Computational Science and Engineering
(International Master's Program)

Technical University of Munich

Master's Thesis

Partitioned flow simulations with preCICE and OpenFOAM

Markus Mühlhäußer

Computational Science and Engineering
(International Master's Program)

Technical University of Munich

Master's Thesis

Partitioned flow simulations with preCICE and OpenFOAM

Author:	Markus Mühlhäußer
Examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor:	Gerasimos Chourdakis, M.Sc.
Submission date:	December 27th, 2022

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

December 27th, 2022 Markus Mühlhäußer

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Gerasimos Chourdakis, for his invaluable patience and feedback at all times. His constant motivation, his belief in my work and the attempts to make my research more organized made this thesis an incredible learning experience.

I also would like to thank the preCICE developer team and especially Benjamin Uekermann for showing interest in my work and including me to their coding days.

Furthermore, I would like to thank all of my dear friends who always believed in my academic abilities and helped me clear my head in stressful times. I greatly appreciate them being there for me whenever I reach out to them even when I neglected the friendships at times.

I also want to deeply thank my family, who always supported me in my studies. Finally, words cannot express my gratitude to my parents for their unconditional emotional and financial support, for always being there for me when I need them the most.

Abstract

Flow simulations are becoming increasingly complex and require dedicated solution strategies to solve large and complex scenarios. Example applications are found in hemodynamics, nuclear reactor simulations and, wind modeling. It can be beneficial to divide such complex flow domains into smaller subdomains, which can be simulated by solvers that best resolve the flow characteristics of the respective region. However, these subdomains need to be coupled. The coupling library preCICE allows to couple various popular fluid solvers with a black-box approach.

This thesis serves as validation for the fluid-fluid module of the preCICE OpenFOAM adapter. Before coupling multiple fluid solvers with varying features, the possibilities and limitations of coupling two similar fluid solvers should be assessed. I investigate incompressible laminar flow partitioned in two domains. The investigation is done using the OpenFOAM solvers icoFoam and pimpleFoam, with a Dirichlet-Neumann surface coupling via preCICE. Fully developed flow can be coupled perfectly using mass-flux aware boundary conditions. In that case, the maximum relative error for velocity compared to the monolithic solution is in the order of 10^{-5}. Coupling of a developing flow profile on a coarse mesh results in maximum errors in the order of 10^{-2} at the cells next to the interface. The error depends linearly on the magnitude of the velocity gradient and the cell size. In cells further away from the interface, the error is approximately one order smaller. Higher order of accuracy around the coupling interface can be reached by manipulating OpenFOAM solvers and thereby abandoning the non-invasive plugin approach of the adapter and the idea of preCICE to treat solvers as black boxes.

Having established a baseline for the error that can be expected in fluid-fluid coupling, I extend the fluid-fluid module of the OpenFOAM adapter to support more complex flows and cover a wider range of OpenFOAM solvers. I add temperature to the coupled variables using a Dirichlet-Neumann coupling approach. The relative error for temperature obtained in the validation case using buoyantPimpleFoam is in the order of 10^{-6}. A further addition to the fluid-fluid module is custom inlet-outlet boundary conditions, which allow backflow across the coupling interface. They incorporate the flux-aware behavior that I established for uni-directional flow coupling and they can be set on both sides of the interface, regardless of the flow direction.

Overall, this thesis provides an understanding for the capabilities and limitations of coupling laminar flows with the fluid-fluid module of the preCICE OpenFOAM adapter and paves the path towards coupling more complex fluid flows in the future.

Contents

1 Introduction

Simulation of systems where different regions are best described by different mathematical models are called multiphysics simulations and are becoming increasingly interesting to computational engineers and researchers. Some examples of multiphysics simulations include fluid-structure interaction and conjugate heat transfer problems. Both problems usually involve a fluid and a solid, both of which are described by different equations, which interact with each other in a certain way. Simulating the fluid-solid system can be done by combining all relevant equations into a big coupled system that is solved monolithically. This approach has proven to be efficient and robust for certain applications, however it is usually very specific to one application and its involved physics.

In contrast to the monolithic approach, a new focus has been set on solving multiphysics simulations by partitioning the system into multiple regions, each of them being modeled by only the equations that best describe that region. This on one hand reduces the mathematical complexity of the equations that need to be solved on each subdomain, but on the other hand raises new challenges when it comes to the question of how these subdomains can be coupled in a numerically stable manner. Benjamin Uekermann goes into great detail about this partitioned approach in his dissertation [16]. There, he also outlines the major advantage of the partitioned simulation approach, which is the great flexibility and reusability of physics solvers. Therefore, optimizing the partitioned approach will be much more sustainable in the future and reduce the cost of development drastically.

All flow problems can, in theory, be described by the full Navier-Stokes equations. The Navier-Stokes equations are a set of three-dimensional, non-linear, partial-differential equations. They include variables such as velocity, pressure, temperature, density, and flow phenomena such as convection, diffusion, and turbulence. This set of equations is inherently complex and cannot be solved analytically. Instead, the equations are discretized in space and time, resulting in a finite linear system that can be solved numerically. Still, depending on the resolution of the discretization, computing fluid motion is extremely computationally intensive. As a result, we usually use simplified models that are sufficient to describe the expected flow characteristics. For example, a fluid flow may be modeled as turbulent or laminar, as compressible or incompressible, as three-dimensional or two-dimensional.

Nowadays, flow simulations are becoming increasingly large and complex. In a monolithic simulation approach, we need to solve the Navier-Stokes model that includes all flow characteristics that are present anywhere in the whole domain. However, analogously to the above-mentioned multiphysics simulations, a partitioned approach could be utilized, where each subdomain is modeled differently. Thus, we can utilize the most

efficient solvers for the underlying flow phenomena inside each subdomain and we can even couple solvers of different discretization methods, such as finite volume, finite element, or even lattice Boltzmann methods.

1.1 Goal of this thesis

The goal of this thesis is to partition flow simulations using the coupling library preCICE [3] (Precise Code Interaction Coupling Environment). preCICE provides the necessary coupling functionality that we need in order to couple two OpenFOAM fluid solvers. OpenFOAM [20] (Open-source Field Operation And Manipulation) is a toolbox of many fluid solvers implemented on the basis of the finite volume method.

We use a Dirichlet-Neumann coupling scheme across a common interface of two non-overlapping domains. Other works involve Neumann-Neumann coupling or for overlapping domains, Dirichlet-Dirichlet coupling [14]. In this thesis, I couple identical fluid solvers to build a basis of what is the minimum that can be expected from flow partitioning. The concepts can easily be transferred to coupling different fluid solvers that communicate shared flow parameters. I use laminar, incompressible solvers that need to exchange the solution variables velocity and pressure. I try to adhere to the general principles of preCICE of being non-invasive and treating involved solvers as black boxes. However, I also look into theoretical improvements that could be made by making the coupling procedure invasive and specific to OpenFOAM solvers. This reduces the flexibility of our coupling environment, but it would still allow coupling most solvers from the OpenFOAM arsenal, including solvers from different OpenFOAM distributions.

The ultimate goal is to provide an interface that can couple all OpenFOAM fluid solvers. Therefore, I extend the existing fluid-fluid module of the preCICE OpenFOAM adapter by coupling temperature and providing interface boundary conditions that do not depend on the flow direction. These are the first steps towards a flexible coupling library that can couple fluid solvers considering any flow characteristics that are part of the complete Navier-Stokes equations.

1.2 Previous work

Coupling multiple single-physics solvers to simulate a complex multiphysics system is a popular approach in application fields such as fluid-structure interaction [4] or conjugate heat transfer [19]. However, there are not as many resources when it comes to fluid-fluid coupling. Most research has been conducted on domain decomposition approaches with the goal to parallelize fluid simulations. Domain decomposition for parallelization often has the advantage of a consistent grid and consistent solver characteristic across the coupled interface. OpenFOAM uses such a parallelization approach, where each processor computes only a subdomain and communicates quantities at the interface to other processors via MPI [11].

Again with parallelization and scalability in mind, Grinberg et al. implement a Dirichlet-Neumann coupling between multiple fluid domains [9]. The coupling is done for a spectral element method by exchanging values of pressure, velocity and velocity gradients. They identify the coupling interface as an error source that needs to be revised.

Fernández et al. developed an explicit scheme for coupling two fluid domains, both solved by 3-dimensional Navier-Stokes equations [6]. Utilizing the Nitsche's interface method, they successfully couple fluid domains via a Robin-Robin coupling scheme for the finite element discretization. They identified an imbalance of static powers that arises in the Dirichlet-Neumann coupling of [9] and address it by specific total pressure formulations. However, the resulting expressions are specific to the finite element method and difficult to translate to a finite volume implementation.

As it was mentioned in Section 1.1, ultimately, we are also interested in coupling fluid solvers based on different mathematical models. As an example, Wittenberg and Neumann coupled an OpenFOAM finite volume solver with a molecular dynamics simulation using their framework MaMiCo [21]. In her dissertation on urban wind flow simulations, Camps Santasmasas used a setup where she coupled OpenFOAM with a lattice Boltzmann method using a modified fork of the preCICE OpenFOAM adapter [15]. Instead of using the suggested interface coupling, she created an overlap region from where the coupling data is obtained.

1.3 Structure of the thesis

The following Chapter 2 is a refresher on the theory of fluid simulations. I go over the governing flow equations and give an introduction to the finite volume method and its solution algorithms as it is implemented by OpenFOAM. In Chapter 3 I present the tools that I utilize in this thesis: OpenFOAM and preCICE. For both tools I give a brief introduction and then focus on the aspects that are relevant for the rest of the thesis.

Chapter 4 describes the investigation process of the fluid-fluid coupling, where we look into three validation cases of increasing complexity, each coming with additional challenges. After explaining the basic fluid-fluid coupling terminology in Section 4.1, a laminar, fully developed pipe flow is coupled in Section 4.2. Next, Section 4.3 presents a case where the coupling interface is placed into still developing flow. The inherently non-zero velocity gradient that we experience here presents an additional challenge. Lastly, Section 4.4 presents a deformed pipe which demonstrates the additional challenges that come with a non-orthogonal mesh. In Section 4.5, I talk about another coupling approach that might perform better, but is more invasive and very specific to the OpenFOAM numerics. Following the investigation, Chapter 5 presents extensions to the OpenFOAM preCICE adapter, regarding heat transfer in fluids and a coupling interface that can handle flow in both directions. Both implementations are supported by a validation case. In Chapter 6, I will give some concrete recommendations on fluid-fluid coupling with preCICE based on the results of this thesis. Chapter 7 summarizes the conclusions of this work and gives a

brief outlook on how to advance fluid-fluid coupling.

I recommend reading Chapter 4 in order presented, to understand the decisions that were made in the process. If only the extensions to the adapter in Chapter 5 are of interest, I suggest to at least look at the conclusions of Section 4.2 and Section 4.3 of the investigation, as otherwise some parts of the setup configuration might be confusing.

2 Theory

In this chapter, we want to go over the fundamentals of the mathematical equations that describe the flow of fluids and then discuss the methods which are used by OpenFOAM to solve these equations. We will go over the basics of the finite volume method and explain some algorithms that are used by OpenFOAM to implement this method. This chapter will only scratch the surface of what lies behind these methods and should be used as a reference throughout this thesis. For a deeper understanding and more complete derivations, I recommend textbooks that cover the finite volume method in completeness ([5] [18]).

2.1 The governing equations

Fluid motion in general is described by the Navier-Stokes equations, a set of partial differential equations that are solved for the solution variables pressure p and the vector quantity velocity $\boldsymbol{U}$. The first equation describes the conservation of mass and is also called the continuity equation, with ρ being the fluid density:

$$\frac{\partial}{\partial t}\rho = -\nabla \cdot (\rho \boldsymbol{U}) \tag{2.1}$$

The second equation is the momentum equation which represents the conservation of momentum. It is a three-dimensional vector equation and a typical transport equation containing a time derivative on the left-hand side and convective, diffusive and source terms on the right-hand side:

$$\underbrace{\frac{\partial}{\partial t}\rho \boldsymbol{U}}_{\text{time derivative}} = \underbrace{-\nabla \cdot (\rho \boldsymbol{U} \otimes \boldsymbol{U})}_{\text{convective term}} + \underbrace{\nabla \cdot \boldsymbol{\tau}}_{\text{diffusive term}} - \underbrace{\nabla p + \rho \boldsymbol{g}}_{\text{source terms}} \tag{2.2}$$

τ is the viscous stress tensor and $\boldsymbol{g}$ the gravitational force as an example of a common external force. Equation 2.1 and Equation 2.2 are the main equations that need to be solved to model the movement of fluids. The energy equation is the third equation and describes the conservation of total energy. The complete energy equation is quite complex and is often simplified by only solving for the internal energy. The transport equation for internal

energy equation can be expressed in terms of enthalpy h as:

$$\underbrace{\frac{\partial}{\partial t}\rho h}_{\text{time derivative}} = -\underbrace{\nabla\cdot(\rho h\boldsymbol{U})}_{\text{convective term}} + \underbrace{\nabla\cdot(k\nabla T)}_{\text{diffusive term}} - \underbrace{\frac{\partial p}{\partial t}+S}_{\text{source/sink terms}}, \tag{2.3}$$

where the enthalpy h relates to the internal energy e by:

$$h = e + \frac{p}{\rho} \tag{2.4}$$

In Equation 2.3, k is the thermal conductivity of the fluid. From the internal enthalpy, the temperature can be deducted when the material's physical properties, such as specific heat capacity, are known. When the fluid's temperature is considered to be constant or not relevant to the fluid's flow profile, the energy equation can be neglected. Furthermore, most solution methods can't resolve the Navier-Stokes equations for highly turbulent flow. In these cases, various simplified turbulent models can be used, many of which require additional transport equations to be solved.

For the most part of this thesis, we will work with laminar flow of constant temperature, neglecting energy and turbulence equations. Moreover, we consider incompressible fluids, meaning the density ρ stays constant in both time and space. This simplifies the continuity Equation 2.1 to:

$$\nabla\boldsymbol{U} = 0 \tag{2.5}$$

In the incompressible momentum equation, we get rid of ρ by dividing through the constant density. The stress tensor τ in the diffusive term can be described by the laplacian of the velocity multiplied with the kinematic viscosity ν. This leads to the following equation:

$$\frac{\partial}{\partial t}\boldsymbol{U} = -\nabla\cdot(\boldsymbol{U}\otimes\boldsymbol{U}) + \nu\nabla^2\boldsymbol{U} - \nabla\hat{p} + \boldsymbol{g} \tag{2.6}$$

The pressure quantity $\hat{p}$ in Equation 2.6 is the pressure divided by density p/ρ and has the dimensions of m^2/s^2. When speaking of pressure in an incompressible context, we refer to $\hat{p}$ simply as p.

2.2 The Finite Volume Method

The main principle for most computational fluid dynamic methods is to transform the partial differential equations from Section 2.1 into a discretized system of linear equations. The finite volume method is especially popular, because its discretization is carried out directly in the physical space and therefore rather straight forward to implement. Furthermore, it inherently focuses on fluxes between control volumes and ensuring conservation, which makes it very suitable for fluid dynamic problems.

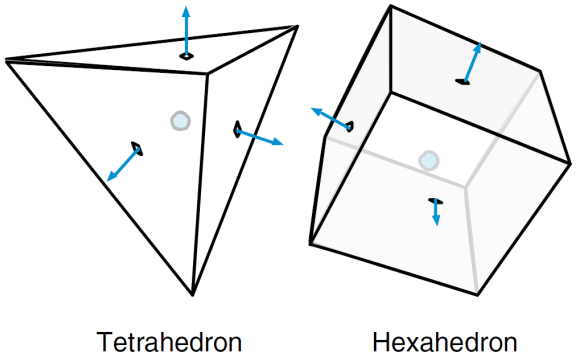


Figure 2.1: Tetrahedra made of four triangles and hexahedra made of six quadrilaterals are the most common cell shapes for finite volume meshes[5].

In the finite volume method, the simulated domain is divided into many smaller control volumes (cells), for each of which the conservation of mass, momentum and energy is ensured by balancing the fluxes at each cell's surface. The whole system of smaller cells that are connected by their faces is called the mesh. In theory, a mesh can consist of any polyhedral cells with polygonal faces. In practice, most meshes use hexahedral or tetrahedral cells or a mix of both, depicted in Figure 2.1.

OpenFOAM uses a cell-centered domain discretization and a face-addressing mesh storage. Being cell-centered means that all quantities, such as velocity or pressure, are stored in the center of the cell and the value can be seen as an average that applies in the whole control volume. All cells are described by their surrounding faces, which are shared by neighboring cells. Each face is part of exactly two cells, where the cell with lower index is assigned as the face owner and the other cell is called neighbor. Boundary faces, which build up the surface of the domain, do not have a neigbour cell.

Having set up the finite volume mesh, the differential equations can be discretized by using the values stored in the cell center. The general transport equation for an incompressible flow for an arbitrary quantity ϕ can be expressed as:

$$\underbrace{\frac{\partial}{\partial t}\phi}_{\text{time derivative}} = - \underbrace{\nabla \cdot (\boldsymbol{U}\phi)}_{\text{convective term}} + \underbrace{\nabla \cdot (D\nabla\phi)}_{\text{diffusive term}} + \underbrace{S_\phi}_{\text{source term}} \tag{2.7}$$

This equation is integrated over all control volumes V as:

$$\int_V \frac{\partial}{\partial t}\phi dV = -\int_V \nabla \cdot (\boldsymbol{U}\phi)dV + \int_V \nabla \cdot (D\nabla\phi)dV + \int_V S_\phi dV \tag{2.8}$$

With the help of the Gauss theorem, the convective and diffusive volume integrals of Equa-

tion 2.8 can be transformed into surface integrals to yield:

$$\frac{\partial}{\partial t}\int_V \phi dV = -\oint_{\partial V} d\boldsymbol{S}\cdot(\boldsymbol{U}\phi) + \oint_{\partial V} d\boldsymbol{S}\cdot(D\nabla\phi) + \int_V S_\phi dV \tag{2.9}$$

The surface integrals as well as the source term integral of Equation 2.9 can be approximated by summations as:

$$\frac{\partial}{\partial t}\int_V \phi dV = -\sum_f \underbrace{\boldsymbol{S_f}\cdot(\boldsymbol{U}\phi)_f}_{\text{convective flux}} + \sum_f \underbrace{\boldsymbol{S_f}\cdot(D\nabla\phi)_f}_{\text{diffusive flux}} + S_\text{c}V + S_\text{p}V\phi \tag{2.10}$$

The source term in Equation 2.10 is divided into a constant part S_c and a non-linear part S_p [10]. The surface integrals are expressed as summations of the respective fluxes over all faces f of the polyhedral control volume. Fluxes in general describe how much of a quantity flows through a surface and are calculated by multiplying the quantity with the surface area. For a vector quantity, this is done by taking the dot product with the face vector $\boldsymbol{S_f}$. In OpenFOAM, the convective flux $(\boldsymbol{S_f}\cdot\boldsymbol{U})$ from Equation 2.10 is stored in a separate field called `phi`. Later on, the equation which is now discretized in space needs to also be discretized in time. For this and more detailed explanations of the Gauss theorem, I refer to further literature such as the extensive book about the finite volume method by Moukalled et al. [5].

