                                * ds(skeleton=True, definedon=mesh.Boundaries("inlet
     ")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u_exact, n))    * ds(skeleton=True, definedon=mesh.Boundaries("inlet
     ")) \
        -   v * (h_stokes)                                          * ds(skeleton=True, definedon=mesh.Boundaries("
     outlet"))

#-------------------------------------------------------------------------------#
#   Solution function for the steady-state Stokes problem:
#-------------------------------------------------------------------------------#

# Set verbose mode for the solver:
ngsglobals.msg_level=0#Verbose_Mode

# Function that handles solution of the Stokes problem
def SolveBVP_Stokes():
    var_time.Set(0.0)
    # Begin Task Manager function to handle automatic updating of mutable variables
    with TaskManager():

        # Print degrees of freedom
        if Verbose_Mode == 1:
            print("\n\t Number of Degrees of Freedom: ", X.ndof)

        # Update the solution gridfunction
        UN.Update()

        # Interpolate the exact solution onto the boundary facets
        UN.components[0].Set((u_exact*n)*n, definedon=mesh.Boundaries("inlet"))

        # Assemble the linear and bilinear matrices
        bl_st.Assemble()
        l_st.Assemble()

        # Create a residual vector
        res = l_st.vec.CreateVector()
        res.data = l_st.vec - bl_st.mat * UN.vec
        # Iteratively solve for UN
        UN.vec.data += bl_st.mat.Inverse(freedofs=X.FreeDofs(), inverse='umfpack') * res

        # Save the Stokes solution as initial condition for INS
        UOld.vec.data = UN.vec

        # Plotting
        Draw (UOld.components[0], mesh, "velocity")
        Draw (UOld.components[1], mesh, "pressure")
        Draw (Norm(UOld.components[0]), mesh, "|velocity|")

#-----------------------------------------------------------------------------#
#   Setup the Incompressible Navier-Stokes problem:
#-----------------------------------------------------------------------------#

# Navier-Stokes Bilinear Form
bl_ns    =   BilinearForm(X)

bl_ns   +=   u_time_bl                                            * dx \
        +   nu * InnerProduct(Grad(u), Grad(v))                   * dx \
        +   nu * alpha * InnerProduct(jump_u, jump_v)             * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_u, OuterProduct(jump_v, n))   * dx(skeleton=True) \
        -   nu * InnerProduct(avggrad_v, OuterProduct(jump_u, n))   * dx(skeleton=True) \
        +   nu * alpha * u * v                                    * ds(skeleton=True, definedon=mesh.Boundaries("inlet|wall|cyl")) \
        -   nu * InnerProduct(Grad(u), OuterProduct(v, n))        * ds(skeleton=True, definedon=mesh.Boundaries("inlet|wall|cyl")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u, n))        * ds(skeleton=True, definedon=mesh.Boundaries("inlet|wall|cyl")) \
        -   div(v)*p                                              * dx \
        -   div(u)*q                                              * dx \
        -   InnerProduct(OuterProduct(u,U0), Grad(v))             * dx \
        +   jump_v * (U0 * n * avg_u + 0.5 * Norm(U0 * n) * jump_u) * dx(skeleton=True) \
        +   v * (0.5 * (U0 * n) * u + 0.5 * Norm(U0 * n) * u )    * ds(skeleton=True, definedon=mesh.Boundaries("inlet|wall|cyl")) \
        +   v *( Max(U0*n, 0.0) * u)                              * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# Navier-Stokes Linear Form
l_ns     =   LinearForm(X)

l_ns    +=   force_navier_stokes * v * dx + u_time_l              * dx \
        +   nu * alpha * u_exact * v                              * ds(skeleton=True, definedon=mesh.Boundaries("inlet")) \
        -   nu * InnerProduct(Grad(v), OuterProduct(u_exact, n))  * ds(skeleton=True, definedon=mesh.Boundaries("inlet")) \
        -   v * (0.5*U0 * n * u_exact - 0.5*Norm(U0 * n) * u_exact) * ds(skeleton=True, definedon=mesh.Boundaries("inlet")) \
        -   v * h_ins                                             * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

#-----------------------------------------------------------------------------#
#   Setup the Incompressible Navier-Stokes Preconditioner:
#-----------------------------------------------------------------------------#

c = Preconditioner(type="direct", bf=bl_ns, flags = {"inverse" : "umfpack" })

#-----------------------------------------------------------------------------#
#   Solution function for the Incompressible Navier-Stokes problem:
```

```
#------------------------------------------------------------------------------#


f = LinearForm(X)
f.Assemble()

U_mean = 1
L = 0.41

time_vals = []
drag_x_vals = []
drag_y_vals = []

res = f.vec.CreateVector()

store = []


def SolveBVP_NavierStokes():

    # Resolve Stokes with new mesh
    SolveBVP_Stokes()
    Redraw (blocking=False)

    t = 0
    # Solve transient INS
    step    =   0  # Iteration step counter
    with TaskManager():
        while t < t_final:
            step += 1
            # Increase time by time step


            t += float(dt)

            # Set the parameter time to update mutable variables

            if Verbose_Mode == 1:
                print ('\t Time step: %d \t\t Time: %1.1e' %(step,t))

            # Assemble the linear and bi-linear matrices, and preconditioner
            bl_ns.Assemble()
            l_ns.Assemble()
            c.Update()

            # solve system
            BVP(bf=bl_ns,lf=l_ns,gf=UN,pre=c,maxsteps=300,prec=1.e-12 ).Do()

            UOld.vec.data = UN.vec
            Redraw (blocking=False)

            #print(drag)

            VH1= H1(mesh, order=Polynomial_Order, dirichlet="wall|inlet|outlet")
            et = GridFunction(VH1)
            et.Set(CoefficientFunction(1.0), definedon=mesh.Boundaries("cyl"))
            gradu00 = GridFunction(VH1)
            gradu01 = GridFunction(VH1)
            gradu10 = GridFunction(VH1)
            gradu11 = GridFunction(VH1)
            ppp = GridFunction(VH1)
            gradu00.Set(grad(UOld.components[0])[0,0])
            gradu01.Set(grad(UOld.components[0])[0,1])
            gradu10.Set(grad(UOld.components[0])[1,0])
            gradu11.Set(grad(UOld.components[0])[1,1])
            ppp.Set(UOld.components[1])
            c_drag = Integrate(et*(nu*(gradu00*n[0] + gradu01*n[1]) - ppp*n[0]), mesh, BND)
            c_drag = c_drag/(1.0*0.05)
            c_lift = Integrate(et*(nu*(gradu10*n[0] + gradu11*n[1]) - ppp*n[1]), mesh, BND)
            c_lift = c_lift/(1.0*0.05)
            print ("c_drag:", c_drag)
            print ("c_lift:", c_lift)
            drag.append(c_drag)
            lift.append(c_lift)


    # Solve the Navier-Stoke system with the new mesh
```

```python
SolveBVP_NavierStokes()

# Write drag data to file
file_drag = open("drag.dat", "w")
for i in drag:
    file_drag.write("%2.2e\n" % i)
file_drag.close()

# Write lift data to file
file_lift = open("lift.dat", "w")
for i in lift:
    file_lift.write("%2.2e\n" % i)
file_lift.close()
```

A.3 H(div)-conforming Discontinuous Galerkin Method for Two-Fluid Flow

```python
################################################################################
#The DG BDM Euler-Euler weak formulation using BDM elements.
################################################################################
from ngsolve import *
from netgen.geom2d import SplineGeometry
ngsglobals.msg_level=1 # Set to 1 for more detailed Output/debugging
import numpy as np
import time

def Max(A,B):
    return IfPos(A-B,A,B) # If A-B>0 return A; else return B

def Min(A,B):
    return IfPos(A-B,B,A) # If A-B>0 return B; else return A

def CalcL2Error(approx, exact, mesh):
    return sqrt(Integrate((approx - exact)**2, mesh))

def CalcChange(A, B, mesh):
    return sqrt(Integrate((A - B)**2, mesh))

################################################################################
# Parameters
################################################################################
dt           = 1e-2 # Time step
final_time   = 1000.0 # Final Time
param_t      = Parameter(0.0)
t_0          = 0.625   # Maximum Inlet time

mesh_size    = 2
k            = 3 # Order of approximation polynomials

################################################################################
# Constants
################################################################################
rho_c     = 1000.0  # Density of Continuous Phase [kg/m^3]
rho_d     = 10.0  # Density of Disperse Phase [kg/m^3]
mu_c      = 5e-3 # Dynamic Viscosity of Continuous [Pa s]
mu_d      = 2e-5 # Dynamic Viscosity of Disperse Phase [Pa s]

x_s       = 0.41  # Characteristic Length scale
v_s       = 1.5 # Characteristic Velocity scale
g_s       = 9.81 # Characteristic Gravity scale
h_s       = 2 # Characteristic Height scale
P_s       = rho_c * g_s * h_s # Characteristic Pressure scale
t_s       = x_s / v_s

bubble_d = 1e-3/x_s # Bubble Size diameter [m]

e         = np.finfo(float).eps # Machine epsilon

Eu_c      = P_s/(rho_c * v_s * v_s) # Continuous Phase Euler Number
Eu_d      = P_s/(rho_d * v_s * v_s) # Continuous Phase Euler Number

```

```
Re_c     = 0.001#rho_c * v_s * x_s / mu_c  # Continuous Phase Reynolds number
Re_d     = 0.001 #rho_d * v_s * x_s / mu_d  # Disperse Phase Reynolds number

Fr        = v_s / (sqrt(g_s*x_s))  # Froude number

grav     = CoefficientFunction((0.0, 0.0))

################################################################################
# Mesh Creation
################################################################################
geo = SplineGeometry()
geo.AddRectangle((0, 0), (2, 0.41), bcs = ("wall", "outlet", "wall", "inlet"))
mesh = Mesh(geo.GenerateMesh(maxh=mesh_size))
mesh.Curve(k)
mesh.Refine()
mesh.Refine()
mesh.Refine()
# mesh.Refine()


# Mesh related functions
h = specialcf.mesh_size # Mesh size
n = specialcf.normal(mesh.dim) # Outward normal vector on element facets

beta = 10.0*(k**2)/h # Penalty Parameter

################################################################################
# Function Spaces
################################################################################

V    = HDiv(mesh, order = k, dirichlet="wall|inlet")
Q    = L2(mesh, order = k-1) # Must be k-1 for stability
X    = FESpace ([V, V, Q], dgjumps = True) # Mixed finite element space
A    = L2(mesh, order = k, dgjumps = True) # Gas Phase Fraction Finite Element Space

################################################################################
# Trial and Test Functions
################################################################################

# u_m: Mixture Velocity Trial Function; v_m: Mixture Velocity Test Function
# u_d: Disperse Phase Velocity Trial Function; v_d: Disperse Phase Velocity Test Function
# p: Pressure Trial Function; q: Pressure Test Function
(u_m, u_d, p), (v_m, v_d, q) = X.TnT()