Looking at Equation 2.10, we can see that we need several quantities at the face centers. These are calculated by interpolating between the values of the adjacent cell centers. In the case of boundary faces, we need to be able to deduct the values from the given boundary conditions. In the end, we can assemble a linear system of equations $\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{b}$, where $\boldsymbol{\phi}$ is the solution vector of all ϕ at the cell centers, $\boldsymbol{b}$ is the source vector and $\boldsymbol{A}$ the coefficient matrix. All discretized terms can be divided into implicit contributions that are part of $\boldsymbol{A}$ and explicit contributions that are added to $\boldsymbol{b}$. Having built the linear system of equations, we can choose from many numerical methods for solving such. The next chapter describes the general algorithms used by OpenFOAM which decide what matrices we need to assemble and how we solve for each of the solution variables.

2.3 The OpenFOAM solution algorithms

The continuity Equation 2.5 and the momentum Equation 2.6 build up a system of coupled differential equations that we want to solve for pressure and all three components of the velocity. As there is no explicit pressure equation, we cannot solve for pressure directly, but instead we use the continuity equation as a restriction to momentum equation. To do so in practice, several iterative solution strategies have been invented. In this thesis, I work with mainly two different solvers: icoFoam and pimpleFoam. icoFoam uses the solution algorithm PISO, pimpleFoam is named after the algorithm PIMPLE. Both of these solvers can be seen as extensions to the SIMPLE algorithm, which I am going to explain first.

These algorithms all belong to the family of segregated solvers, meaning that we solve two individual equations for velocity and pressure instead of one large system. The original formulations of these algorithms as well as the derivations found in most textbooks work on a staggered grid and solve the equations for the variable corrections only (see e.g. [5]). For this thesis, we will look into the adaptations by OpenFOAM which work directly on the solution variables stored in the finite volume mesh.

2.3.1 The SIMPLE algorithm

The SIMPLE algorithm stands for Semi-Implicit Method for Pressure-Linked Equations. It is an iterative predictor-corrector method targeting steady-state flow. It starts with an initial guess for pressure and velocity, which, in OpenFOAM, are the initial conditions given in the 0 time directory. The initial values of the face flux ϕ are calculated from the velocity field and are needed in the convection term of the discretized momentum equation. The first step after initialization involves assembling the general momentum matrix $\boldsymbol{M}$ of the discretized momentum equation, which can be expressed in matrix form as

$$\boldsymbol{M}\boldsymbol{U} - \boldsymbol{b} = -\nabla p, \tag{2.11}$$

where $\boldsymbol{b}$ contains the source terms of the discretization. In a first momentum predictor step, Equation 2.11 is solved to obtain an intermediate predicted velocity field. Next, the solver splits the coefficient matrix $\boldsymbol{M}$ into a diagonal matrix $\boldsymbol{A}$ and a matrix $\boldsymbol{H}(\boldsymbol{U})$, containing all the off-diagonal components and the source term $\boldsymbol{b}$:

$$\boldsymbol{A}\boldsymbol{U} - \boldsymbol{H}(\boldsymbol{U}) = -\nabla p \tag{2.12}$$

The matrix $\boldsymbol{H}(\boldsymbol{U})$ is calculated explicitly from the previously predicted velocity values. Rearranging Equation 2.12 gives us:

$$\boldsymbol{U} = \frac{\boldsymbol{H}(\boldsymbol{U})}{\boldsymbol{A}} - \frac{\nabla p}{\boldsymbol{A}} \tag{2.13}$$

Because $\boldsymbol{A}$ is a diagonal matrix, it can be inverted very easily. Finally, we can use the expression for $\boldsymbol{U}$ from Equation 2.13 and substitute it into the continuity Equation 2.5 to get

$$\nabla \cdot \left(\frac{\boldsymbol{H}}{\boldsymbol{A}} - \frac{\nabla p}{\boldsymbol{A}} \right) = 0, \tag{2.14}$$

which, on the next turn, can be rearranged to form the pressure Poisson equation:

$$\nabla \cdot \frac{\boldsymbol{H}}{\boldsymbol{A}} = \nabla \cdot \frac{\nabla p}{\boldsymbol{A}} \tag{2.15}$$

The next step of the SIMPLE algorithm involves solving Equation 2.15 for pressure p.

In a final momentum corrector step, we correct the velocity as in Equation 2.13 with the newly calculated, corrected pressure values. Doing so, we ensure that the corrected velocity values satisfy the continuity equation. However, unless we have reached the converged steady-state solution, the corrected velocity values no longer satisfy the momentum equations. Thus, the SIMPLE algorithm proceeds to the next iteration by setting the pressure corrected values as the initial guesses of the next time step. Figure 2.2 shows the flow chart for the full SIMPLE algorithm as it is used by OpenFOAM. It also shows the optional steps for solving the energy equation, which is usually done between the momentum predictor step and the pressure solving step. If turbulence modeling is added to the solver, any additional equations are solved in the end of each time step.

2.3.2 The PISO algorithm

The Pressure Implicit with Splitting of Operators (PISO) algorithm is very similar to the SIMPLE algorithm, with an additional pressure correction step. In OpenFOAM, an additional PISO loop is added around the pressure equation, where the number of corrections can be chosen by the user. While the SIMPLE solvers are mostly being used for steady-state scenarios, the additional pressure correction loop accelerates the convergence of the solution and makes these algorithms suitable for transient simulations. Figure 2.3 shows the full PISO algorithm as it is implemented in OpenFOAM.

2.3.3 The PIMPLE algorithm

The SIMPLE and PISO algorithm can be merged into one algorithm known as PIMPLE. The PIMPLE algorithm implements an additional loop over all equations, which basically allows to run the SIMPLE algorithm multiple times before advancing to the next time step. By combining features of both SIMPLE and PISO, the PIMPLE algorithm is the most versatile and can be used both for transient and steady-state simulations. Additionally, it allows using larger time steps than in the PISO algorithm. Figure 2.4 shows the flow graph for the PIMPLE algorithm.

2.4 Boundary Conditions

Fluid dynamics problems need to be constraint by initial and boundary conditions. Boundary conditions specify what happens to the relevant flow variables at the boundary of the geometric domain. To achieve the desired simulation results, it is crucial that the user sets the correct boundary conditions, which can be challenging, as they must fulfil the following conditions [8]:

- They need to reflect the physical conditions of the simulated case.
- They need to make the case well-posed, such that they provide a unique, stable and physical solution to all involved equations.

- They must be compatible across multiple coupled equations, which applies in particular to pressure and velocity.

To solve a discretized system of second order differential equations, we need information about the surface-normal gradient and the value of our variable at the boundary, because we cannot infer these values from interpolation. There exist two main types of boundary conditions corresponding to both required quantities. All other boundary conditions can be derived from the other two basic conditions. The first one is called the Dirichlet boundary condition and prescribes a fixed value ϕ_{ref} that the solution of the variable needs to take on that boundary.

$$\phi_{\mathrm{b}} = \phi_{\mathrm{ref}} \tag{2.16}$$

The second type is the Neumann boundary condition which fixes the gradient normal to the boundary $\nabla_{\mathrm{n}}\phi_{\mathrm{ref}}$ for that variable.

$$\nabla_{\mathrm{n}}\phi_{\mathrm{b}} = \nabla_{\mathrm{n}}\phi_{\mathrm{ref}} \tag{2.17}$$

Having set the value or gradient explicitly, the respective other value can be calculated implicitly with the help of the discretization parameters. Subsection 3.1.6 will go into more detail about the boundary conditions and their implementations in OpenFOAM.

2.5 Heat transfer

In OpenFOAM, the heat distribution is calculated by solving an additional energy equation. In Section 2.1, we explained that the energy equation is often replaced by a transport equation for enthalpy h (see Equation 2.3). The enthalpy relates to the temperature T in respect to the specific heat capacity c_{p} of the fluid as:

$$c_{\mathrm{p}} = \frac{\partial h}{\partial T} \tag{2.18}$$

Instead of deriving a complete transport equation for temperature, we will look at the general concepts of heat transfer.

Heat is transferred in fluids in mainly three different mechanisms: Conduction, convection and radiation [17]. We will only look into heat transfer by conduction and convection. Conduction is the heat transfer on a molecular scale. Through vibrations and direct contact, internal energy is transferred from molecules of higher energy to molecules of lower energy. Thermal conduction is explained by the Fourier law of heat conduction:

$$q = -k\nabla T \tag{2.19}$$

q is the heat flux density, which describes the amount of heat that is transferred through $1m^2$ per second. ∇T is the temperature gradient and k is the thermal conductivity of the

material.

Convection describes the heat transfer caused by the movement of the fluid. The fluid movement can occur naturally in the presence of gravity due to the fluid density gradient between locations of different temperature. Forced convection on the other hand describes the process of transporting heat through an externally generated flow, e.g. by a fan. The basic relationship for heat transfer by convection is described by Newton's law for convection:

$$q = h(T - T_\infty), \tag{2.20}$$

where h is the heat transfer coefficient, T is the temperature at the interface and T_∞ is the fluid temperature far away from the coupling interface. The heat transfer coefficient h is a variable parameter that depends on the physical parameters of the fluid and the general physical situation.

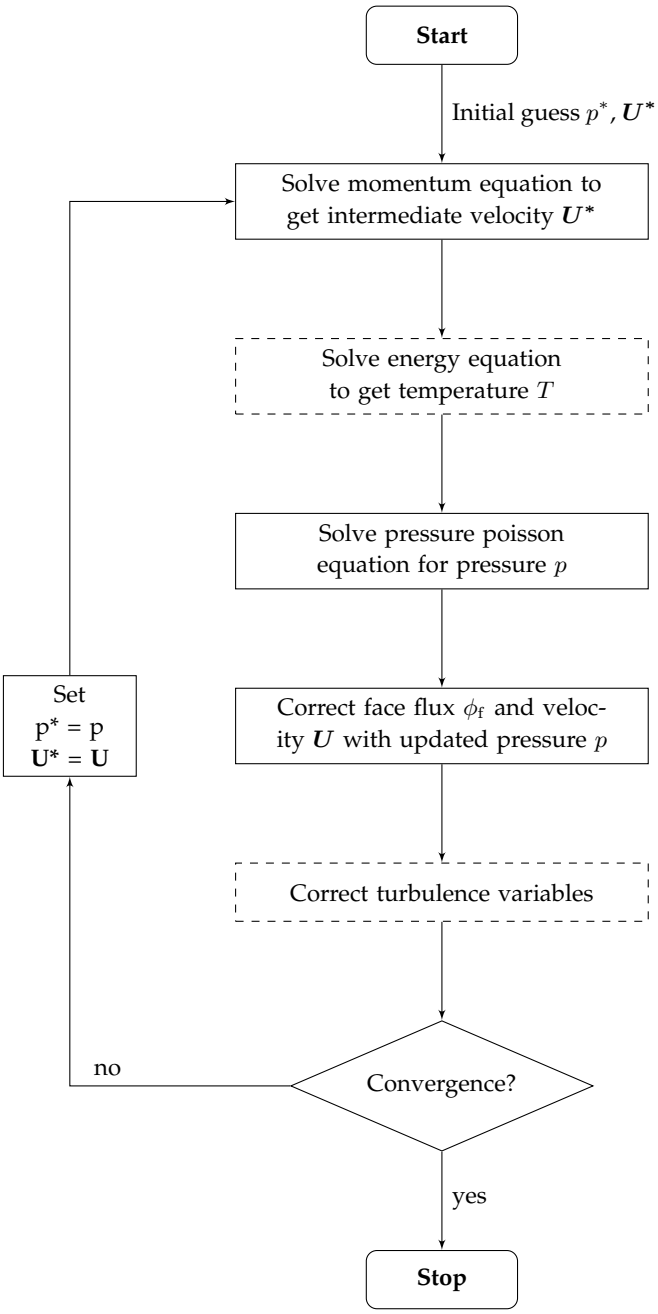


Figure 2.2: The SIMPLE algorithm

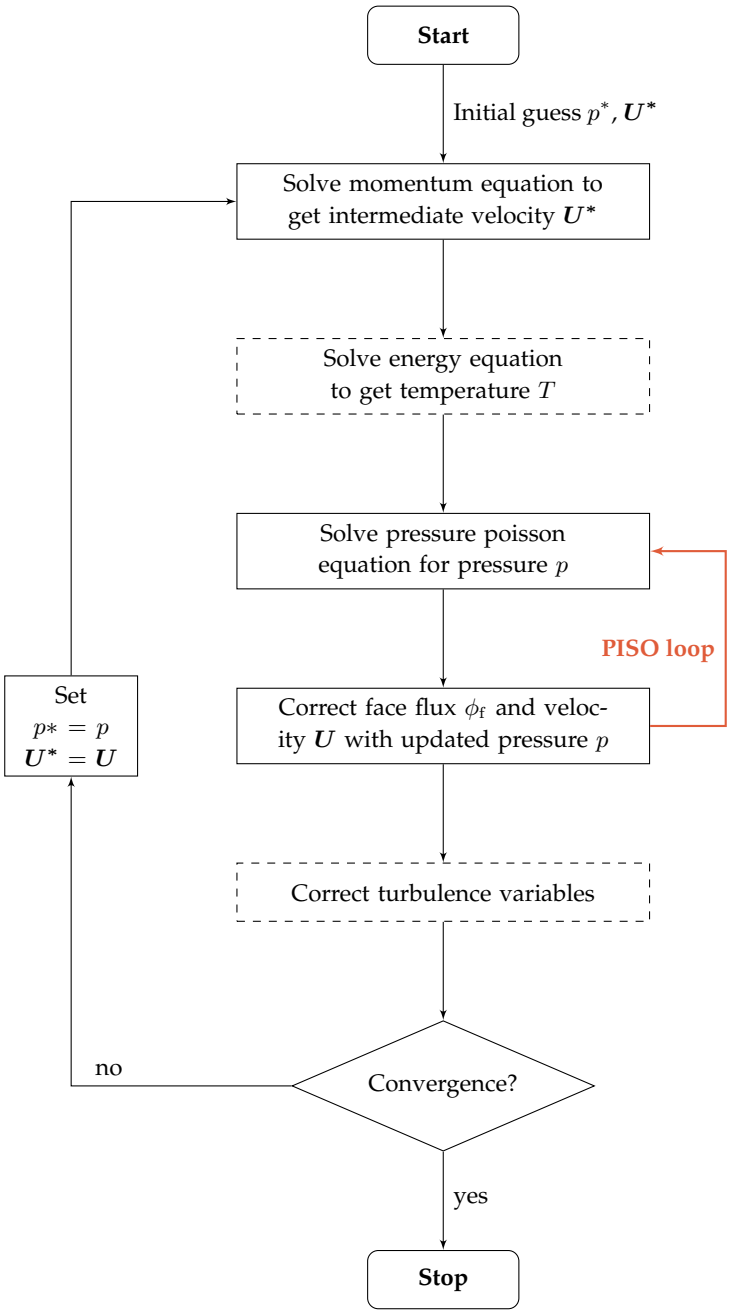


Figure 2.3: The PISO algorithm

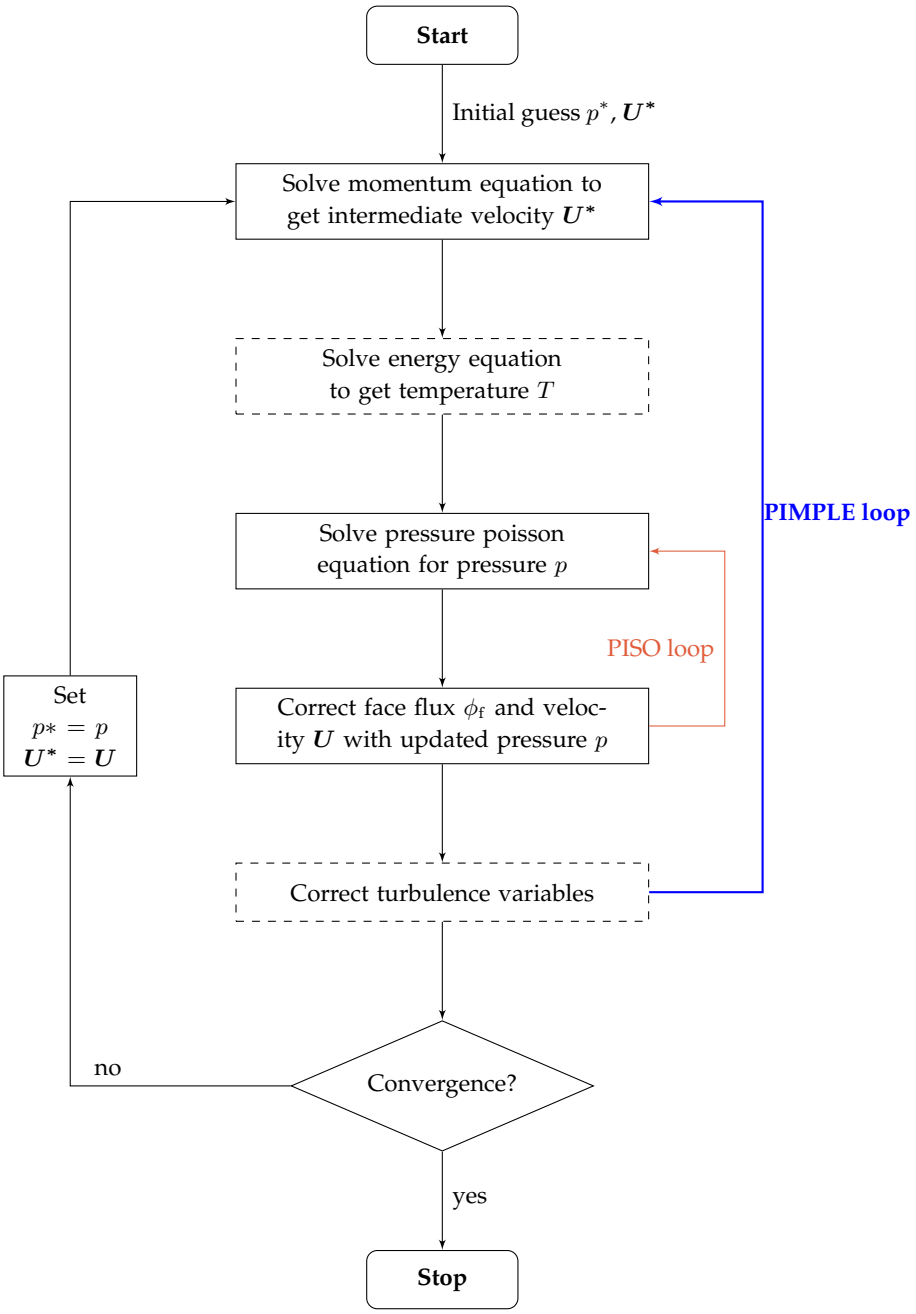


Figure 2.4: The PIMPLE algorithm

3 Tools

The investigations on in this thesis are done using of OpenFOAM and preCICE. We look at OpenFOAM first, the program that I am going to use for the fluid simulations. I go over its basic functionality and take a closer look into its boundary condition implementation, which will be relevant during Chapter 4 and Chapter 5. Afterwards, I present preCICE, the tool which allows coupling multiple OpenFOAM instances. I show the fundamental concepts of preCICE and how to use it in combination with OpenFOAM.