 # a_d: Disperse Phase Trial Function; z_d: Disperse Phase Test Function
a_d, z = A.TnT()

################################################################################
# Gridfunctions
################################################################################
UN      = GridFunction(X) # Gridfuction for Velocities and Pressure
UOld    = GridFunction(X) # Gridfunction for the solution at previous time step

A_D     = GridFunction(A) # Gridfunction for Disperse Phase Fraction
A_D_Old = GridFunction(A) # Gridfunction for the Disperse Phase Fraction at previous time step

# Solution at PREVIOUS time step
U_M_0   = CoefficientFunction (UOld.components[0]) # Mixture Velocity
U_D_0 = CoefficientFunction(UOld.components[1]) # Disperse Phase Velocity
P_0     = CoefficientFunction (UOld.components[2]) # Pressure
U_C_0 = (U_M_0 - A_D_Old * U_D_0)/(1.0 - A_D_Old) # Continuous Phase Velocity

################################################################################
# Boundary Conditions
################################################################################

#vel_inlet = Min(param_t/t_0, 1.0)*0.0616*exp(-((x/0.025)**2)/(2*((0.1)**2))) # Inlet Velocity: Gaussian Distribution

#ad_inlet  = Min(param_t/t_0, 1.0)*0.026*exp(-((x/0.025)**2)/(2*((0.1)**2)))  # Disperse Phase
ad_inlet = 0.025

# Phase Fraction Boundary Conditions
ad_bnd = CoefficientFunction(ad_inlet) # Dispersed Phase Dirichlet Inlet condition
ad_wall_bnd  = CoefficientFunction(0.00)  # Dispersed Phase Dirichlet Wall condition
force_ad = CoefficientFunction(0.0)

# Dispersed Phase Velocity field
```

```
vel_inlet = 1.5*4*y*(0.41-y)/(0.41*0.41)
ud_bnd = CoefficientFunction((vel_inlet, 0.0)) # Dispersed Phase Velocity at Inlet
ud_wall_bnd = CoefficientFunction((0.0, 0.0)) # Dispersed Phase Velocity at Walls

# Mixture Velocity field
um_bnd = CoefficientFunction((vel_inlet, 0.0)) # Mixture Velocity at Inlet
um_wall_bnd = CoefficientFunction((0.0, 0.0)) # Mixture Velocity at Walls

# Neumann boundary condition
neumann_bnd = CoefficientFunction((0.0, 0.0)) # Neumann boundary condition for velocity
disperse_phase_neumann_bnd = CoefficientFunction((0.0,0.0)) # Neumann boundary condition disperse phase fraction

###############################################################################
# Initial Conditions
###############################################################################
UN.components[0].Set(CoefficientFunction((0.0,0.0)))
UN.components[1].Set(CoefficientFunction((0.0,0.0)))
A_D.Set(CoefficientFunction(0.025))

###############################################################################
# Intermediary Viscous Stress Functions
###############################################################################
S_ud            = CoefficientFunction(grad(u_d))
Stress_ud       = CoefficientFunction(0.5*(S_ud + S_ud.trans))
S_ud_O          = CoefficientFunction(grad(u_d.Other()))
Stress_ud_Other = CoefficientFunction(0.5*(S_ud_O + S_ud_O.trans))

S_vd            = CoefficientFunction(grad(v_d))
Stress_vd       = CoefficientFunction(0.5*(S_vd + S_vd.trans))
S_vd_O          = CoefficientFunction(grad(v_d.Other()))
Stress_vd_Other = CoefficientFunction(0.5*(S_vd_O + S_vd_O.trans))

S_vm            = CoefficientFunction(grad(v_m))
Stress_vm       = CoefficientFunction(0.5*(S_vm + S_vm.trans))
S_vm_O          = CoefficientFunction(grad(v_m.Other()))
Stress_vm_Other = CoefficientFunction(0.5*(S_vm_O + S_vm_O.trans))

S_um            = CoefficientFunction(grad(u_m))
Stress_um       = CoefficientFunction(0.5*(S_um + S_um.trans))
S_um_O          = CoefficientFunction(grad(u_m.Other()))
Stress_um_Other = CoefficientFunction(0.5*(S_um_O + S_um_O.trans))

S_um_ac            = CoefficientFunction(((1.0-A_D_Old)*grad(u_m) + OuterProduct(u_m,grad(A_D_Old)))*(1.0/(1.0-A_D_Old)**2))
Stress_um_ac       = CoefficientFunction(0.5*(S_um_ac  + S_um_ac.trans))
S_um_ac_O          = CoefficientFunction((1.0-A_D_Old.Other())*grad(u_m.Other()) + OuterProduct(u_m.Other(),grad(A_D_Old).Other())*(1.0/(1.0-A_D_Old.Other())**2))
Stress_um_ac_Other   = CoefficientFunction(0.5*(S_um_ac_O  + S_um_ac_O.trans))

s_ud_ad_ac    = CoefficientFunction(((1.0-A_D_Old)*(A_D_Old*grad(u_d) + OuterProduct(grad(A_D_Old),u_d)) + OuterProduct(grad(A_D_Old),A_D_Old*u_d))*(1.0/(1.0-A_D_Old)**2))
Stress_ud_ad_ac   = CoefficientFunction(0.5*(s_ud_ad_ac  + s_ud_ad_ac.trans))
s_ud_ad_ac_O    = CoefficientFunction(((1.0-A_D_Old.Other())*(A_D_Old.Other()*grad(u_d.Other()) + OuterProduct(grad(A_D_Old).Other(),u_d.Other())) + OuterProduct(grad(A_D_Old).Other(),A_D_Old.Other()*u_d.Other()))*(1.0/(1.0-A_D_Old.Other())**2))
Stress_ud_ad_ac_Other   = CoefficientFunction(0.5*(s_ud_ad_ac_O  + s_ud_ad_ac_O.trans))

###############################################################################
# Jumps and Averages for DG numerical Fluxes
###############################################################################

jump_um = u_m - u_m.Other() # [[u_m]]
jump_ud = u_d - u_d.Other() # [[u_d]]
jump_vm = v_m - v_m.Other() # [[v_m]]
jump_vd = v_d - v_d.Other() # [[v_d]]
jump_ud_ad = u_d*A_D_Old - u_d.Other()*A_D_Old.Other() # [[u_d*a_d]]
jump_um_ac = u_m*(1.0/(1.0-A_D_Old)) - u_m.Other()*(1.0/(1.0-A_D_Old.Other())) # [[u_m*a_c]]
jump_ud_ad_ac = u_d*(A_D_Old/(1.0-A_D_Old)) - u_d.Other()*(A_D_Old.Other()/(1.0-A_D_Old.Other())) # [[u_d*a_d/a_c]]

avg_u_m = 0.5*(u_m + u_m.Other()) # {{u_m}}
avg_ud_ad = 0.5*(A_D_Old*u_d + A_D_Old.Other()*u_d.Other()) # {{u_d*a_d}}
avg_ud_ud_ad = 0.5*(A_D_Old*OuterProduct(U_D_O, u_d) + A_D_Old.Other()*OuterProduct(U_D_O.Other(),u_d.Other())) # {{u_d*a_d}}
avg_um_ac = 0.5*(u_m*(1.0/(1.0-A_D_Old)) + u_m.Other()*(1.0/(1.0-A_D_Old.Other()))) # {{u_m*a_c}}
avg_ud_ad_ac = 0.5 * (A_D_Old*u_d*(1.0/(1.0-A_D_Old)) + A_D_Old.Other()*u_d.Other()*(1.0/(1.0-A_D_Old.Other()))) # {{u_d*a_d/a_c}}
avg_um_um_ac = 0.5*((1.0 -A_D_Old)*OuterProduct(U_M_O, u_m) + (1.0 -A_D_Old.Other())*OuterProduct(U_M_O.Other(),u_m.
    Other())) # {{u_d*a_d}}


avg_Stress_vd   = 0.5*(Stress_vd + Stress_vd_Other)  # {{e(v_d)}}
avg_Stress_vm   = 0.5*(Stress_vm + Stress_vm_Other) # {{e(v_m)}}
avg_ad_Stress_ud = 0.5*(A_D_Old*Stress_ud + A_D_Old.Other()*Stress_ud_Other) # {{a_d* e(u_d)}}
avg_ac_Stress_um = 0.5*((1.0 - A_D_Old)*Stress_um + (1.0 - A_D_Old.Other())*Stress_um_Other) # {{a_d* e(u_d)}}

avg_Stress_um_ac = 0.5*((1.0 - A_D_Old)*Stress_um_ac + (1.0 - A_D_Old.Other())*Stress_um_ac_Other) # {{a_c* e(u_m/a_c)}}
avg_Stress_ud_ad_ac = 0.5*((1.0 - A_D_Old)*Stress_ud_ad_ac + (1.0 - A_D_Old.Other())*Stress_ud_ad_ac_Other)  # {{a_c* e(
    u_d*a_d/a_c)}}

grad_a_d           = CoefficientFunction(grad(a_d))
# epsilon_a_d       = CoefficientFunction(0.5*(grad_a_d  + grad_a_d.trans))

grad_a_c          = CoefficientFunction(-grad(a_d))
# epsilon_a_c      = CoefficientFunction(0.5*(grad_a_c  + grad_a_c.trans))

################################################################################
# Helper Functions
################################################################################

v_r_0    = (U_D_0 - U_M_0) # Relative velocity: v_d - v_c
Re_var = rho_c*Norm(v_r_0)*bubble_d/mu_d # Reynolds Variable
#C_D = Max((24/(Re_var + e))*(1.0 + 0.15*(Re_var**0.687)), CoefficientFunction(0.44)) # Drag Coefficient
C_D = 0.44

Div_a_d_v_d      = CoefficientFunction((grad(A_D_Old)*v_d + A_D_Old*div(v_d))) # Div(a_d*v_d)
Div_a_c_v_m      = CoefficientFunction((-grad(A_D_Old)*v_m + (1.0 - A_D_Old)*div(v_m))) # Div(a_c*v_m)

Div_a_d_u_d      = CoefficientFunction((grad(A_D_Old)*u_d + A_D_Old*div(u_d))) # Div(a_d*v_d)
Div_a_c_u_m      = CoefficientFunction((-grad(A_D_Old)*u_m + (1.0 - A_D_Old)*div(u_m))) # Div(a_c*v_m)


################################################################################
# Bilinear form
################################################################################
"""
dx : evaluates integral over elements
dx(skeleteon=True) : evaluates integral over interior element boundaries (facets)
ds(skeleteon=True) : evaluates integral over domain boundaries
"""

a_INS = BilinearForm(X)

################################################################################
# Bilinear Form Disperse Phase Momentum Equation
################################################################################