3.1 OpenFOAM

OpenFOAM (Open-source Field Operation And Manipulation) is a C++ library designed primarily for three-dimensional continuum mechanics [20]. It offers a wide variety of applications for solving complex fluid flow computations including chemical reactions, heat transfer, and turbulence[1]. Further solvers for problems in e.g. solid dynamics, electromagnetics, molecular dynamics or finance are also available. Accompanying the solver applications, OpenFOAM provides a range of utilities for pre- and post-processing.

OpenFOAM is, as its name suggests, open source and freely available under the GNU General Public License. There are multiple forks and variations of OpenFOAM, the main three being:

- OpenFOAM by ESI-OpenCFD (openfoam.com)
- OpenFOAM by the OpenFOAM Foundation (openfoam.org)
- foam-extend by Wikki Ltd (wikki.co.uk)

In the context of this thesis, OpenFOAM refers to version v2112 by ESI-OpenCDF, which we will use throughout this thesis. New releases of this variant are scheduled every six months and the source code can be retrieved from GitLab[2]. Users of OpenFOAM have the possibility to easily modify existing solvers or add new ones, provided they have the required knowledge about the involved physics and programming techniques. Each solver

[1] List of standard OpenFOAM solvers:
https://www.openfoam.com/documentation/user-guide/a-reference/a.1-standard-solvers

[2] OpenFOAM by ESI-OpenCFD source code:
https://develop.openfoam.com/Development/openfoam

and utility is its own application that can be executed in the system's command line interface. Since OpenFOAM has no native graphical user interface, the results, which are stored as plain text files, must be visualized by external programs such as ParaView.

3.1.1 The OpenFOAM case structure

Even though there are many different solvers that need to be configured individually, there exists a general case folder structure that most solvers follow. Figure 3.1 shows the file tree of a case for the icoFoam solver.

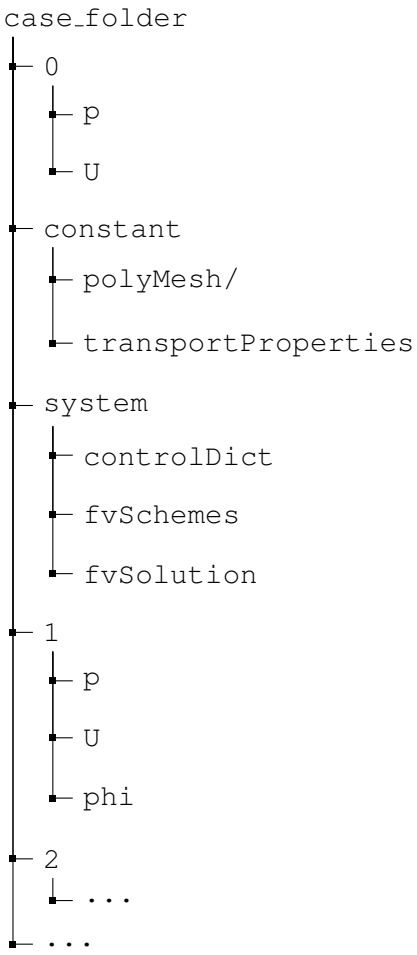


Figure 3.1: Basic folder structure of an OpenFOAM case for the icoFoam solver. It contains the time directories `0`, `1`, `2`, `...` as well as the configuration directories `constant` and `system`

Most solvers require the three directories `0/`, `system/`, and `constant/` at the start

of the simulation, as they contain the simulation settings and case configuration. The `0/` folder is the initial time directory and contains the initial and boundary conditions for all of the solver's required fields. In the case of icoFoam, these are the pressure and velocity fields. The `constant/` directory contains the mesh information inside `polyMesh/`, as well as all the variables that stay constant throughout the simulation. Because icoFoam is an incompressible solver, the viscosity `nu` is specified in the `transportProperties` dictionary. In OpenFOAM terminology, all input text files are called dictionaries. The dictionaries inside the `system/` folder contain all the necessary settings for the solver to run, such as runtime and output information in `controlDict`, numerical schemes in `fvSchemes` or algorithm controls in `fvSolution`. The additional time directories are created by OpenFOAM during the execution of the solver, depending on the output controls that are set in the `controlDict`.

3.1.2 The flow of an OpenFOAM solver

Each OpenFOAM solver is a separate C++ application and thus defines its own `main()` function. In the beginning, most applications include the fundamental finite volume code from OpenFOAM, containing all the necessary data structures, discretization schemes and so on. Listing 3.1 shows the source code for icoFoam, which is a very basic solver for incompressible, laminar flow.

At the start of the program, the solver sets up the scenario by reading in all the configuration dictionaries found in the case folder. This is done by inlining the header files such as `createTime.H`, `createMesh`, or `createFields.H`, which contain routines for creating the time object and the mesh or initializing the required physical fields. These files are often defined globally, but some solvers have their locally defined versions with necessary adaptations specific to the application. After the case has been set up, the general time loop is entered to execute the solver-dependent algorithm. In the case of icoFoam, it is the PISO algorithm, which is explained in Subsection 2.3.2. In general, this means assembling the equations and the matrices that arise from the discretization schemes. These matrix equations are solved numerically and the results are written to the disk after each time step by `runTime.write()`.

3.1.3 OpenFOAM equation language

The OpenFOAM source code is a heavily templated C++ framework, that can easily overwhelm users of moderate programming knowledge. Therefore, it is goal of OpenFOAM to abstract the code details and implement the partial differential equations that need to be solved, in a descriptive way [20]. Listing 3.2 shows how the basic momentum equation

$$\frac{\partial \rho \boldsymbol{U}}{\partial t} + \nabla \cdot \phi \boldsymbol{U} - \nabla \cdot \mu \nabla \boldsymbol{U} = -\nabla p \tag{3.1}$$

```
#include "fvCFD.H"
#include "pisoControl.H"

int main(int argc, char *argv[])
{
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"
    pisoControl piso(mesh);
    #include "createFields.H"
    #include "initContinuityErrs.H"

    while (runTime.loop())
    {
        // Momentum predictor
        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );

        if (piso.momentumPredictor())
        {
            solve(UEqn == -fvc::grad(p));
        }

        // --- PISO loop
        while (piso.correct())
        {
            ...
        }

        runTime.write();
    }

    return 0;
}
```

Listing 3.1: `FOAM_APP/solvers/incompressible/icoFoam/icoFoam.C` Excerpt showing the `main` function of the `icoFoam` solver.

would be represented in OpenFOAM source code. It can be seen that each term of Equation 3.1 can be matched directly to a function in the code. This allows the user to quickly understand what equations are solved in each application and makes it easy for him to assemble his own additional equations. Further operators that might be needed for differential equations can be found in the OpenFOAM programming guide [13].

```
solve
    (
    fvm::ddt(rho, U)
    + fvm::div(phi, U)
    - fvm::laplacian(mu, U)
    ==
    - fvc::grad(p)
    );
```

Listing 3.2: OpenFOAM representation of the basic momentum equation [13].

3.1.4 OpenFOAM function objects

OpenFOAM provides an interface for extending existing solvers without the need to modify their source code. This can be done via so-called function objects, which can be loaded at runtime and which offer the possibility to execute additional code during every iteration of the solver's time loop. This functionality is mainly used for post-processing tools that handle output data at runtime. There are tools for calculating and writing derived fields, sampling and monitoring data, converting the output format and more [7]. The function objects can be loaded by specifying them as arguments in the command line, or by adding them to the `controlDict` file. Users can program and add custom function objects by directly writing the code into the `controlDict`. For more complex tools, it is recommended to build a shared library that can then be included as a function object.

All function objects inherit from the `functionObject` base class[3], which implements the methods that the OpenFOAM solvers call at specific points during the simulation. Every solver in OpenFOAM has at least a `Time` object that stores and regulates the time information during the simulation. This `Time` object makes calls to the list of loaded function objects at the start and the end of the time loop and once every iteration. Additional calls to the function objects are made whenever the time step size or the mesh changes. These are the only time points when a function object can execute its own code.

[3] OpenFOAM API Guide v2112, functionObject class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1functionObject.html

3.1.5 The OpenFOAM object registry

As mentioned before, OpenFOAM is a framework with many different solvers and each of these solvers needs different methods, schemes or data. Therefore, OpenFOAM has many different classes that build up the finite volume toolbox. By abstraction and templating, the code stays flexible and each solver can use the components that are required for its particular method to solving the underlying differential equations. While each solver handles things differently and uses different fields and quantities, some things are the same for all solvers. These common objects are bundled in a single framework, called the `objectRegistry`.

The `objectRegistry` is basically a hierarchical database that stores references to all common objects with additional information like name or input and output control. Common registered objects include the mesh, the basic fields like velocity and pressure or the configuration dictionaries like `fvSchemes` and `fvSolution`. When we want to access a solver's data from external code such as functionObjects, we can only access objects that have been registered in the `objectRegistry`. If we know the name of an object, say the velocity field U, then we can retrieve a reference to that object with a call to the function `lookupObject<Type>("U")`. Fields and other objects that are created within the individual solvers that are not registered in the `objectRegistry` are not accessible from other classes such as boundary conditions or function objects.

3.1.6 Boundary conditions in OpenFOAM

In Section 2.4 we have already established that the correct boundary conditions are crucial to the simulation outcome. As OpenFOAM uses the finite volume method to discretize the flow governing equations, boundary values must be specified on all boundary faces of the mesh. These boundary faces are grouped into patches, which are defined by the user during the mesh creation. Each boundary patch usually describes a different physical boundary for our model. There exist different types of patches, such as `patch`, `wall`, or `empty`, which help OpenFOAM to better "understand" the mesh and also limit the allowed boundary conditions for the user. Wall patches for example support special wall function boundary conditions and the empty patches are used for two-dimensional simulations and signal OpenFOAM the dimension in which the equations do not need to be solved. Attributes, such as the name and type of the patch, as well as geometric information of the associated boundary faces including face areas or distances to cell centers are stored in the class `fvPatch`[4].

The user defines all the necessary boundary patches of the mesh and assigns boundary conditions to all these patches, such that the physical problem is modeled appropriately. The boundary conditions for each flow variable are specified in the respective files in-

[4] OpenFOAM API Guide v2112, fvPatch class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fvPatch.html

side the `0/` directory. OpenFOAM offers a wide range of different boundary conditions, ranging from simple Dirichlet and Neumann conditions to more specialized ones, such as periodic boundary conditions. A full list of available boundary conditions can be found in the OpenFOAM user guide [12]. The functionality of these boundary conditions is implemented in individual classes that all inherit from `fvPatchField`[5].

All boundary conditions need an implementation of the functions `evaluate()` and `snGrad()`, which return the boundary face value and the normal gradient respectively. Both functions evaluate the results in an explicit manner based on the current variable fields. However, when the transport equations are discretized into a finite volume matrix equation of the form $\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{b}$, there are two terms to which boundary faces contribute implicitly. For the discretized advection term we need an expression for the boundary face values $\boldsymbol{\phi_{bf}}$ and for the diffusion term we need an expression for the boundary face normal gradients $\boldsymbol{\nabla_n \phi_{bf}}$. For many boundary conditions, the expressions for face value and gradient can be split into an explicit part and an implicit part that is multiplied with the adjacent cell value $\boldsymbol{\phi_c}$. The implicit part contributes to the coefficient matrix $\boldsymbol{A}$, whereas the explicit part will be added to the source term $\boldsymbol{b}$. The respective contributions are evaluated by the following four functions that the `fvPatchField` class implements:

- `Foam::Field valueInternalCoeffs()`
- `Foam::Field valueBoundaryCoeffs()`
- `Foam::Field gradientInternalCoeffs()`
- `Foam::Field gradientBoundaryCoeffs()`

With $\boldsymbol{\phi_c}$ being the value at the adjacent internal cell, the patch face values are expressed in the following way:

$$\boldsymbol{\phi_{bf}} = \texttt{valueInternalCoeffs()} * \boldsymbol{\phi_c} + \texttt{valueBoundaryCoeffs()} \tag{3.2}$$

Similarly, the gradient at the boundary faces is expressed with the gradient coefficients as:

$$\boldsymbol{\nabla_n \phi_{bf}} = \texttt{gradientInternalCoeffs()} * \boldsymbol{\phi_c} + \texttt{gradientBoundaryCoeffs()} \tag{3.3}$$

The internal coefficients in Equation 3.2 and Equation 3.3 represent the implicit part that is multiplied with $\boldsymbol{\phi_c}$ and as such is added to the finite volume matrix. The boundary coefficients, on the other hand, are the explicit part that is added to the source term $\boldsymbol{b}$. Listing 3.3 shows the declarations for all above-mentioned methods in the file `fvPatchField.H`.

[5] OpenFOAM API Guide v2112, fvPatchField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fvPatchField.html

In the following, the OpenFOAM implementation of some relevant boundary conditions is described. Their expressions of face value and face gradient are given in the form of Equation 3.2 and Equation 3.3.

fixedValue

The `fixedValue` boundary condition is the OpenFOAM equivalent to a standard Dirichlet boundary condition and is implemented in the `fixedValueFvPatchField`[6] class. When specifying a `fixedValue` boundary condition, we also provide reference values ϕ_{ref} for each boundary face. The face values $\boldsymbol{\phi_{\mathrm{bf}}}$ simply evaluate to

$$\phi_{\mathrm{bf}} = \phi_{\mathrm{ref}} \tag{3.4}$$

The face normal gradients can then be expressed by using a forward difference approximation:

$$\nabla_{\mathrm{n}}(\phi)_{\mathrm{bf}} = \frac{\phi_{\mathrm{ref}} - \phi_{\mathrm{c}}}{\Delta} = -C_{\Delta}\phi_{\mathrm{c}} + C_{\Delta}\phi_{\mathrm{ref}} \tag{3.5}$$

Δ is the distance from the cell center to the boundary face center. However, OpenFOAM does not store the distances directly but instead uses the inverse of the distance $C_{\Delta} = 1/\Delta$. Therefore, we will find the expression to the right of Equation 3.5 in the source code.

fixedGradient

Classic Neumann conditions are realized with the `fixedGradient` boundary condition implemented in `fixedGradientFvPatchField`[7]. In this case, the boundary face values are not given explicitly, but are extrapolated from the neighboring internal cell values ϕ_{c}, with the help of the provided gradient $\nabla_{\mathrm{n}}(\phi)_{\mathrm{ref}}$ and the delta coefficients C_{Δ}.

$$\phi_{\mathrm{bf}} = \phi_{\mathrm{c}} + \frac{\nabla_{\mathrm{n}}(\phi)_{\mathrm{ref}}}{C_{\Delta}} \tag{3.6}$$

However, the boundary face gradient can simply be set to the given reference gradient:

$$\nabla_{\mathrm{n}}(\phi)_{\mathrm{bf}} = \nabla_{\mathrm{n}}(\phi)_{\mathrm{ref}} \tag{3.7}$$

As a special case, there also exists the `zeroGradient` boundary condition, which can be used when the gradient of a variable is assumed to be zero at the boundary. The

[6] OpenFOAM API Guide v2112, fixedValueFvPatchField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fixedValueFvPatchField.html

[7] OpenFOAM API Guide v2112, fixedGradientFvPatchField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fixedGradientFvPatchField.html

`zeroGradient` boundary condition is evaluated by simply setting the boundary face values to the internal cell values, which is the result of setting $\nabla(\phi)_{\text{ref}}$ to zero in Equation 3.6. This boundary condition is used for most quantities other than pressure at domain outlets, where a fully developed flow is assumed.

inletOutlet

The `inletOutlet` condition, implemented in `inletOutletFvPatchField`[8], is based on the `mixed` boundary condition. The `mixed` boundary condition is a linear blend of the `fixedValue` and `fixedGradient` condition with an additional value fraction parameter v. This condition is formally known as a Robin boundary condition. The expression for the boundary value is:

$$\phi_{\text{bf}} = (1 - v)\phi_{\text{c}} + v\phi_{\text{ref}} + (1 - v)\frac{\nabla(\phi)_{\text{ref}}}{C_{\Delta}} \tag{3.8}$$

The face gradient is evaluated as

$$\nabla_{\text{n}}(\phi)_{\text{bf}} = -vC_{\Delta}\phi_{\text{c}} + vC_{\Delta}\phi_{\text{ref}} + (1 - v)\nabla(\phi)_{\text{ref}} \tag{3.9}$$

The value fraction parameter v operates as a blending factor between 0 and 1, where $v = 0$ corresponds to the `fixedGradient` condition and $v = 1$ is equal to the `fixedValue` boundary condition.

The `inletOutlet` condition is meant for open boundary patches, where we do not know if there will be an outflow or inflow at the boundary faces. The parameter v is used as a switch between `fixedValue` and `fixedGradient` based on the flow direction at the patch. The flow direction is automatically evaluated from the sign of the boundary face flux `phi`. The user only needs to set the reference value ϕ_{ref} for inflow, $\nabla_{\text{n}}(\phi)_{\text{ref}}$ is set to zero during the initialization. The outflow condition is therefore evaluated as `zeroGradient`.

fixedFluxExtrapolatedPressure

The `fixedFluxExtrapolatedPressureFvPatchScalarField`[9] class implements a boundary condition that is a special case of the `fixedGradient` condition for pressure. In contrast to the `fixedGradient` condition, the user does not specify the gradient beforehand, but it is calculated by the solver at each time step. This boundary condition for

[8] OpenFOAM API Guide v2112, inletOutletFvPatchField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1inletOutletFvPatchField.html

[9] OpenFOAM API Guide v2112, fixedFluxExtrapolatedPressureFvPatchScalarField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fixedFluxExtrapolatedPressureFvPatchScalarField.html

pressure is supposed to be combined with a `fixedValue` velocity boundary condition. The API documentation says that the pressure gradient is set to the provided value, such that the flux on the boundary is that specified by the velocity boundary condition. We will have a deeper look into this boundary condition during our investigation of fluid-fluid coupling in Subsection 4.2.4.