# Change in Momentum
a_INS +=   A_D_Old * u_d * v_d  * dx

# Pressure
a_INS += - dt * Div_a_d_v_d * p * dx

# Advection
a_INS += - dt * InnerProduct(OuterProduct(U_D_0,A_D_Old*u_d), grad(v_d)) * dx
a_INS +=   dt * jump_vd * (avg_ud_ud_ad * n + 0.5 * Norm(U_D_0 * n) * jump_ud_ad) * dx(skeleton=True)
a_INS +=   dt * v_d * (0.5 * A_D_Old * u_d * (U_D_0 * n) + 0.5 * Norm(U_D_0 * n) * A_D_Old * u_d ) * ds(skeleton=True,
    definedon=mesh.Boundaries("wall|inlet"))
a_INS +=   dt * v_d * (Max(U_D_0*n, 0.0) * A_D_Old * u_d) * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# Viscous Stress
a_INS +=   dt * (2.0 * mu_d) * InnerProduct(A_D_Old * Stress_ud, Stress_vd) * dx
a_INS += - dt * (2.0 * mu_d) * InnerProduct(avg_Stress_vd, OuterProduct(jump_ud_ad, n)) * dx(skeleton=True)
a_INS += - dt * (2.0 * mu_d) * InnerProduct(avg_ad_Stress_ud, OuterProduct(jump_vd, n)) * dx(skeleton=True)
a_INS += - dt * (2.0 * mu_d) * InnerProduct(OuterProduct(u_d*A_D_Old, n), Stress_vd) * ds(skeleton=True, definedon=mesh.
    Boundaries("wall|inlet"))
a_INS += - dt * (2.0 * mu_d) * InnerProduct(OuterProduct(v_d, n), A_D_Old * Stress_ud) * ds(skeleton=True, definedon=
    mesh.Boundaries("wall|inlet"))
a_INS +=   dt * (2.0 * mu_d) * beta * InnerProduct(jump_ud_ad, jump_vd) * dx(skeleton=True)
a_INS +=   dt * (2.0 * mu_d) * beta * u_d * A_D_Old * v_d * ds(skeleton=True, definedon=mesh.Boundaries("wall|inlet"))

# Drag Force
a_INS +=   dt * (0.75) * (rho_c) * A_D_Old * (C_D/bubble_d) * Norm(v_r_0) * u_d * v_d * dx
a_INS += - dt * (0.75) * (rho_c) * A_D_Old * (C_D/bubble_d) * Norm(v_r_0) * (u_m) * v_d * dx

a_INS += - dt * InnerProduct (OuterProduct(grad_a_d,u_d), Stress_vd) * dx #IS THIS CORRECT?

################################################################################
# Bilinear Form Continuous Phase Momentum Equation
################################################################################

a_INS +=  ((1.0 - A_D_Old ) * u_m * v_m) * dx

# Pressure
a_INS += - dt * Div_a_c_v_m * p * dx

# Advection
a_INS += - dt * InnerProduct(OuterProduct(U_M_0,(1.0 - A_D_Old)*u_m), grad(v_m)) * dx
a_INS +=   dt * jump_vm * (avg_um_um_ac * n + 0.5 * Norm(U_M_0 * n) * jump_um_ac) * dx(skeleton=True)
a_INS +=   dt * v_m * (0.5 * (1.0 -A_D_Old) * u_m * (U_M_0 * n) + 0.5 * Norm(U_M_0 * n) * (1.0 -A_D_Old) * u_m ) * ds(
     skeleton=True, definedon=mesh.Boundaries("wall|inlet"))
a_INS +=   dt * v_m * (Max(U_M_0*n, 0.0) * (1.0 - A_D_Old) * u_m) * ds(skeleton=True, definedon=mesh.Boundaries("outlet"
     ))

# Viscous Stress
a_INS +=   dt * (2.0 * mu_c) * InnerProduct((1.0 - A_D_Old) * Stress_um, Stress_vm) * dx
a_INS += - dt * (2.0 * mu_c) * InnerProduct(avg_Stress_vm, OuterProduct(jump_um_ac, n)) * dx(skeleton=True)
a_INS += - dt * (2.0 * mu_c) * InnerProduct(avg_ac_Stress_um, OuterProduct(jump_vm, n)) * dx(skeleton=True)
a_INS += - dt * (2.0 * mu_c) * InnerProduct(OuterProduct(u_m*(1.0 - A_D_Old), n), Stress_vm) * ds(skeleton=True,
     definedon=mesh.Boundaries("wall|inlet"))
a_INS += - dt * (2.0 * mu_c) * InnerProduct(OuterProduct(v_m, n), (1.0 - A_D_Old) * Stress_um) * ds(skeleton=True,
     definedon=mesh.Boundaries("wall|inlet"))
a_INS +=   dt * (2.0 * mu_c) * beta * InnerProduct(jump_um_ac, jump_vm) * dx(skeleton=True)
a_INS +=   dt * (2.0 * mu_c) * beta * u_m * (1.0 - A_D_Old) * v_m * ds(skeleton=True, definedon=mesh.Boundaries("wall|
     inlet"))

# Drag Force
a_INS += - dt * (0.75) * (rho_c) * A_D_Old * (C_D/bubble_d) * Norm(v_r_0) * u_d * v_m * dx
a_INS +=   dt * (0.75) * (rho_c) * A_D_Old * (C_D/bubble_d) * Norm(v_r_0) * (u_m) * v_m * dx

a_INS += - dt * InnerProduct (OuterProduct(grad_a_c, u_m), Stress_vm) * dx # FIX

# Mass Convservation
a_INS += - dt * q * Div_a_d_u_d  * dx
a_INS += - dt * q * Div_a_c_u_m  * dx

################################################################################
#  Linear Form Disperse Phase Momentum Equation
################################################################################

f_INS = LinearForm(X)

# Change in Momentum
f_INS +=    A_D_Old * U_D_0*v_d * dx

# Advection
f_INS +=  - dt * v_d * ( 0.5 * U_D_0 * n * (ad_bnd * ud_bnd) - 0.5 * Norm(U_D_0 * n) * (ad_bnd * ud_bnd) ) * ds(skeleton
     =True, definedon=mesh.Boundaries("inlet"))
f_INS +=  - dt * v_d * ( 0.5 * U_D_0 * n * (ad_wall_bnd * ud_wall_bnd) - 0.5 * Norm(U_D_0 * n) * (ad_wall_bnd *
     ud_wall_bnd) ) * ds(skeleton=True, definedon=mesh.Boundaries("wall"))
f_INS +=  - dt * v_d * neumann_bnd * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# Viscous Stress

f_INS +=    dt * (2.0 * mu_d) * beta * ad_bnd * ud_bnd * v_d * ds(skeleton=True, definedon=mesh.Boundaries("inlet"))
f_INS +=    dt * (2.0 * mu_d) * beta * ad_wall_bnd * ud_wall_bnd * v_d * ds(skeleton=True, definedon=mesh.Boundaries("
     wall"))

f_INS +=  - dt * (2.0 * mu_d) * InnerProduct(OuterProduct(ad_bnd * ud_bnd, n), Stress_vd) * ds(skeleton=True, definedon=
     mesh.Boundaries("inlet"))
f_INS +=  - dt * (2.0 * mu_d) * InnerProduct(OuterProduct(ad_wall_bnd * ud_wall_bnd, n), Stress_vd) * ds(skeleton=True,
     definedon=mesh.Boundaries("wall"))

# Gravity
a_INS +=   -dt * A_D_Old * grav * v_d * dx
# Gravity
a_INS +=  -dt * (1.0 - A_D_Old) * grav * v_m * dx

################################################################################
# Linear Form Continuous Phase Momentum Equation
################################################################################

```

```
# Change in Momentum
f_INS +=    (1.0 - A_D_Old) * U_M_0*v_m * dx

# Advection
f_INS +=  - dt * v_m * ( 0.5 * U_M_0 * n * ((1.0 - ad_bnd) * um_bnd) - 0.5 * Norm(U_M_0 * n) * ((1.0 - ad_bnd) * um_bnd)
     ) * ds(skeleton=True, definedon=mesh.Boundaries("inlet"))
f_INS +=  - dt * v_m * ( 0.5 * U_M_0 * n * ((1.0 - ad_wall_bnd) * um_wall_bnd) - 0.5 * Norm(U_M_0 * n) * ((1.0 -
     ad_wall_bnd) * um_wall_bnd) ) * ds(skeleton=True, definedon=mesh.Boundaries("wall"))
f_INS +=  - dt * v_m * neumann_bnd * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# Viscous Stress

f_INS +=    dt * (2.0 * mu_c) * beta * (1.0 - ad_bnd) * um_bnd * v_m * ds(skeleton=True, definedon=mesh.Boundaries("
     inlet"))
f_INS +=    dt * (2.0 * mu_c) * beta * (1.0 - ad_wall_bnd) * um_wall_bnd * v_m * ds(skeleton=True, definedon=mesh.
     Boundaries("wall"))

f_INS +=  - dt * (2.0 * mu_c) * InnerProduct(OuterProduct((1.0 - ad_bnd) * um_bnd, n), Stress_vm) * ds(skeleton=True,
     definedon=mesh.Boundaries("inlet"))
f_INS +=  - dt * (2.0 * mu_c) * InnerProduct(OuterProduct((1.0 - ad_wall_bnd) * um_wall_bnd, n), Stress_vm) * ds(
     skeleton=True, definedon=mesh.Boundaries("wall"))

###########################################################################
# DG Method for updating Disperse Phase Fraction
###########################################################################
a_alpha_d  = BilinearForm(A)

a_alpha_d += a_d*z * dx
a_alpha_d += -dt * (a_d * U_D_0*grad(z)) * dx
a_alpha_d += dt *(z-z.Other())*(U_D_0*n*0.5*(a_d + a_d.Other()) + 0.5*Norm(U_D_0*n)*(a_d - a_d.Other()))* dx(skeleton=
     True)
a_alpha_d += dt * z * ( 0.5*U_D_0*n*a_d + 0.5*Norm(U_D_0*n)*a_d)* ds(skeleton=True, definedon=mesh.Boundaries("inlet|
     wall"))
a_alpha_d += dt * z * a_d * Max(U_D_0*n, 0) * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# a_alpha_d +=   dt * D * InnerProduct(grad(a_d),  grad(z)) * dx
# a_alpha_d += - dt * D * InnerProduct(0.5*(grad(z) + grad(z).Other()), OuterProduct(a_d - a_d.Other(), n)) * dx(
     skeleton=True)
# a_alpha_d += - dt * D * InnerProduct(0.5*(grad(a_d) + grad(a_d).Other()), OuterProduct(z- z.Other(), n)) * dx(skeleton
     =True)
# a_alpha_d += - dt * D * InnerProduct(OuterProduct(a_d, n), grad(z)) * ds(skeleton=True, definedon=mesh.Boundaries("
     wall|inlet"))
# a_alpha_d += - dt * D * InnerProduct(OuterProduct(z, n), grad(a_d)) * ds(skeleton=True, definedon=mesh.Boundaries("
     wall|inlet"))
# a_alpha_d +=   dt * D * beta * InnerProduct(a_d - a_d.Other(), z - z.Other()) * dx(skeleton=True)
# a_alpha_d +=   dt * D * beta * a_d * z * ds(skeleton=True, definedon=mesh.Boundaries("wall|inlet"))