3.2 preCICE

preCICE[10] (Precise Code Interaction Coupling Environment) is a C++ library for partitioned multi-physics simulations. It couples multiple single-physics solvers, each of which simulates only a designated subpart of the whole multiphysics scenario. This approach offers high flexibility, as no case-specific solvers need to be developed. Instead, multiple existing solvers, also called participants, can be combined to solve a complex system. Common multiphysics problems that can be solved with the help of preCICE include conjugate heat transfer (CHT) and fluid-structure interaction (FSI) [3]. Besides its core library, preCICE offers adapters for a multitude of existing solvers such as OpenFOAM, FEniCS, CalculiX, and many more. Furthermore, it provides a high-level application programming interface (API) with C++, C, Fortran, Python and Matlab bindings, that allows the users to add their own solvers to the coupled simulations. Next to the official adapters, the community provides a wide variety of additional unofficial ones. A full list can be found on the preCICE website[11]. Figure 3.2 shows an overview of how the preCICE library, its adapters and the respective solvers interact with each other.

The external solvers are treated as "black boxes", which means that preCICE has no knowledge about each participant's internal numerics. It only reads the output of each solver and modifies their input, but does not interfere during a simulation iteration. While also partially supporting volume coupling, most preCICE adapters are designed for surface coupling, where two coupling participants share a common interface. In that case, solvers are only modified by setting boundary conditions at the common interface, depending on the boundary values of the other participant. Doing so, preCICE is kept minimally invasive and flexible.

The communication between the individual solvers, involving the exchange of interface data, is done by either MPI or TCP/IP sockets. The exchanged data must be mapped to match the grid of each participant, which might not be consistent across two solvers. preCICE provides multiple advanced data mapping schemes that allow a heterogeneous coupling interface, where solvers can use meshes of different refinements and even different discretization methods such as finite volume or finite elements. Furthermore, the solvers are allowed to run with different time step sizes, since preCICE handles the synchronization of all participants.

[10] preCICE website: https://precice.org/
[11] Overview of available preCICE adapters: https://precice.org/adapters-overview.html

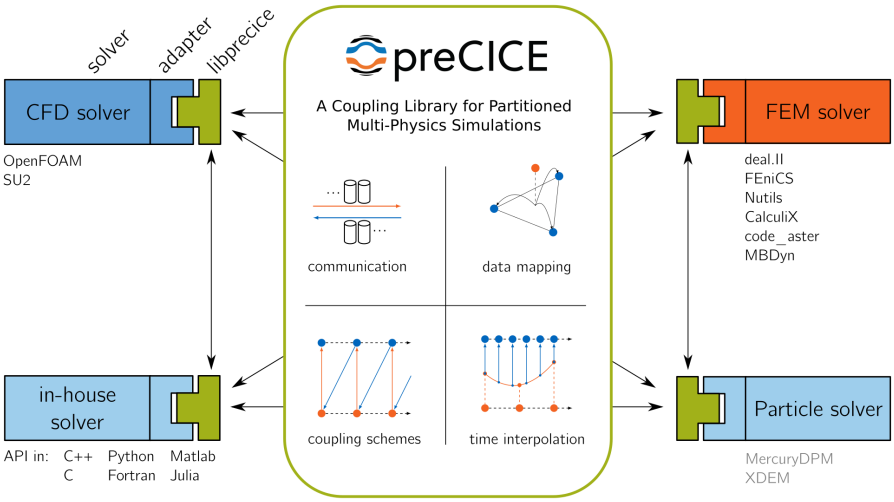


Figure 3.2: preCICE overview [3]

The coupling scheme can be chosen to be either serial or parallel and either explicit or implicit. For serial coupling, both solvers take alternating turns and wait for the other solver to finish so they can use their latest output data. During parallel coupling, both solvers run at the same time and use the other solver's output data from the last time step. When explicit coupling is specified, each solver is running chronologically, one time step after another, with updated boundary values whenever preCICE interferes. However, for a meaningful multiphysics simulation, the physical fields at the interface must be in agreement for both solvers. This requires an iterative, fixed-point method that checks for convergence of the interface values at each time step. Therefore, when implicit coupling is specified, preCICE repeats each time step of the simulation multiple times until the output values from both solvers match at the interface up to a specified convergence limit. Plain implicit coupling however corresponds to a mathematical fixed-point iteration, which can be unstable and slow. That is why preCICE provides multiple acceleration methods, such as under-relaxation or quasi-Newton, that can reduce the number of iterations for the implicit coupling.

In the end, the user specifies the coupled solver, the exchanged data, and the coupling configuration in an XML file that is read by preCICE. Listing 3.4 shows an example of such configuration file for fluid-fluid coupling. More information on the XML parameters can be found on the preCICE website[12].

[12] preCICE configuration file reference:
https://precice.org/configuration-xml-reference.html

3.2.1 The OpenFOAM adapter

The OpenFOAM adapter [1] is one of preCICE's most versatile adapters. It supports OpenFOAM solvers for conjugate heat transfer, fluid-structure interaction and fluid-fluid coupling. The necessary data handling classes are divided into problem-specific modules. The core of the OpenFOAM adapter is implemented as an OpenFOAM function object [1]. This means that the adapter does not touch any OpenFOAM solver's source code and can be loaded at runtime. Function objects are called at the end of every time step. The OpenFOAM adapter, when being called, checks the time step and then reads the interface boundary values that are required by the preCICE coupling. During implicit coupling, it also stores a copy of all required fields from the last time step. If another implicit coupling iteration is needed, then the solver's time step (which OpenFOAM has already advanced) is reset and the field values from the last time step are restored.

In order to bridge the gap between the general preCICE configuration to the case-specific OpenFOAM setup, a file called `preciceDict` is placed in the solver's system directory. This is an OpenFOAM style dictionary file that can be read using OpenFOAM's input-output functions. It provides the path to the preCICE configuration file, specifies the adapter's relevant module and the solver's name as a preCICE participant. Furthermore, it lists all the specified coupling interfaces, which need to be mapped to boundary patches of the OpenFOAM mesh. It then lists all the data that is exchanged on that interface by preCICE. Listing 3.5 shows an example `preciceDict` file for a fluid-fluid coupling setup. It is important to note that the names of the participant, the mesh and the data must match the names specified in the preCICE configuration.

The Fluid-Fluid module

The Fluid-Fluid module is the latest addition to the preCICE OpenFOAM adapter and was implemented by Gerasimos Chourdakis in the context of his dissertation [3]. It is currently still at an experimental stage. The module consists of one `FluidFluid` class for initialization and configuration and four flow-relevant `CouplingDataUser` classes. The four quantities that are implemented for coupling fluid solvers are: Velocity, Pressure, VelocityGradient and PressureGradient.

The adapter stores references to the OpenFOAM velocity and pressure fields. The values needed for the preCICE coupling are read from the interface boundary of the respective field reference. The boundary gradients can be retrieved by a call to the boundary's `snGrad()` function, which calculates the gradient with a forward difference approximation from the cell center values to the boundary values. The OpenFOAM boundary patch that is supposed to read Pressure or Velocity from preCICE needs to be initialized with the `fixedValue` boundary condition. The participants reading PressureGradient or VelocityGradient must use a `fixedGradient` boundary condition.

```
//- Evaluate the patch field, sets Updated to false
virtual void evaluate
(
    const Pstream::commsTypes commsType =
        Pstream::commsTypes::blocking
);

//- Return patch-normal gradient
virtual tmp<Field<Type>> snGrad() const;

//- Return the matrix diagonal coefficients corresponding to the
//  evaluation of the value of this patchField with given weights
virtual tmp<Field<Type>> valueInternalCoeffs
(
    const tmp<Field<scalar>>&
) const
{}

//- Return the matrix source coefficients corresponding to the
//  evaluation of the value of this patchField with given weights
virtual tmp<Field<Type>> valueBoundaryCoeffs
(
    const tmp<Field<scalar>>&
) const
{}

//- Return the matrix diagonal coefficients corresponding to the
//  evaluation of the gradient of this patchField
virtual tmp<Field<Type>> gradientInternalCoeffs() const
{}

//- Return the matrix source coefficients corresponding to the
//  evaluation of the gradient of this patchField
virtual tmp<Field<Type>> gradientBoundaryCoeffs() const
{}
```

Listing 3.3: `FOAM_SRC/finiteVolume/fields/fvPatchFields/fvPatchField/fvPatchField.H` - Modified Excerpt
The `fvPatchField` class is the base class for all boundary conditions. Shown are the declarations of the most important methods that define a boundary conditions behavior. (see Subsection 3.1.6)

```
<precice-configuration>
  <solver-interface dimensions="3">
  ...
    <participant name="Fluid1">
      <write-data name="Velocity" mesh="Fluid1-Mesh" />
      <read-data name="Pressure" mesh="Fluid1-Mesh" />
      <mapping:nearest-neighbor constraint="consistent"/>
    </participant>

    <participant name="Fluid2">
      <read-data name="Velocity" mesh="Fluid2-Mesh" />
      <write-data name="Pressure" mesh="Fluid2-Mesh" />
      <mapping:nearest-neighbor constraint="consistent" />
    </participant>

    <coupling-scheme:serial-implicit>
      <time-window-size value="0.01" />
      <max-time value="1.0" />
      <participants first="Fluid1" second="Fluid2" />
      <exchange data="Velocity" mesh="Fluid1-Mesh"
        from="Fluid1" to="Fluid2" />
      <exchange data="Pressure" mesh="Fluid2-Mesh"
        from="Fluid2" to="Fluid1" />

      <max-iterations value="100" />
      <relative-convergence-measure
        limit="1.0e-6" data="Pressure" mesh="Fluid2-Mesh" />
      <relative-convergence-measure
        limit="1.0e-6" data="Velocity" mesh="Fluid1-Mesh" />
      <acceleration:IQN-ILS> ... </acceleration:IQN-ILS>
    </coupling-scheme:serial-implicit>

  </solver-interface>
</precice-configuration>
```

Listing 3.4: Excerpt of a preCICE v2 configuration file for a fluid-fluid coupling case: The configuration defines the participants `Fluid1` and `Fluid2` with their respective meshes and the exchanged data `Velocity` and `Pressure`. Both participants use a consistent nearest-neighbor data mapping scheme. The coupling scheme is set to `serial-implicit` with the Quasi-Newton accelerator `IQN-ILS`.

```
preciceConfig "../precice-config.xml";

participant Fluid1;

modules(FF);

interfaces
{
Interface1
{
    mesh               Fluid1-Mesh;
    patches            (couplingOutlet);

    readData
    (
      Pressure
    );

    writeData
    (
      Velocity
    );
};
}
```

Listing 3.5: Example of a `preciceDict` that is placed in the case's `system` directory. The name of the participant, mesh, and data match the preCICE configuration file from Listing 3.4.

4 Investigation

During the 14th OpenFOAM workshop Chourdakis et al. [2] showed that the fluid-fluid coupling in principle works for coupling fluid solvers of different dimensionality. However, they also noted that the validation case of two coupled three-dimensional solvers did not generate the desired accuracy, especially around the coupling interface.

This chapter describes the investigative process in which I attempt to find a better solution and to understand the sources of error. In Section 4.1, we look at the validation case from [2], which is a coupled case of laminar flow through a three-dimensional pipe. We look into the equations involved at the coupling interface and find adequate boundary conditions that improve the coupling results. In Section 4.3 we proceed to investigate a more complex pipe, where I place the coupling interface into non-developed flow. We are going to discover why this limits the accuracy that we can expect from fluid-fluid coupling via preCICE. As a last validation case, I am going to present a widening pipe case in Section 4.4, which is constructed by a non-orthogonal mesh. We investigate how the non-orthogonality of the finite volume cells impact the coupling results. Lastly, I present ideas in Section 4.5 on other coupling approaches that violate the preCICE concept of non-invasive black-box coupling.

4.1 Basic fluid coupling

The initial goal of this thesis is to solve the problem of the validation case presented in [2]: Partitioning the laminar flow through a straight pipe. During this investigation chapter we look into this and two other pipe-like coupling scenarios. All cases are steady-state scenarios, meaning that the flow converges to a solution that after a while does not further change over time. Furthermore, the flow across the coupling interface is laminar and unidirectional, so no backflow is expected at the coupling interface. Therefore, a coupling outlet and a coupling inlet can be defined for the partitioned scenario (see Figure 4.1). All cases are oriented such that the main flow direction is along the z-axis, with the inlet at $z = 0\mathrm{m}$. In the partitioned case, I call the participant containing the inlet the left half and the participant with the outlet the right half.

Before running the partitioned cases, each scenario is simulated monolithically. The converged result will then be used as a reference to which we can compare the coupled simulations. As a next step, I split up the domains in the middle across the z-axis and run the coupled simulations using preCICE.

Similarly to the standard pipe inlet, the coupling inlet is defined with a `fixedValue`

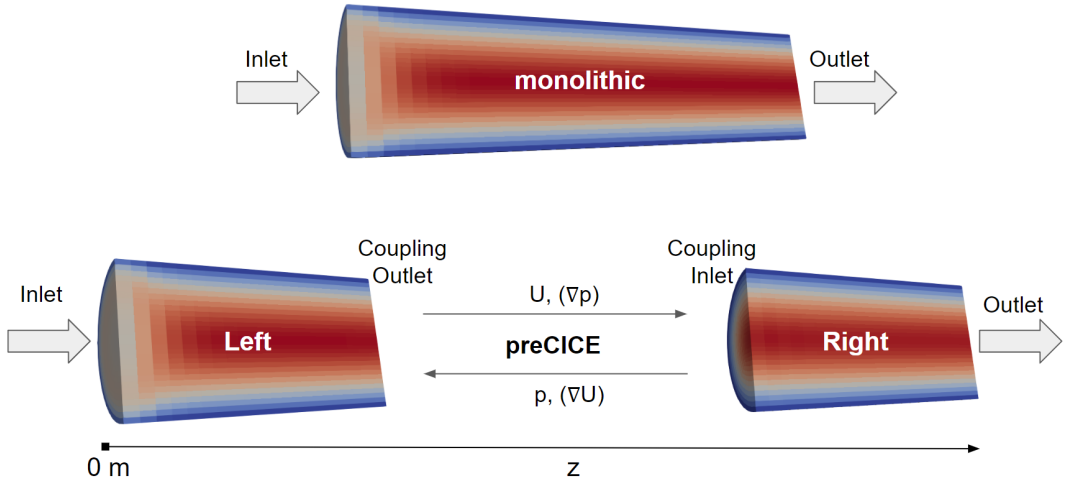


Figure 4.1: General case setup for the unidirectional, partitioned simulations with preCICE compared to a monolithic simulation.

velocity boundary condition and a `fixedGradient` pressure boundary condition. However, for the coupling inlet, the user-set reference values are just an initial guess. In each coupling time window, preCICE will adjust these to match the values obtained from the coupling outlet. The same idea applies to the coupling outlet, only the other way around, with `fixedValue` boundary condition for pressure and `fixedGradient` boundary condition for velocity. The Navier-Stokes equations in their most simplified, incompressible form, as in Equation 2.5 and Equation 2.6, need to be solved for velocity and pressure. Therefore, we always need to define boundary conditions for these two variables and always communicate at least these two quantities. The velocity values are sent from the coupling outlet to the coupling inlet and the pressure values the other way around. Additionally, the pressure and velocity gradients can optionally be communicated in the opposite direction to their respective normal values. Figure 4.1 gives an overview of the coupled quantities and their exchange directions.

4.2 The standard pipe

Simulating a laminar flow down a straight pipe is one of the most basic CFD problems. Since it is one of the most straightforward scenarios, it will represent the first case for this chapter. One usually sets a uniform velocity at the inlet, which then develops to the well-studied Poiseulle flow profile. The pipe set up in [2] had a 3D mesh of 20,000 cells and used the `pimpleFoam` solver. To better understand the details of what is happening at the coupling interface and to narrow down error sources, I simplify the case setup without changing the flow characteristics.

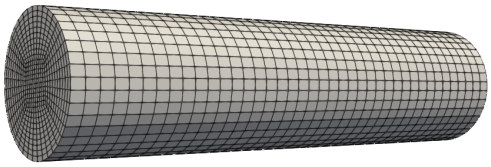
4.2.1 Simplifying the pipe problem

To narrow down error sources and to make the investigations simpler, I downscale the problem to a two-dimensional mesh with only 400 cells. The resulting mesh is 1x10x40 cells with uniform cells of size $1\mathrm{m} \cdot 1\mathrm{m} \cdot 1\mathrm{m}$. Apart from the geometry, most simulation parameters are kept the same as in the three-dimensional case. The fluid viscosity is $10\mathrm{m}^2/\mathrm{s}$. The boundary conditions are also the same: At the inlet, a $0.1\mathrm{m}/\mathrm{s}$ velocity Dirichlet condition and a zero gradient pressure Neumann condition, at the outlet a zero gradient velocity and pressure of $0\mathrm{m}^2/\mathrm{s}^2$. Since the problem is laminar and has no deforming mesh, the OpenFOAM solver is changed to `icoFoam` to further simplify the case. `icoFoam` is a transient solver for incompressible, laminar flow of Newtonian fluids and as such implements the basic Navier-Stokes equations as in Equation 2.5 and Equation 2.6 without external forces. Even though `icoFoam` is rarely used in practice, it is an excellent choice for understanding the basic flow computations.

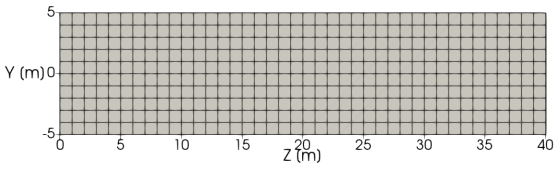
The whole downscaling process is illustrated in Figure 4.2. To evaluate the correctness of the coupled simulations, the velocity values are sampled along the center of the pipe. Since velocity and pressure are tightly coupled, it is usually enough to only look at the velocity graphs, which is usually more descriptive. It can be seen from the resulting graphs in Figure 4.2c and Figure 4.2d that the error characteristics at the coupling interface are still the same in the simplified pipe problem. Therefore, we can be quite certain that the error sources were not eliminated during the downscaling process. These initial velocity graphs show that for both cases, coupling of only velocity and pressure seems to produce the best results. Yet, the interface is not smooth. The reason for this lies in the details of the finite volume technique.

At the coupling inlet, I use a zero gradient Neumann condition for pressure. Because there are no-slip walls and a non-zero viscosity though, there is a steadily decreasing pressure along the length of the pipe in the theoretical solution. In the finite volume method, values are only stored in the cell centers and interpolated to the cell faces. Since the steady-state solution has a continuous pressure drop, every cell's pressure value is smaller than its left neighbor's value. Furthermore, the interpolated face value between two cells along the z-axis is always in between the values of those two cells. Because I use linear interpolation and uniform cell lengths in this case, the face values are always exactly in the middle. That also holds for the faces in the monolithic solution that are at $z = 20\mathrm{m}$. It is however, not true for the coupled boundary faces at $z = 20\mathrm{m}$ in the coupled scenario. Here, the value at the face is not in between the values of its neighboring cells, but because of the `zeroGradient` boundary condition it is the same as the value of the right cell. That is the reason we cannot get a perfectly smooth coupled solution by just communicating velocity and pressure itself. The same problematic is true for the `zeroGradient` velocity boundary condition at the coupling outlet, though in this particular case it is not a valid argument since there is in fact a zero gradient even in the continuous solution.