f_alpha_d = LinearForm(A)

f_alpha_d += dt*force_ad*z * dx

f_alpha_d += A_D_Old*z * dx
f_alpha_d += - dt * z * (0.5*U_D_0*n*ad_bnd - 0.5*Norm(U_D_0*n)*ad_bnd)* ds(skeleton=True, definedon=mesh.Boundaries("
     inlet|wall"))
f_alpha_d += - dt * z * disperse_phase_neumannn_bnd * n * ds(skeleton=True, definedon=mesh.Boundaries("outlet"))

# f_alpha_d +=    dt * D * beta * ad_bnd * z * ds(skeleton=True, definedon=mesh.Boundaries("inlet"))
# f_alpha_d +=    dt * D * beta * ad_wall_bnd * z * ds(skeleton=True, definedon=mesh.Boundaries("wall"))
#
# f_alpha_d +=  - dt * D * InnerProduct(OuterProduct(ad_bnd, n),  grad(z)) * ds(skeleton=True, definedon=mesh.Boundaries
     ("inlet"))
# f_alpha_d +=  - dt * D * InnerProduct(OuterProduct(ad_wall_bnd, n), grad(z)) * ds(skeleton=True, definedon=mesh.
     Boundaries("wall"))


###########################################################################
# Implicit Time-stepping
###########################################################################
vtk = VTKOutput(ma=mesh,coefs=[UN.components[0][0], UN.components[0][1], UN.components[1][0], UN.components[1][1], UN.
     components[0], UN.components[1], UN.components[2], A_D ],names=["vel_g_x", "vel_g_y", "vel_l_x", "vel_l_y", "
     vel_g_mag", "vel_l_mag" "pressure", "alpha_d"],filename="2FF",subdivision=3)

with TaskManager():

    Draw(U_C_0[0], mesh, "Continuous_Phase_Velocity_X")
    Draw(U_C_0[1], mesh, "Continuous_Phase_Velocity_Y")
    Draw(U_D_0[0], mesh, "Dispersed_Phase_Velocity_X")
    Draw(U_D_0[1], mesh, "Dispersed_Phase_Velocity_Y")
    Draw(Norm(P_0), mesh, "Pressure")
    Draw(A_D_Old, mesh, "Dispersed_Phase_Fraction")

    pre_INS = Preconditioner(type="direct", bf=a_INS, flags = {"inverse" : "umfpack" } )
    pre_INS_disperse_phase = Preconditioner(type="direct", bf=a_alpha_d, flags = {"inverse" : "umfpack" } )

    UOld.vec.data    = UN.vec # U^N = U^N+1
    A_D_Old.vec.data = A_D.vec # a_d^N = a_d^N+1

    t       = 0.0 # Initial time
    step    = 0 # Iteration step counter
    vtk.Do()
    while t <= final_time:
        step += 1
        t += float(dt)
        param_t.Set(t)

        print( 'Time step:  ', step , '  time:  ',t )

        UN.components[0].Set(um_bnd, definedon=mesh.Boundaries("inlet")) # Mixture Velocity Inlet Condition: May depend
     on time
        UN.components[1].Set(ud_bnd, definedon=mesh.Boundaries("inlet")) # Dispersed Velocity Inlet Condition: May
     depend on time

        #Solve for u_m, u_d, and p
        a_INS.Assemble() # Build mass matrix
        f_INS.Assemble() # Build vector
        pre_INS.Update() # Update preconditioner
        BVP(bf=a_INS,lf=f_INS,gf=UN,pre=pre_INS,maxsteps=5,prec=1e-30).Do() # Solve linear system
        UOld.vec.data  = UN.vec # Update Velocity solution

        #Solve for disperse phase fraction
        a_alpha_d.Assemble() # Build mass matrix
        f_alpha_d.Assemble() # Build vector
        pre_INS_disperse_phase.Update() # Update preconditioner
        BVP(bf=a_alpha_d,lf=f_alpha_d,gf=A_D,pre=pre_INS_disperse_phase,maxsteps=5,prec=1e-30).Do() # Solve linear
     system
        A_D_Old.vec.data = A_D.vec # Update vector solution

        if step % 10 == 0:
            vtk.Do()   # Output Solution as .vtk file
            Redraw()    # Revisualize Solution
        Redraw()
vtk.Do()
```

Appendix B

Supporting Information

B.1 Local Lax–Friedrichs Flux

We will use the LLF flux for the convective term of the two-fluid model, which is defined in equation (2.13). First, we will need to find the eigenvalues of the system of PDEs, just examining a simplified version of the continuous phase, we have

$$\frac{\partial(\alpha_c)}{\partial t} + \nabla \cdot (\alpha_c \mathbf{v}_c) = 0 \quad \text{in } \Omega \times I, \tag{B.1}$$

$$\frac{\partial(\alpha_c v_c)}{\partial t} + \nabla \cdot (\alpha_c \mathbf{v}_c \otimes \mathbf{v}_c) = 0 \text{ in } \Omega \times I, \tag{B.2}$$

and $v_c = (v_x, v_y)$ is the continuous phase velocity. In this case, we are interested in the problem

$$\begin{aligned} \partial_t \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) &= 0 && \text{in } \Omega \times I, \\ \mathbf{U}(\mathbf{x}, 0) &= \mathbf{U}_0(\mathbf{x}) && \text{in } \Omega, \end{aligned} \tag{B.3}$$

where

$$U = \begin{bmatrix} \alpha_c \\ \alpha_c v_x \\ \alpha_c v_y \end{bmatrix}, \quad F(U) = \begin{bmatrix} \alpha_c v_x & \alpha_c v_y \\ \alpha_c v_x v_x & \alpha_c v_y v_x \\ \alpha_c v_x v_y & \alpha_c v_y v_y \end{bmatrix}. \tag{B.4}$$

We will need to compute the eigenvalues of the Jacobian of the flux in an arbitrary direction $\mathbf{n} = (n_x, n_y)$. First, note that

$$\mathbf{F}(\mathbf{U}) \cdot \mathbf{n} = \begin{bmatrix} \alpha_c v_x n_x + \alpha_c v_y n_y \\ \alpha_c v_x v_x n_x + \alpha_c v_y v_x n_y \\ \alpha_c v_x v_y n_x + \alpha_c v_y v_y n_y \end{bmatrix}. \tag{B.5}$$

Its Jacobian is given by

$$\partial \mathbf{F}\cdot\mathbf{n}/\partial\mathbf{U} = \begin{bmatrix} 0 & n_x & n_y \\ -v_x^2 n_x - v_x v_y n_y & 2v_x n_x + v_y n_y & v_x n_y \\ -v_y^2 n_y - v_x v_y n_x & v_y n_x & 2v_y n_y + v_x n_x \end{bmatrix}, \tag{B.6}$$

which if we let $q = v_x n_x + v_y n_y$ then

$$\partial \mathbf{F}\cdot\mathbf{n}/\partial\mathbf{U} = \begin{bmatrix} 0 & n_x & n_y \\ -v_x q & q + v_x n_x & v_x n_y \\ -v_y q & v_y n_x & q + v_y n_y \end{bmatrix}. \tag{B.7}$$

The eigenvalues of this Jacobian are $\lambda = q$. Thus, our LLF flux becomes

$$\begin{aligned} &\underline{\mathbf{H}\left((\alpha_c\mathbf{v}_c)^+, (\alpha_c\mathbf{v}_c)^-, \mathbf{n}^+\right)} = \\ &\frac{1}{2}((\alpha_c\mathbf{v}_c\otimes\mathbf{v}_c)^+ + (\alpha_c\mathbf{v}_c\otimes\mathbf{v}_c)^-) - \frac{1}{2}|(\mathbf{v}_c\cdot\mathbf{n})\mathbf{n}|((\alpha_c\mathbf{v}_c)^+ - (\alpha_c\mathbf{v}_c)^-). \end{aligned} \tag{B.8}$$

Similarly, the LLF flux for the dispersed phase is

$$\begin{aligned} &\underline{\mathbf{H}\left((\alpha_d\mathbf{v}_d)^+, (\alpha_d\mathbf{v}_d)^-, \mathbf{n}^+\right)} = \\ &\frac{1}{2}((\alpha_d\mathbf{v}_d\otimes\mathbf{v}_d)^+ + (\alpha_d\mathbf{v}_d\otimes\mathbf{v}_d)^-) - \frac{1}{2}|(\mathbf{v}_d\cdot\mathbf{n})\mathbf{n}|((\alpha_d\mathbf{v}_d)^+ - (\alpha_d\mathbf{v}_d)^-). \end{aligned} \tag{B.9}$$

B.2 Proof of Symmetric Gradient Identity

Lemma B.2.1. $\epsilon(\mathbf{u}) : \nabla(\mathbf{v}) = \epsilon(\mathbf{u}) : \epsilon(\mathbf{v})$.

Proof. Using the fact that the symmetric gradient is $\epsilon(\mathbf{u}) = \frac{1}{2}(\nabla(\mathbf{u}) + \nabla(\mathbf{u})^T)$, then

$$\begin{aligned} \epsilon(\mathbf{u}) : \nabla(\mathbf{v}) &= \frac{1}{2}(\nabla(\mathbf{u}) + \nabla(\mathbf{u})^T) : \nabla(\mathbf{v}) \\ &= \frac{1}{2}(\nabla(\mathbf{u}) : \nabla(\mathbf{v}) + \nabla(\mathbf{u})^T : \nabla(\mathbf{v})) \\ &= \frac{1}{4}(\nabla(\mathbf{u}) : \nabla(\mathbf{v}) + \nabla(\mathbf{u}) : \nabla(\mathbf{v}) + \nabla(\mathbf{u})^T : \nabla(\mathbf{v}) + \nabla(\mathbf{u})^T : \nabla(\mathbf{v})) \\ &= \frac{1}{4}(\nabla(\mathbf{u}) : \nabla(\mathbf{v}) + \nabla(\mathbf{u})^T : \nabla(\mathbf{v}) + \nabla(\mathbf{u}) : \nabla(\mathbf{v})^T + \nabla(\mathbf{u})^T : \nabla(\mathbf{v})^T) \\ &= \frac{1}{2}(\nabla(\mathbf{u}) + \nabla(\mathbf{u})^T) : \frac{1}{2}(\nabla(\mathbf{v}) + \nabla(\mathbf{v})^T) \\ &= \epsilon(\mathbf{u}) : \epsilon(\mathbf{v}). \end{aligned} \tag{B.10}$$

□