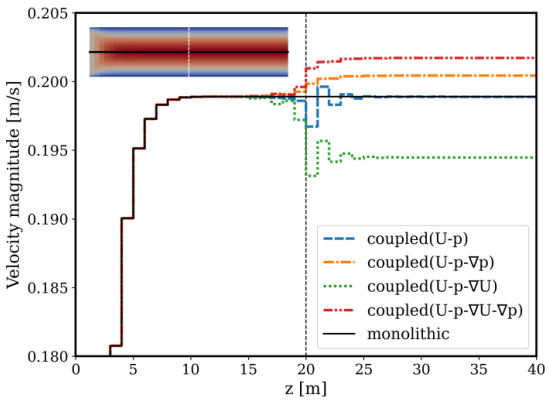
The above explanation leads to the idea that we need to additionally communicate the gradients to achieve the interface values that match the interpolated face values at $z =$



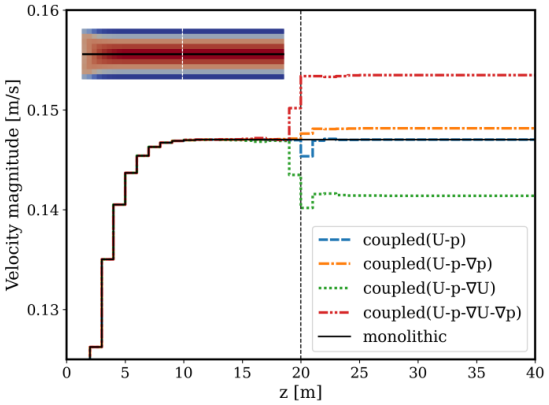
(a) Mesh of the 3D pipe



(b) Mesh of the 2D pipe



(c) Velocity magnitude along the center of the 3D pipe for various preCICE coupling configurations



(d) Velocity magnitude along the center of the 2D pipe for various preCICE coupling configurations

Figure 4.2: Comparison of the meshes and steady state solutions for several preCICE setups between the 3D pipe and the simplified 2D pipe.

20m of the monolithic solution. Adding the communication of the gradients in preCICE and using non-zero Neumann conditions would be the intuitive solution. Though it can be seen from the graphs in Figure 4.2 that this always leads to unphysical increases or decreases in the velocity, which violates the mass conservation law of the pipe domain.

4.2.2 Decoupling of the coupled simulation

As it was shown above for the coupled pipe scenario, there is always an error at the coupling interface, regardless of the communicated quantities via preCICE. When coupling the two OpenFOAM parts, only the values at the boundary faces are exchanged and the two solvers do not interact with each other further. Therefore, it is a requirement for a correctly converged coupled simulation that both parts converge to the correct solution by their own if the face values from the monolithic solution are set as fixed values for the boundary faces. This procedure is illustrated in Figure 4.3.

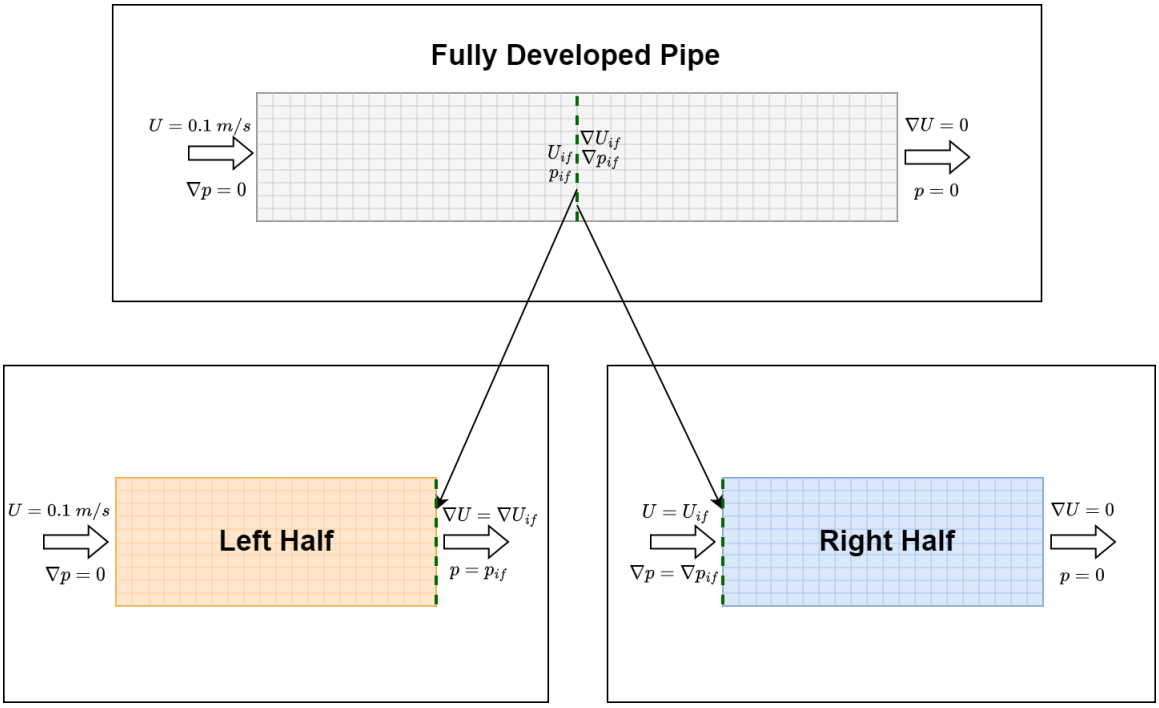


Figure 4.3: Decoupling the pipe scenario: Both coupling participants are simulated individually without preCICE interaction. The reference values ($U_{if}, p_{if}, ...$) for the coupling boundaries are taken from the face values in the fully converged monolithic solution that correspond to the coupling interface

4.2.3 The left half

The left half of the partitioned case contains the coupling outlet. I use the same boundary conditions for the coupling outlet as for the normal outlet. This is a `zeroGradient` velocity boundary condition and a `fixedValue` pressure boundary condition. As shown in Figure 4.3, I simulate the left half isolated, by setting the pressure values at the coupling outlet to those that are obtained from the monolithic solution. The velocity gradients at the center of the converged monolithic solution are all zero. Therefore, the `zeroGradient` boundary condition for the velocity is kept. Running the simulation of the left half with this setup leads to a solution that is identical to the corresponding cells in the full solution. Therefore, we can conclude that the coupling errors arise on the other side of the coupling interface.

4.2.4 The right half

The validation scenario for the coupling inlet consists of the isolated right half of the pipe test case. Again, I use similar boundary conditions for the coupling inlet as we would for a normal inlet, which are a `fixedValue` velocity boundary condition and a `zeroGradient` pressure boundary condition. Instead of the `zeroGradient` condition for pressure and a uniform velocity, the set pressure gradients and velocity values are those obtained from the cell faces at the same location in the fully converged monolithic solution as illustrated in Figure 4.3. These face values can be calculated by face interpolation of the cell-centered OpenFOAM fields. Simulating only the right half of the pipe test case with fixed reference boundary conditions that do not get modified by preCICE, a jump in the velocity can be observed very close to the inlet. Figure 4.4 shows the similarity of the error that is also obtained in the coupled case. This indicates that these boundary conditions are not suitable for the coupled case where pressure gradient is not zero at the coupling inlet. To understand why this is happening, we have to take a closer look at how inlet boundary conditions are handled by OpenFOAM solvers.

In Section 2.1, it was explained that we are looking for solutions of velocity and pressure that satisfy both the continuity and the momentum equations. Most OpenFOAM solvers create the matrix formulation Equation 2.13 of the momentum equation. This equation should also hold true at the boundary fields in:

$$\boldsymbol{U}_{\mathrm{bf}} = \left(\frac{\boldsymbol{H}}{\boldsymbol{A}}\right)_{\mathrm{bf}} - \left(\frac{\nabla p}{\boldsymbol{A}}\right)_{\mathrm{bf}} \tag{4.1}$$

At the inlet boundary, Equation 4.1 is very restricted because we set a fixed value for $\boldsymbol{U}_{\mathrm{bf}}$ and ∇p_{bf} with the boundary conditions. Therefore, to make sure the momentum equation stays fulfilled at the boundaries, we can only adjust the values of $\boldsymbol{A}$ and $\boldsymbol{H}$. Most OpenFOAM solvers create the intermediate vector field `HbyA`, which corresponds to the first term on the right side of Equation 4.1. OpenFOAM uses this field's boundary field values to correct for the momentum equation, right after initializing the field `HbyA`. More

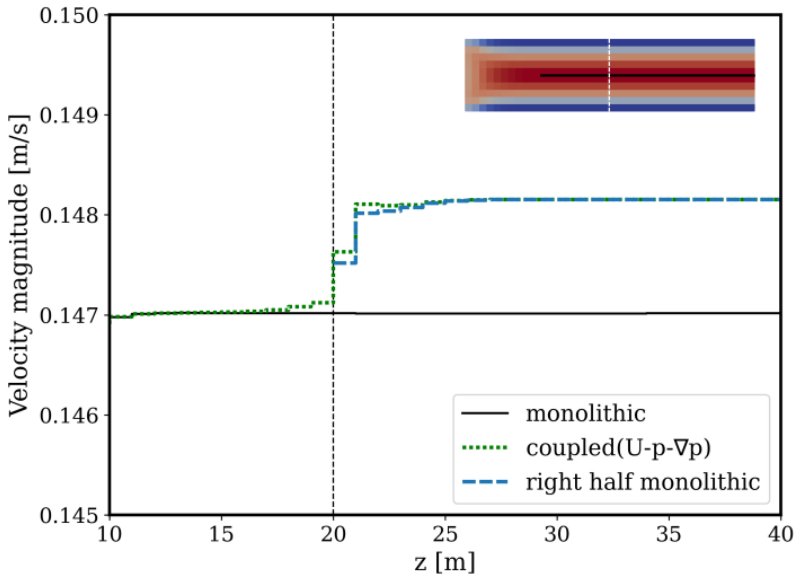


Figure 4.4: Velocity graphs of the isolated right half and the coupled case with `fixedValue` velocity and `fixedGradient` pressure boundary conditions at the coupling inlet.

specifically, this is done by the function `constrainHbyA()`, which is shown in Listing 4.1.

In line 7 of Listing 4.1, the code checks for boundary patches at which the velocity is not assignable. This evaluates to `true` for boundaries where the velocity is fixed, such as no-slip walls or a fixed velocity inlet. Additionally, it checks that the pressure boundary condition is not of the type `fixedFluxExtrapolatedPressure` at the same patch. If both conditions are fulfilled, then the `HbyA` boundary values are set to the velocity boundary values of the same patch as shown in Equation 4.2.

$$\left(\frac{\boldsymbol{H}}{\boldsymbol{A}}\right)_{\mathrm{bf}} = \boldsymbol{U}_{\mathrm{bf}} \tag{4.2}$$

By doing so, the momentum equations will be fulfilled at all patches where we have a fixed zero pressure gradient, which usually is the case at walls and inlets. This becomes clear when we compare Equation 4.2 with the momentum Equation 4.1 solved for `HbyA`:

$$\left(\frac{\boldsymbol{H}}{\boldsymbol{A}}\right)_{\mathrm{bf}} = \boldsymbol{U}_{\mathrm{bf}} + \left(\frac{\nabla p}{\boldsymbol{A}}\right)_{\mathrm{bf}} \tag{4.3}$$

Since the coupling inlet differs from a normal inlet in the way that its pressure gradient is not zero, the momentum equation will be violated at the patch after setting the `HbyA` values to the velocity values. A closer look at Listing 4.1 indicates that we can

```
volVectorField::Boundary& HbyAbf = HbyA.boundaryFieldRef();

forAll(U.boundaryField(), patchi)
{
    if
    (   // True for fixedValue velocity patches
        !U.boundaryField()[patchi].assignable()
    && !isA<fixedFluxExtrapolatedPressureFvPatchScalarField>
        (
            p.boundaryField()[patchi]
        )
    )
    {
        HbyAbf[patchi] = U.boundaryField()[patchi];
    }
}
```

Listing 4.1: `FOAM_SRC/finiteVolume/general/constrainHbyA/constrainHbyA.C`
Main logic of the function `constrainHbyA()`, which sets the boundary field values of `HbyA` to the boundary values of U at, e.g., wall and inlet patches.

avoid the described error by having a `fixedFluxExtrapolatedPressure` boundary condition for pressure at the coupling inlet. Instead of modifying the `HbyA` patch field, this boundary condition has a variable pressure gradient that is adjusted to fulfil the momentum equation. This is done by a call to `constrainPressure()`, which is usually called just before assembling the pressure equation. Listing 4.2 shows the code of the `constrainPressure()` function where the pressure normal gradient $\boldsymbol{n} \cdot \nabla p_{\text{bf}}$ is set according to Equation 4.4.

$$\boldsymbol{n} \cdot \nabla p_{\text{bf}} = \frac{\left(\left(\boldsymbol{S} \cdot \frac{\boldsymbol{H}}{\boldsymbol{A}} \rho \right)_{\text{bf}} - \rho_{\text{bf}} * MRF.\text{relative}(\boldsymbol{S}_{\text{bf}} \cdot \boldsymbol{U}_{\text{bf}}) \right)}{|\boldsymbol{S}_{\text{bf}}| \left(\frac{\rho}{\boldsymbol{A}} \right)_{\text{bf}}} \tag{4.4}$$

The `constrainPressure()` function is defined globally and called by almost all fluid solvers. For the incompressible case, ρ is defined as uniformly one and can be dropped from the equation. MRF in Equation 4.4 stands for Moving Reference Frame and is a method that is used in rotational fluid scenarios. The pipe is not a rotating case and this means that the term $MRF.\text{relative}()$ evaluates to 1 and therefore can be dropped as well. The field $\boldsymbol{S}$ is the surface vector field, which is obtained by multiplying the face normal vectors with the face area magnitudes: $\boldsymbol{S} = \boldsymbol{n} \cdot |\boldsymbol{S}|$. With these simplifications, we can turn Equation 4.4 to:

$$\boldsymbol{n} \cdot \nabla p_{\text{bf}} = \boldsymbol{n} \cdot \left[\left(\left(\frac{\boldsymbol{H}}{\boldsymbol{A}} \right)_{\text{bf}} - \boldsymbol{U}_{\text{bf}} \right) \Big/ \left(\frac{1}{\boldsymbol{A}} \right)_{\text{bf}} \right] \tag{4.5}$$

This is exactly the momentum Equation 4.1 solved for the pressure gradient multiplied by the surface normal $\boldsymbol{n}$. With that, we can conclude that at the coupling inlet, we need a `fixedFluxExtrapolatedPressure` boundary condition. Looking through the solver implementation of OpenFOAM shows, that this is the only possible pressure boundary condition to go along with a `fixedValue` velocity and an effective non-zero pressure gradient.

The blue, dashed graph in Figure 4.5 shows the results for the isolated right half after specifying the `fixedFluxExtrapolatedPressure` boundary condition. A minimal relative error in the order of 10^{-4} can be identified behind the inlet. However, this issue is most likely the result of manually interpolating the boundary velocity values rather than due to the chosen boundary conditions.

```
1  const fvMesh& mesh = p.mesh();
2
3  volScalarField::Boundary& pBf = p.boundaryFieldRef();
4  volVectorField::Boundary& UBf = U.boundaryField();
5
6  // HbyA flux field
7  surfaceScalarField::Boundary& phiHbyABf =
8      phiHbyA.boundaryField();
9  // density times rAU field
10 typename RAUType::Boundary& rhorAUBf = rhorAU.boundaryField();
11
12 surfaceVectorField::Boundary& SfBf = mesh.Sf().boundaryField();
13 surfaceScalarField::Boundary& magSfBf =
14     mesh.magSf().boundaryField();
15
16 forAll(pBf, patchi) {
17 // fixedFluxExtrapolatedPressure patch is an updateableSnGrad type
18     typedef updateablePatchTypes::updateableSnGrad snGradType;
19     const auto* snGradPtr = isA<snGradType>(pBf[patchi]);
20
21     if (snGradPtr)
22     {   // Calculate pressure gradient
23         const_cast<snGradType&>(*snGradPtr).updateSnGrad(
24             ( phiHbyABf[patchi]
25             - rho.boundaryField()[patchi]
26             *MRF.relative(SfBf[patchi] & UBf[patchi], patchi))
27             /(magSfBf[patchi]*rhorAUBf[patchi] )
28         );
29     }
30 }
```

Listing 4.2: `FOAM_SRC/finiteVolume/cfdTools/general/constrainPressure/constrainPressure.C`
The `constrainPressure()` function updates the pressure gradient according to Equation 4.4 for patch types that have an `updateableSnGrad` such as the `fixedFluxExtrapolatedPressure` patch fields.

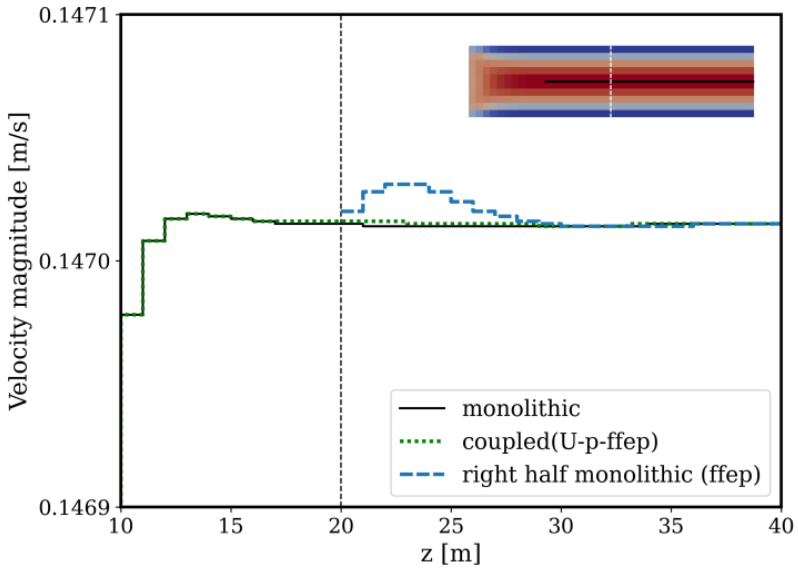
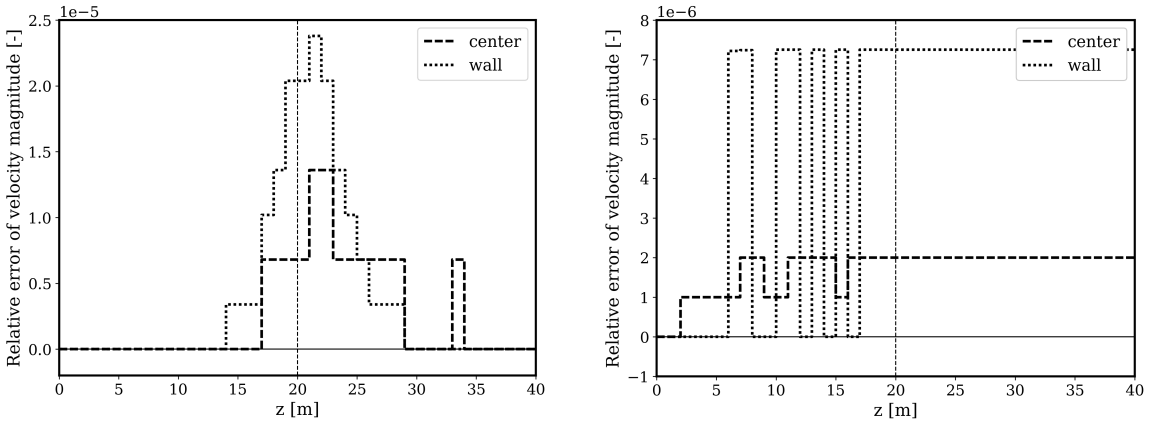


Figure 4.5: Velocity graphs of the isolated right half and the coupled scenario with `fixedValue` velocity and `fixedFluxExtrapolatedPressure` (ffep) pressure boundary conditions at the coupling inlet. The isolated simulation of the right half produces a slightly larger error because the boundary values for velocity are entered manually, whereas the coupled simulation self-corrects the velocity values.

4.2.5 The coupled result

The coupled simulation with the new coupling inlet boundary conditions converges to a solution that is almost exactly the same as the monolithic solution. Figure 4.5 shows the result, and it can be seen that the error from the isolated right half is further reduced in the coupled scenario. Figure 4.6a shows the relative error, as compared with the monolithic solution, along the pipe's center as well as close to the wall. It can be seen that the relative error is almost twice as large in the cells next to the no-slip walls. However, the maximum relative error is still very small with approximately 2.4×10^{-5}.

In the beginning of this chapter, I simplify the original pipe problem by downscaling to a coarser two-dimensional mesh and switching the solver from `pimpleFoam` to `icoFoam`. In the end, I also run the coupled simulation on the original pipe problem with the `fixedFluxExtrapolatedPressure` boundary condition. The resulting error graphs for the three-dimensional pipe are shown in Figure 4.6b. The relative errors are in the order of 10^{-6} and the discrete values are a good indicator that the resulting errors are purely numerical.



(a) Relative coupling errors for the velocity in the two-dimensional pipe.

(b) Relative coupling errors for the velocity in the three-dimensional pipe.

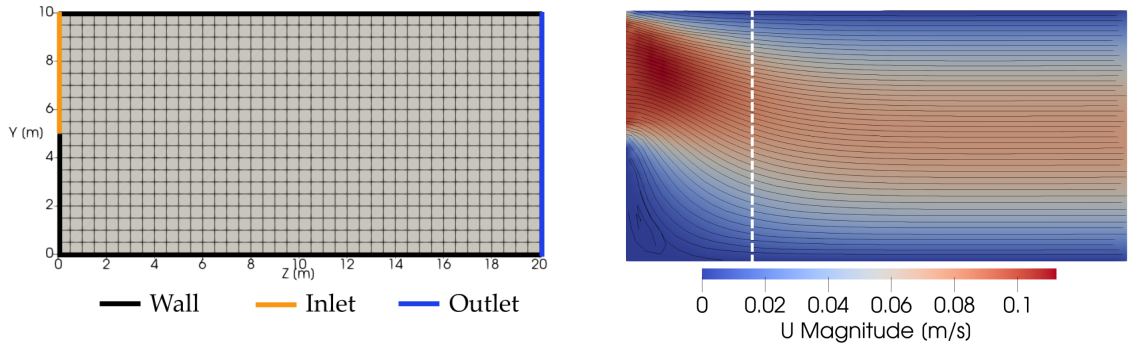
Figure 4.6: Relative error of velocity magnitude for the coupled pipe cases with the `fixedFluxExtrapolatedPressure` boundary condition and communicated values of U and p. The graphs are sampled along the z-axis in the center of the pipe and in the cells next to the pipe wall.

4.2.6 Conclusion

As a result of the first part of the investigation on fluid-fluid coupling, we can successfully couple the laminar flow through a pipe, independent of the mesh size. This is done by setting the `fixedFluxExtrapolatedPressure` (ffep) boundary condition at the coupling inlet and communicate velocity and pressure values via preCICE. The ffep boundary condition ensures flux consistency across the interface by ensuring that the momentum Equation 4.1 is fulfilled at the coupling inlet. Due to the implementation of OpenFOAM's solvers this is not the case, when a non-zero gradient is set at an inlet patch.

4.3 The half-inlet pipe

In Section 4.2, the scenario of a laminar flow through a pipe is partitioned successfully by adapting the coupling inlet pressure boundary condition. With that, the non-zero pressure gradient at the coupling interface is respected and the correctness of the momentum equation at the boundary is ensured. Because I couple the simulation across the fully developed pipe flow, the velocity gradient does not need to be taken care of, because it is naturally zero at the coupling interface. Therefore, the `zeroGradient` boundary condition at the outlet is already physically correct. As a next step, we look at a case which has a non-zero velocity gradient across the coupling interface. I introduce a pipe whose inlet covers only a part of the left boundary. Figure 4.7 shows the mesh and dimensions of the half-inlet pipe



(a) Mesh and dimensions of the half-inlet pipe case.

(b) Velocity field with streamlines

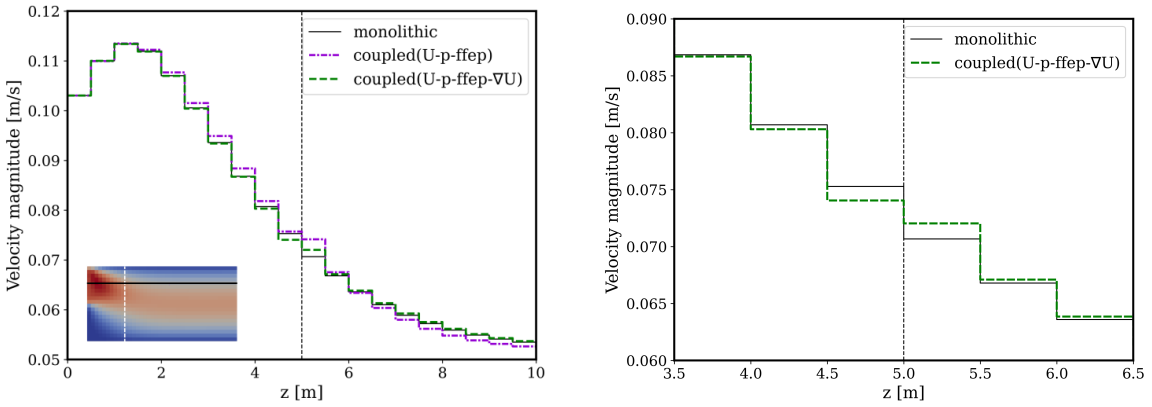
Figure 4.7: The two-dimensional half-inlet pipe case

test case. The fluid has a viscosity of $\nu = 10\mathrm{m}^2/\mathrm{s}$ and an inlet velocity of $U_{in} = 0.1\mathrm{m}/\mathrm{s}$. Because the inlet is smaller than the outlet, we can expect a decreasing velocity following the flow streamlines (see Figure 4.7b). The coupling interface is placed at $z = 5\mathrm{m}$, close enough to the inlet to have a significant velocity gradient, but far enough away from any potential vortices that might emerge in the cavity beneath the inlet.

I run the case as a coupled simulation with the coupling setup from the standard pipe Section 4.2. At the coupling inlet the boundary conditions are set to `fixedValue` for velocity and `fixedFluxExtrapolatedPressure` for pressure. At the coupling outlet I use `zeroGradient` for velocity together with `fixedValue` for pressure. preCICE communicates U and p. The violet velocity graph in Figure 4.8a shows that, as expected, we do not get a correctly converged solution with this setup. The non-physical `zeroGradient` boundary condition at the coupling outlet causes a far-reaching error in the velocity curve. Logically, I add ∇U to the communicated quantities in preCICE and send the values from the coupling inlet stream upwards to the `fixedValue` boundary condition at the coupling outlet. This configuration will be called "ffep" coupling. The ffep coupling, gives an overall good result as shown by the green graph in Figure 4.8, though still, an error is introduced in the cells close to the coupling interface.

4.3.1 The left half

Again, I look at both coupled participants individually and try to get a correct solution for both halves, without any interference by preCICE. The left half of the scenario is simulated with the reference values for pressure and velocity gradient that are obtained from the converged solution as it was shown in Subsection 4.2.2. As it can be seen from the graph in Figure 4.9a, the simulation of the left half does not converge correctly, which implies a problem with the provided boundary conditions at the outlet. From the previous



(a) Comparison of coupling with and without velocity gradient.

(b) Detailed graph around the interface for coupling with velocity gradient.

Figure 4.8: a) Velocity graphs for two coupling configurations of the half-inlet pipe. Both setups use a `fixedValue` velocity and `fixedFluxExtrapolatedPressure` (ffep) boundary conditions at the coupling inlet. At the coupling outlet is a `fixedValue` pressure boundary condition with a `zeroGradient` (violet) or `fixedValue` (green) velocity boundary condition. b) Zoomed in velocity graph of the ffep coupling including velocity gradient, showing the typical error characteristic around the interface.

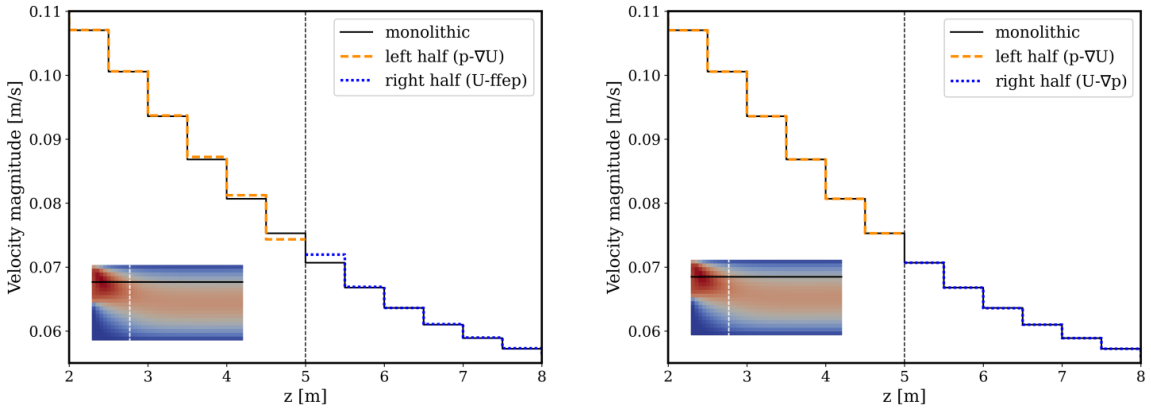
standard pipe scenario we know that we should make sure the momentum Equation 4.1 is fulfilled at the boundaries. Usually, we assume a fully developed flow at the outlet of our domain, which means that the velocity does not change along the flow direction, which is the equivalent of a `zeroGradient` condition. Furthermore, in a fully developed flow the other terms belonging to the momentum equation (such as H/A and $1/A$) do not change either. The intermediate fields `HbyA` and `rAU`, which correspond to the mentioned terms, are initialized with `zeroGradient` boundary conditions. Therefore, assuming a fully developed flow at the outlet, no further adjustments of their boundary fields are necessary.

However, we do not always have a fully developed flow at the coupling outlet and this means the `zeroGradient` boundary condition for the intermediate fields from the momentum equation is not correct. Looking through the source code of OpenFOAM's `icoFoam` solver, there seems to be no adequate boundary condition that leads to an adjustment of the boundary field values for any terms involved in the momentum equation. This means that we cannot constrain the momentum equation at an outflow patch where the velocity gradient is non-zero. Therefore, I violate the idea of preCICE being a non-invasive coupling tool and modify the function `constrainHbyA()` to set the `HbyA` boundary values at the coupling outlet according to Equation 4.3. Lines 26 to 32 of Listing 4.3 show the modified code in OpenFOAM, that calculates the boundary values for `HbyA` at the coupling outlet patch.

Running the simulation for the isolated left half with the `HbyA` correction at the coupling outlet returns the correct solution when compared to the monolithic results (see Figure 4.9b).

4.3.2 The right half

For the standard pipe scenario we established the `fixedValue` velocity boundary condition and the `fixedFluxExtrapolatedPressure` pressure boundary condition for the coupling inlet. Figure 4.9a shows that simulating only the right half of the new half-inlet pipe case leads to a noticeable error in the cells next to the inlet. Comparing the pressure gradients that are calculated by the `constrainPressure()` function as explained in Subsection 4.2.4 with the pressure gradients at the same location in the monolithic solution confirms the problem. OpenFOAM calculates the pressure gradient according to Equation 4.5. However, we do not know the real boundary values of the terms H/A and $1/A$, but instead these are just the wrongly extrapolated values from the adjacent cell centers. As an alternative, I try to copy the approach from the coupling outlet and also adjust the correction of `HbyA` at the coupling inlet to the full momentum equation. This implies that the pressure boundary condition needs to be switched back to `fixedGradient`. Simulating the right half with the code changes in Listing 4.3 again produces correct results by the look of it.



(a) Left half with fixed pressure and velocity gradient at the outlet. Right half with fixed velocity and ffep at the inlet.

(b) Both halves are simulated with the modified `constrainHbyA()` function from Listing 4.3.

Figure 4.9: Velocity graphs of individually simulated halves of the half-inlet pipe scenario. The boundary condition values are taken from the full monolithic solution.

```
volVectorField::Boundary& HbyAbf = HbyA.boundaryFieldRef();
volVectorField gradpbyA = rAU*fvc::grad(p);
volVectorField::Boundary& gradpbyAbf =
    gradpbyA.boundaryFieldRef();

forAll(U.boundaryField(), patchi) {
    if (
        !U.boundaryField()[patchi].assignable()
    && !isA<fixedFluxExtrapolatedPressureFvPatchScalarField>
        (
            p.boundaryField()[patchi]
        )
    )
    {
        if (isA<fixedGradientFvPatchScalarField>
            (p.boundaryField()[patchi]))
        {
            // At coupling inlet
            HbyAbf[patchi] = U.boundaryField()[patchi]
                           + gradpbyAbf[patchi];
        }
        else {
            HbyAbf[patchi] = U.boundaryField()[patchi];
        }
    }
    if (isA<fixedGradientFvPatchVectorField>
        (U.boundaryField()[patchi]))
    {
        // At coupling outlet
        HbyAbf[patchi] = U.boundaryField()[patchi]
                       + gradpbyAbf[patchi];
    }
}
```

Listing 4.3: `FOAM_SRC/finiteVolume/general/constrainHbyA/constrainHbyA.C` - Modified
Modified `constrainHbyA()` function that updates the boundary values of `HbyA` at the coupling inlet and coupling outlet according to the full momentum equation.

4.3.3 The coupled result

Finally, I run a coupled simulation with the aforementioned code changes and use preCICE to exchange all four quantities: U, p, ∇U, ∇p. The coupling scheme options used in `precice-config.xml` are equivalent to those presented in Listing 4.4, including the commented out pressure gradient lines. As the OpenFOAM source code had to be modified, this coupling configuration will be called "invasive" coupling. The invasive simulation converges almost perfectly to the monolithic solution as shown in Figure 4.10.

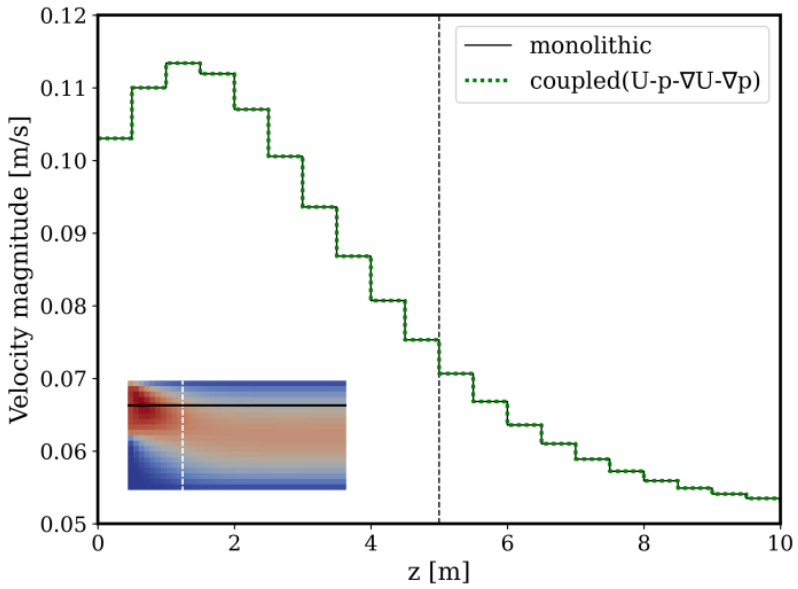


Figure 4.10: Velocity graph of the perfectly converged coupled simulation for the half-inlet pipe. preCICE communicates all quantities $U, p, \nabla U, \nabla p$ and the interface values of the field `HbyA` are adjusted to fulfil the momentum equation.

The first results of the invasive coupling approach are very promising. However, further tests reveal that this configuration does not always perform this good. For example, if the fluid viscosity is decreased, we find a significant error at the coupling interface. This phenomenon is shown in Figure 4.11.

Changing the solver to `pimpleFoam` also limits the improvement obtained by the invasive approach. Figure 4.12 shows the mean relative error of velocity magnitude over time for both `icoFoam` and `pimpleFoam`. In Figure 4.12a we can see how the invasive coupling error keeps decreasing until it reaches around 5×10^{-5} at the end of the simulation at $t = 2$s. This error is significantly lower than what is achieved with the ffep coupling, which is around 4×10^{-3}. Figure 4.12b shows the error graphs when trying those cases with the `pimpleFoam` solver. As with `icoFoam`, the invasive coupling gives improved results compared to the ffep coupling. This time however, the invasive coupling error settles

```
<coupling-scheme:serial-implicit>
  <time-window-size value="0.01" />
  <max-time value="2.0" />
  <participants first="Fluid1" second="Fluid2" />
  <exchange data="Pressure"  />
  <!--exchange data="PressureGradient"  /-->
  <exchange data="Velocity"  />
  <exchange data="VelocityGradient"  />
  <max-iterations value="25" />
  <relative-convergence-measure
    limit="1.0e-5" data="Pressure" mesh="Fluid2-Mesh" />
  <!--relative-convergence-measure
    limit="1.0e-5" data="PressureGradient" mesh="Fluid2-Mesh" /-->
  <relative-convergence-measure
    limit="1.0e-5" data="Velocity" mesh="Fluid1-Mesh" />
  <relative-convergence-measure
    limit="1.0e-5" data="VelocityGradient" mesh="Fluid2-Mesh" />
  <acceleration:IQN-ILS>
    <data mesh="Fluid2-Mesh" name="Pressure" />
    <data mesh="Fluid2-Mesh" name="VelocityGradient" />
    <initial-relaxation value="0.01" />
    <max-used-iterations value="10" />
    <time-windows-reused value="2" />
    <filter type="QR1" limit="1e-7" />
  </acceleration:IQN-ILS>
</coupling-scheme:serial-implicit>
```

Listing 4.4: Simplified excerpt of `precice-config.xml` for the half-inlet pipe case. Configuration corresponds to the ffep coupling configuration. (fixedFluxExtrapolatedPressure boundary condition at coupling inlet). Configuration including `PressureGradient` lines corresponds to the invasive coupling option.

at around 3×10^{-3} compared to 5×10^{-3} for the ffep coupling, and with that gives only a slight improvement.

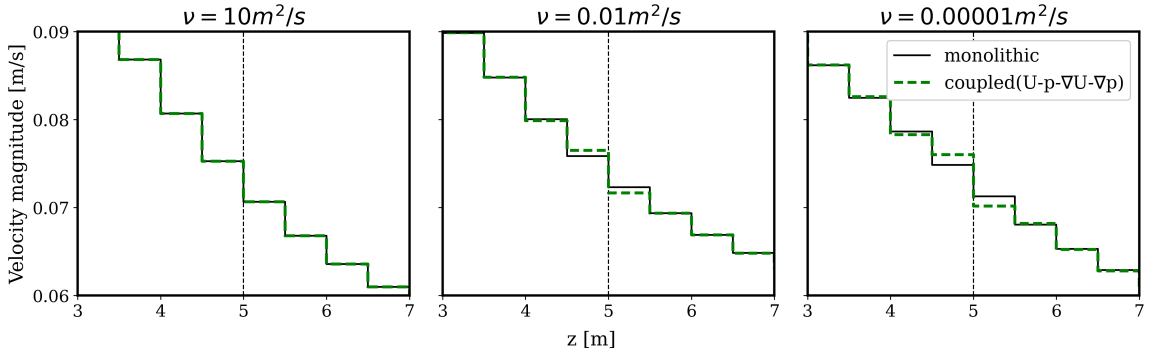
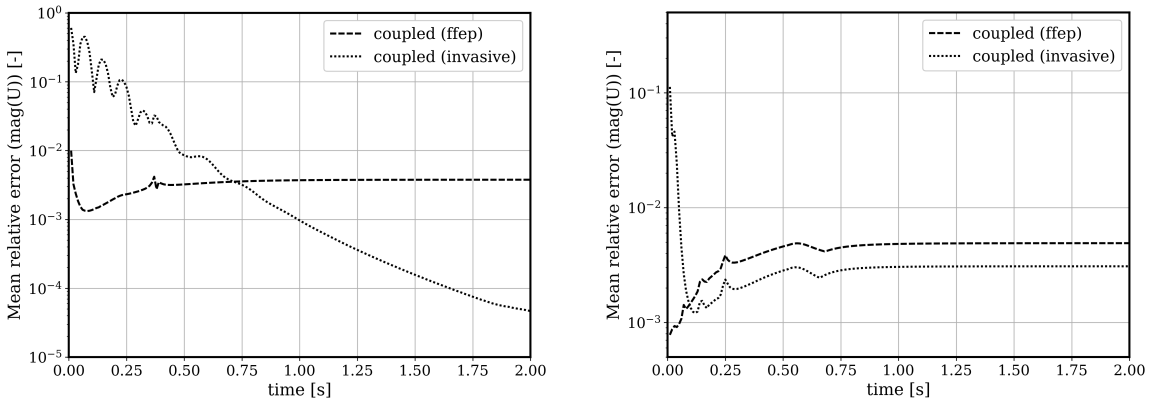


Figure 4.11: Coupled half-inlet pipe case with different viscosity values. For lower viscosity values the error at the coupling interface increases as the flow becomes more convective.



(a) Mean relative error of velocity magnitude over time for coupled cases using `icoFoam`.

(b) Mean relative error of velocity magnitude over time for coupled cases using `pimpleFoam`.

Figure 4.12: Mean relative error of velocity magnitude over simulation time for coupled half-inlet pipe cases using a) `icoFoam` or b) `pimpleFoam`

Comparing the invasive coupling to the ffep coupling shows that it is the correct idea to constrain the involved fields to the momentum equation at the coupling boundary. It makes sense, however, that perfect results cannot be achieved for most cases, because in the calculations of the `HbyA` boundary values, I still use the wrongly extrapolated values from the field `rAU`.

Furthermore, Figure 4.12 reveals another problem of the invasive coupling approach.

The invasive coupling results are a lot worse in the first time steps, compared to the ffep coupling. In the icoFoam case, it takes 72 time steps until the invasive approach is superior at $t = 0.72\text{s}$, in the case of `pimpleFoam` still 10 time steps. This can prove problematic for highly transient simulations where we want our coupled simulation to be converged at all time steps immediately.

Table 4.1 gives a comparison of total iterations and mean errors at the end of the simulation time $t = 2\text{s}$ for the invasive and ffep coupling variants. We can see that the reduced errors for the invasive variants come with the cost of more than double the amount of implicit iterations. The drawbacks of the invasive coupling method make its usefulness questionable. Therefore, Table 4.1 shows two additional configurations that can reduce the error of the ffep coupling scheme. The first option decreases the discretization error at the interface by halving the grid size in z-direction ($\Delta z_{1/2} = 0.5\Delta z$). The other option is having a lower velocity gradient $\nabla U_{1/2}$ at the coupling interface. By modifying the half-inlet pipe case such that the inlet-outlet ratio is increased from 0.5 to 0.75, the velocity gradient at the interface is approximately halved. Figure 4.13 shows the effect on the relative error of velocity. Both, grid resolution and velocity gradient seem to be linearly correlated to the error. From Figure 4.13 it can also be seen that the largest errors for the ffep coupling usually appear in the cells next to the interface. The interface adjacent values are closer together than their corresponding monolithic cell values as it is seen in Figure 4.8b. This lets us define the expected maximum absolute coupling error as:

$$\epsilon(\boldsymbol{U}) < |\boldsymbol{U_N} - \boldsymbol{U_P}|, \tag{4.6}$$

where U_N and U_P are the cell values on both sides of the interface. In cells further away from the interface, the error quickly reduces by one order in magnitude.

solver	coupling configuration	iterations	error ($\lvert\boldsymbol{U}\rvert$)	error (p)
`icoFoam`	ffep	1211	3.770×10^{-3}	2.368×10^{-3}
`icoFoam`	invasive	2442	4.655×10^{-5}	5.991×10^{-5}
`pimpleFoam`	ffep	1164	4.898×10^{-3}	2.509×10^{-3}
`pimpleFoam`	invasive	2751	3.081×10^{-3}	1.842×10^{-3}
`icoFoam`	ffep (fine mesh, $\Delta z_{1/2}$)	1532	1.228×10^{-3}	9.813×10^{-4}
`icoFoam`	ffep (small $\nabla U_{1/2}$)	1130	1.116×10^{-3}	4.979×10^{-4}

Table 4.1: Statistics for various coupled simulation for the half-inlet pipe scenarios. Shown are the total number of implicit preCICE iterations, the mean relative error for velocity magnitude and pressure compared to the monolithic solution at $t = 2.0\text{s}$. Simulation time step size is $\Delta t = 0.01\text{s}$ and maximum amount of iterations per preCICE time window is 25.

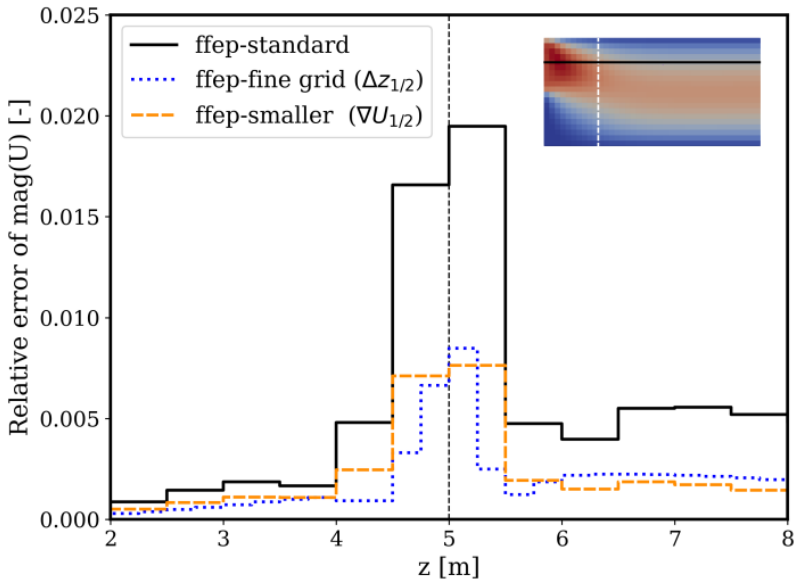


Figure 4.13: Relative velocity errors for coupling of the half-inlet pipe with the ffep setup. Comparison of the standard mesh to a refined mesh and a modified geometry with lower velocity gradient at the interface.

4.3.4 Conclusion

The previously described modifications of `constrainHbyA()` greatly improved the converged coupled solution in a very restricted scenario. The results showed that I was looking into the right direction by forcing the fulfillment of the momentum equation at the coupling interface. Overall, I do not think that it is worth to abandon the non-invasive concept of preCICE for the improved coupling accuracy on account of the mentioned disadvantages, which are: High number of implicit coupling iterations and bad performance in the beginning of the simulation.

The instability of the here presented solution is too big to make this approach practical. Ultimately, it appears that we lack too much information at the coupling boundary which prevents us from implementing a perfect fluid coupling solution over a common coupling interface.

Therefore, I decide to proceed with the ffep solution, which produced stable results with a mean error in the order of 10^{-3}, The ffep coupling configuration consists of the `fixedFluxExtrapolatedPressure` boundary condition at the coupling inlet and communication of U, p and ∇p via preCICE. The obtained error is usually the largest at the interface, but can be reduced by mesh refinement at the interface.

4.4 The irregularly widening pipe

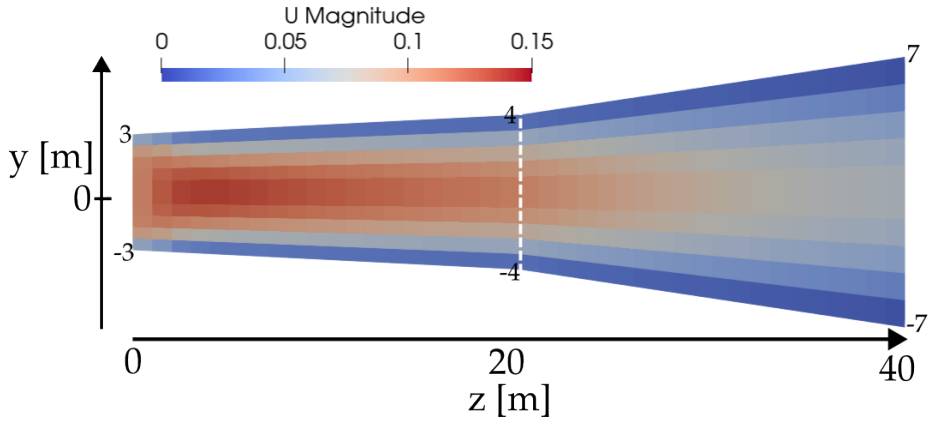


Figure 4.14: Mesh and dimensions of the two-dimensional widening pipe.

The last scenario that shall be presented in the context of this investigation chapter, is an irregularly widening pipe. The fluid flows through a two-dimensional pipe that steadily opens up. The number of cells in y-direction stay the same throughout, which leads to a non-orthogonality of the cells as can be seen in Figure 4.14. The opening angle of the pipe increases at the center of the pipe, where I place the coupling interface. This mesh setup results in different magnitudes of non-orthogonality on both sides of the coupling interface. The impact on the coupling results of this mesh will be investigated during this chapter. The steadily increasing cross-area of the pipe leads to decreasing velocity along the z-axis, which is shown in the sampled graph in Figure 4.15.

I run this case in a coupled setup by splitting the domain at the center of the z-axis. Like in the previous chapter, there is a non-zero velocity gradient across the interface. Additionally, the velocity decreases faster on the right side of the interface than to its left due to the increased opening angle. Furthermore, this leads to the adjacent cells having a different magnitude of non-orthogonality, which adds to the complexity of this scenario.

4.4.1 Flux correction

Figure 4.15 also shows the velocity graph (green) with the recommended setup, which uses the `fixedFluxExtrapolatedPressure` boundary condition at the coupling inlet and exchanges U, p, and ∇p with preCICE. The coupling at the interface introduces an error that propagates downstream and eventually leads to an unphysical decrease of the mass flow rate at the outlet. This error is introduced by overestimating the velocity gradient magnitude, which is taken from the coupling inlet to the coupling outlet. Because the mesh geometry continues differently on the right side of the coupling interface, we have trouble estimating the right values at the interface. Finding the correctly interpolated values at the

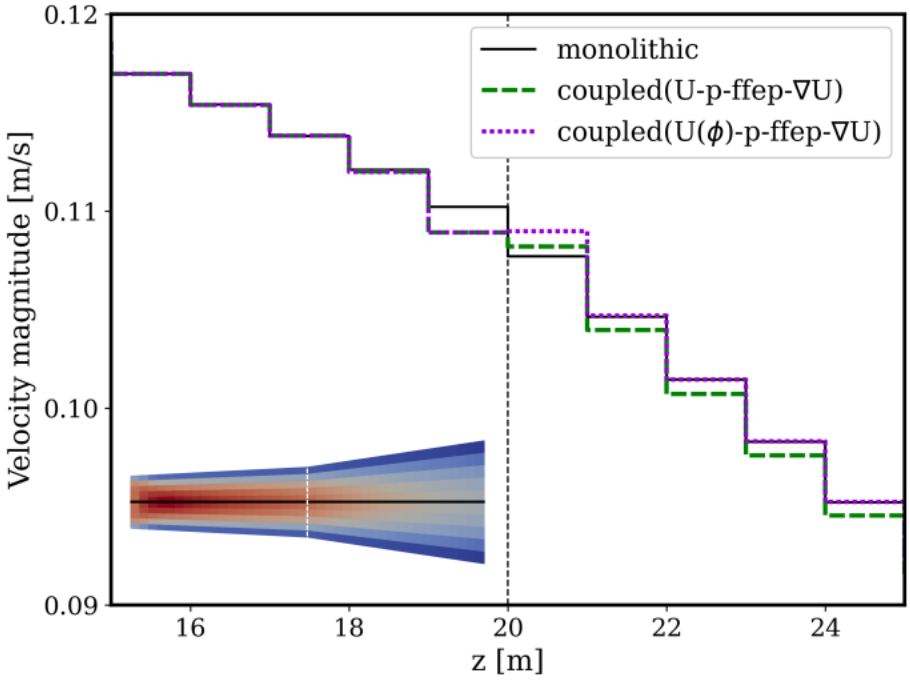


Figure 4.15: Velocity graphs for the widening pipe case with `fixedValue` velocity and `fixedFluxExtrapolatedPressure` boundary conditions at the coupling inlet and `fixedGradient` velocity and `fixedValue` pressure boundary conditions at the coupling outlet. The violet graph shows the result, when the communicated velocity is corrected by the mass flux.

interface is, of course, a common problem for interface coupling procedures. In the case of flow coupling however, it can prove especially troublesome if the interface interpolation introduces a non-physical source or sink. In a closed pipe system for example, such an error can cumulate and increase or decrease the velocities until the simulation breaks.

Therefore, I adjust the velocity that is sent from the coupling inlet to the coupling outlet by a face flux correction. The formula for flux correction is taken from the OpenFOAM boundary condition `fluxCorrectedVelocity`[1]:

$$\boldsymbol{U}_{\text{new}} = \boldsymbol{U}_{\text{bf}} - \boldsymbol{n}(\boldsymbol{n} \cdot \boldsymbol{U}_{\text{bf}}) + \frac{\boldsymbol{n}\phi_{\text{bf}}}{|\boldsymbol{S}_{\text{f}}|} \tag{4.7}$$

$\boldsymbol{U}_{\text{bf}}$ is in this case the wrongly calculated velocity at the boundary, $\boldsymbol{n}$ are the patch normal vectors, $|\boldsymbol{S}_{\text{f}}|$ the face area, and ϕ_{bf} the boundary flux values that are obtained from the

[1] OpenFOAM API Guide v2112, fluxCorrectedVelocity class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1fluxCorrectedVelocityFvPatchVectorField.html

field `phi`. First, the velocity component orthogonal to the boundary face is subtracted and then, the flux divided by the face area is added. Listing 4.5 shows the code implementation of Equation 4.7. Correcting the velocity values with this formula before sending them to the coupling inlet assures that mass is conserved and that the mass flux is the same in the right domain. Figure 4.15 shows the result for this implementation. The maximum error in the cells next to the interface is increased, but further away from the coupling boundary, the results match the monolithic solution very closely and most importantly, the mass is conserved.

4.4.2 Corrected Laplacian scheme

When working on non-orthogonal meshes, it is recommended to adjust the numerical method for `snGradSchemes` inside the `fvSchemes` dictionary. The `snGradSchemes` determines how to calculate the gradient of a quantity at the face centers and is mainly used as part of the laplacian schemes.

The basic laplacian scheme `Gauss linear uncorrected` assumes the face f between two cells to be perpendicular to the delta vector $\boldsymbol{\Delta}$ connecting the adjacent cell centers P and N and then calculates the face normal gradient $(\nabla_\mathrm{n}\phi)_\mathrm{f}$ by using finite differences. This is the first term in Equation 4.8:

$$(\nabla_\mathrm{n}\phi)_\mathrm{f} = \frac{\phi_\mathrm{N} - \phi_\mathrm{P}}{\boldsymbol{n} \cdot \boldsymbol{\Delta}} + \boldsymbol{k} \cdot (\nabla\phi)_\mathrm{f} \tag{4.8}$$

When specifying the scheme `Gauss linear corrected` instead, OpenFOAM adds a non-orthogonal correction term $\boldsymbol{k} \cdot (\nabla\phi)_\mathrm{f}$ to the face gradient calculation. The correction vector $\boldsymbol{k}$, which is needed in the second term of Equation 4.8 is calculated as

$$\boldsymbol{k} = \boldsymbol{n} - \frac{\boldsymbol{\Delta}}{\boldsymbol{n} \cdot \boldsymbol{\Delta}} \tag{4.9}$$

where $\boldsymbol{\Delta}$ is simply the distance vector between the adjacent cell centers: $\boldsymbol{\Delta} = \boldsymbol{C_\mathrm{N}} - \boldsymbol{C_\mathrm{P}}$ and $\boldsymbol{n}$ the normalized face vector $\boldsymbol{S}$. Figure Figure 4.16 gives a visualization of the involved vectors.

Inherently, there are no neighboring cells next to the boundary faces of the mesh. Therefore, the distance vector $\boldsymbol{\Delta}$ and consequently $\boldsymbol{k_\mathrm{bf}}$ cannot be calculated properly. This problem is resolved by simply setting the correction vectors to zero and thereby does not apply any non-orthogonal corrections at the boundary faces, including the coupling interface.

In the current implementation state, this error is overshadowed by larger error and has only a minimal impact. Comparing the mean relative error of velocity magnitude and pressure of the coupled simulations to the corresponding monolithic ones confirms that there is an impact (see Table 4.2). While this is only a minor error source compared to the previously described problems, it should be kept in mind, when a perfect coupling solution should be required. This might be resolved by exchanging additional geometric

```
1  void preciceAdapter::FF::Velocity::write(double* buffer)
2  {
3      int bufferIndex = 0;
4      // For every boundary patch of the interface
5      for (uint j = 0; j < patchIDs_.size(); j++)
6      {
7          int patchID = patchIDs_.at(j);
8
9          vectorField Up = U_->boundaryFieldRef()[patchID];
10         scalarField phip = phi_->boundaryFieldRef()[patchID];
11         vectorField n =
12             U_->boundaryField()[patchID].patch().nf();
13         scalarField magS =
14             U_->boundaryField()[patchID].patch().magSf();
15         // Calculate flux corrected velocity values
16         vectorField U_corrected =
17             Up - n*(n & Up) + n*phip/magS;
18
19         // For every cell of the patch
20         forAll(U_->boundaryFieldRef()[patchID], i)
21         {
22             // x-dimension
23             buffer[bufferIndex++] =
24                 U_corrected[i].x();
25
26             // y-dimension
27             buffer[bufferIndex++] =
28                 U_corrected[i].y();
29
30             // z-dimension
31             buffer[bufferIndex++] =
32                 U_corrected[i].z();
33         }
34     }
35 }
```

Listing 4.5: `OpenFOAM-Adapter/FF/Velocity.C` - Velocity::write()
The velocity value is corrected by the face flux before writing them to the preCICE buffer.

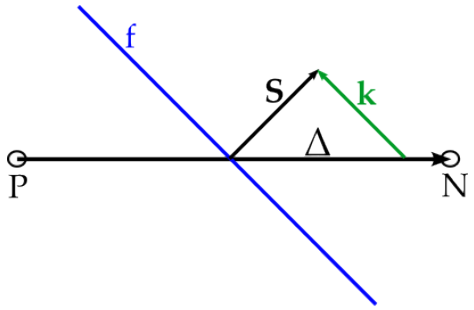


Figure 4.16: Visualization for the terms involved at the corrected gradient calculation. The face f is not perpendicular to the vector Δ that connects the neighboring cell centers P and N

data, such as the cell center coordinates, between the coupling participants.

laplacian scheme	uncorrected	corrected
mean relative error mag(U)	8.11×10^{-4}	9.40×10^{-4}
mean relative error p	9.51×10^{-4}	1.14×10^{-3}

Table 4.2: The mean relative error of velocity magnitude over all cells. The coupled simulation using uncorrected snGradSchemes is compared to the monolithic solution using the uncorrected snGradSchemes. The coupled simulation with corrected snGradSchemes is compared to the monolithic solution using corrected snGradSchemes.

4.5 The domain decomposition approach

What has not been utilized up until now, is the fact that OpenFOAM itself already has coupled boundary conditions. Looking at the boundary condition classes of OpenFOAM, we can find the class `coupledFvPatchField`[2]. This class implements a method called `patchNeighbourField()` which returns a field containing the cell values that are next to the boundary "on the other side" of the coupled boundary. This means a coupled boundary does not only have information of values at the boundary faces but also to the values in the cells next to it. With this approach all quantities can be interpolated properly onto the boundary faces. This method is often used in domain decomposition problems

[2]OpenFOAM API Guide v2112, coupledFvPatchField class reference:
https://www.openfoam.com/documentation/guides/v2112/api/classFoam_1_1coupledFvPatchField.html

and the `patchNeighbourField` is the equivalent of a halo layer that is added next to the internal subdomain which is visualized in Figure 4.17.

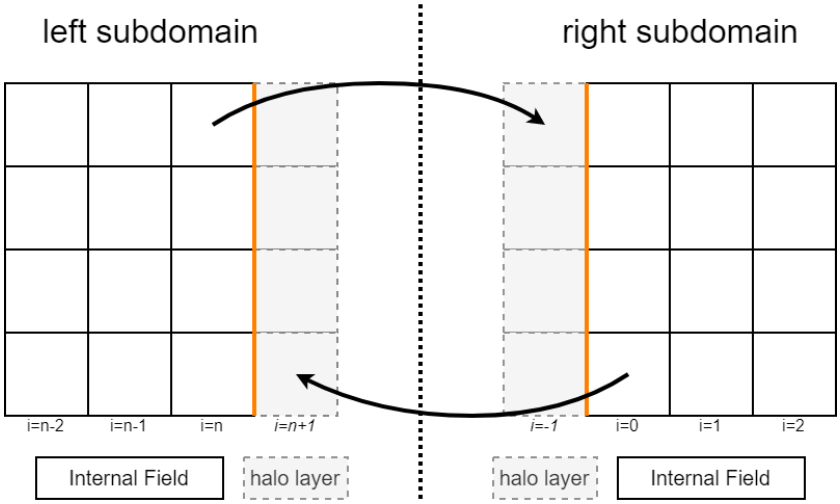


Figure 4.17: Domain decomposition with halo layer cells

The `coupledFvPatchField` class itself is an abstract class and therefore cannot be used as a boundary condition. The main inheriting classes are `cyclicFvPatchField` and `processorFvPatchField`. The cyclic patch field is used to implement periodic boundary conditions. Periodic boundary conditions are employed if a domain is approximated as infinite and steady in one or more directions. For example, when modeling a very long pipe or the ocean surface. The idea behind periodic boundary conditions is that the outflow from one patch is then taken as the inflow into the other patch. To do so, this boundary condition requires the user to set two corresponding patches on the same mesh. When we do flow partitioning with preCICE, we need a slightly different method for our partitioned flow.

In contrast, the processor boundary condition is usually not set by the users directly but is created automatically by `decomposePar`, when running OpenFOAM simulations in parallel. OpenFOAM uses a domain decomposition approach with MPI to distribute the work to multiple processors. The first step is to decompose the mesh and the initial fields into approximately equally sized subdomains using `decomposePar`. This creates multiple subfolders in the main case folder such that file input and output operations can also be done in parallel. Afterwards, the desired solver is launched with `mpirun`, which creates multiple solver instances and each processor solves the flow problem on only its part of the mesh. At boundaries where the full domain was cut into smaller subdomains, processor boundary conditions are assigned. The halo layer values are acquired by sending and receiving the boundary cell values to the corresponding processors via MPI. At the end of the simulation, there are multiple folders for each processor with the time directories and

the solutions for each subdomain. These can be combined again by using OpenFOAM's `reconstructPar` utility or opened separately in post-processing tools such as ParaView.

Flow partitioning with preCICE is essentially very similar to running parallel simulations in OpenFOAM. In the same way as OpenFOAM uses domain decomposition for parallelization, we divide the full fluid domain into two or more subdomains which we call participants. Instead of using MPI calls directly, we call the preCICE library to communicate values between the coupled boundaries. This raises the following question: Why do we not copy the domain decomposition methods and boundary conditions used by OpenFOAM and utilize them for our preCICE partitioning?

4.5.1 A custom coupled boundary condition

The first step would be to create our own custom coupled boundary condition that inherits from `coupledFvPatchField` or imitates its functionality. This boundary condition needs the buffer to hold the values from the halo layer which will be provided by the neighboring preCICE participant. Additionally, the evaluating and coefficients functions need to be modified such that the face values are interpolated by the internal boundary cells and the halo layer cells. In that way, we can create a custom coupled boundary condition for the main flow variables such as pressure and velocity. As previously explained, OpenFOAM does not only operate on the main flow variable fields but also on intermediate fields such as `HbyA` or `rAU`, which are used in the SIMPLE / PISO solution process. Those fields are initialized with `extrapolatedCalculatedFvPatchField` boundary conditions everywhere. This boundary condition simply applies a zero Gradient and extrapolates the cell values on to the boundary faces. This is wrong in the case of a coupled boundary, where we want the interpolated value of the adjacent cells on the boundary face. That's why OpenFOAM cycles over the boundaries after field initialization and compares the boundary conditions with the underlying patch types. Coupled boundary conditions in OpenFOAM are always used in conjunction with the corresponding patch types that are defined in the `polyMesh` directory. Therefore, cyclic boundary conditions need to be combined with cyclic patches and processor boundaries are combined with processor patches. When OpenFOAM creates processor boundaries during the domain decomposition, it has to set the processor boundary conditions in the fields files but additionally the processor boundary types in the `polyMesh/boundary` file. Coming back to the initialization of the field `HbyA`, OpenFOAM will recognize an incompatibility of the underlying processor patch and the extrapolatedCalculated boundary condition and will automatically exchange the boundary condition on this patch to the matching processor boundary condition. This ensures that the temporary field `HbyA` has coupled boundary conditions at the coupled patches and extrapolatedCalculated boundary conditions at all non-coupled patches. As a result, this poses the first complication to our custom coupled boundary condition. Additionally to the custom `fvPatchField` class, we need to provide matching coupled `fvPatch` and `fvPolyPatch` classes. Only then, OpenFOAM will also set our coupled boundary condition for intermediate fields that are created by each individ-

ual solver.

Assuming we can incorporate such custom boundary conditions with corresponding patch types, we would soon reach another problem, which is the communication frequency. The preCICE OpenFOAM adapter is a function object and as such is called only at certain points of the solver's algorithm. Importantly, this is only once per time step at most. Neighboring field values that are needed by our coupled boundary condition would be updated only at the end of each time step. Problematically, intermediate fields such as `HbyA` do no longer exist at the end of the solver's time step and therefore, we cannot access these fields' values at all. OpenFOAM's processor boundaries on the other hand are updated every time a field's `correctBoundaryConditions()` method is called. Generally speaking, this is happening every time after a field is newly created or the values have been altered. When the `HbyA` field is initialized, its `correctBoundaryConditions()` method is called which in turn calls the evaluate function of each boundary patch. The processor boundary then handles the MPI communication to send the neighboring values to these patches. To imitate this behavior in our custom preCICE-coupled boundary conditions we would have to provide communication calls to preCICE in the `evaluate()` method. Figure 4.18 shows the code flow when evaluating a field's boundary conditions with the proposed functionality of a custom coupled boundary condition.

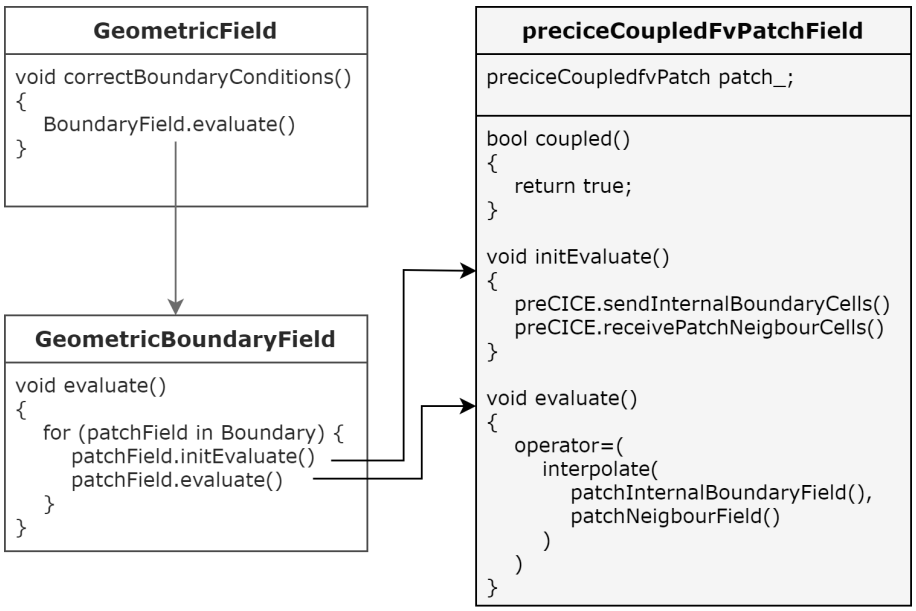


Figure 4.18: OpenFOAM boundary flow for proposed custom boundary condition

While all of this could be feasible and potentially provide the most accurate coupling interface, it is quite complex and out of scope of this thesis. Additionally, some concerns about the usability of above method should be mentioned as well. This approach can provide great results when coupling two OpenFOAM solvers, potentially from different

OpenFOAM versions. It will probably not be useful, however, when one preCICE participant is run by a different CFD program. Another program might not use the finite volume method, might not use intra-time step hooks or use any of the intermediate fields like above-mentioned `HbyA`. For these cases, the above explained construct is most likely not usable at all.

5 Coupling of advanced flows

In Chapter 4, I established a basic functionality for coupling incompressible, laminar flow scenarios, where the fluid has a consistent flow direction at the coupling interface. In Chapter 2 however, we discussed that the complete Navier-Stokes equations include more advanced flow characteristics such as the energy equation and turbulence modeling. Eventually, the goal is to extend the preCICE OpenFOAM adapter to support all standard OpenFOAM fluid solvers.

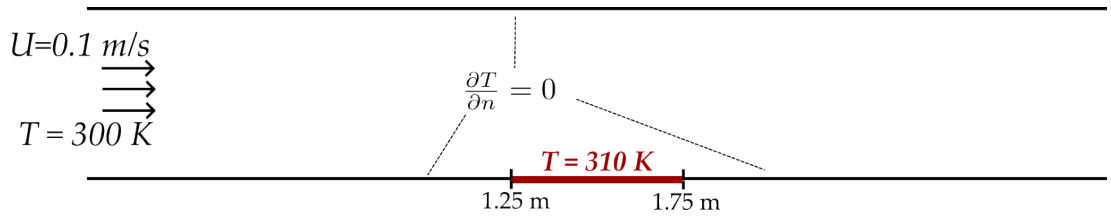
During this chapter, I will first look into adding temperature to the coupled variables, which is needed when we couple two solvers that include the energy equation, e.g. the `buoyantPimpleFoam` solver. I present the implementation and one validation case in Section 5.1. The full PIMPLE algorithm also includes the calculation of specified turbulence models (see Subsection 2.3.3). However, analysis of turbulence flows often comes with flow vortices of different sizes. Thus, before adding turbulence parameters to the coupling, I develop a custom inlet-outlet boundary condition in Section 5.2 that can be used at the coupling interface. The extension of actual turbulence will be left for development outside the context of this thesis.

5.1 Temperature coupling

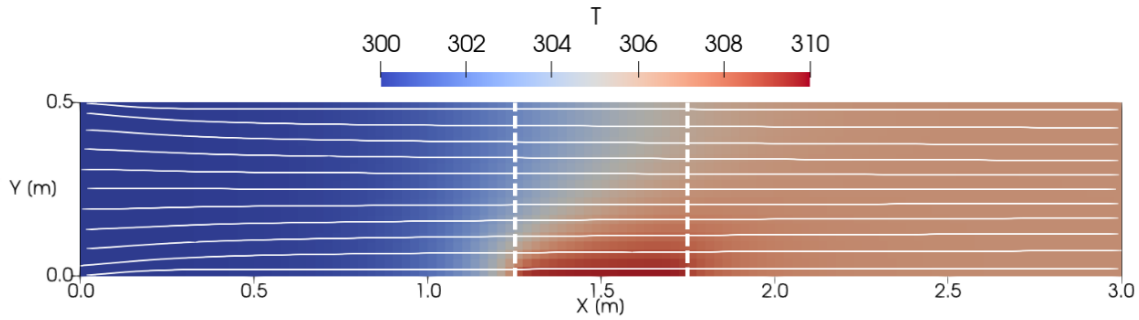
The basic formulas for heat transfer in fluids via conduction and convection are explained in Section 2.5. Based on this, I implement temperature coupling by assuring the continuity of conductive and convective heat flux across the coupling interface. Advantageous compared to general conjugate heat transfer applications, the same material is on both sides of the coupling interface. This means that the material parameters such as the thermal conductivity k, the heat transfer coefficient h and even the environment temperature T_∞ are equal on both sides of the interface. Therefore, if we set the flux equations to be equal on both sides of the interface we can divide by these material parameters. Subsequently, we are left with a classic Dirichlet-Neumann coupling approach where we have to obey the continuity of temperature T and temperature gradient ∇T at the interface.

Regarding the implementation, adding heat transfer to the fluid-fluid module of the preCICE OpenFOAM adapter is done by adding new `CouplingDataUser` classes for temperature and temperature. Both classes implement a constructor as well as `read()` and `write()` methods that are similar to the classes for pressure and pressure gradient. The default name of the temperature field is `T`.

Coupling temperature is done by specifying a `fixedValue` boundary condition on one



(a) Boundary conditions for the flow over heated wall segment scenario.



(b) Solution for temperature. Streamlines indicate the underlying flow.

Figure 5.1: Mesh and geometry of the two-dimensional pipe with a heated wall segment at the bottom between $x = 1.25\text{m}$ and $x = 1.75\text{m}$.

side of the coupling interface and a `fixedGradient` boundary condition on the other.

5.1.1 Flow over heated wall segment

I validate the temperature addition to the OpenFOAM preCICE adapter with a modified version of the "Flow over a heated plate" tutorial case for the preCICE CHT module. I do not model the solid, but instead only look at the laminar fluid flow through a pipe, which has a fixed temperature at a small section of the bottom wall.

Setup

The case geometry and the temperature boundary conditions are shown in Figure 5.1. All walls use a no-slip boundary condition for velocity and an adiabatic temperature boundary condition except for the heated wall segment, where a fixed temperature of $310K$ is prescribed. The incoming flow at the pipe inlet has a temperature of $300K$. The fluid's thermal properties are taken from the preCICE tutorial case with a Prandtl number $Pr = 0.01$ and a thermal conductivity of $k = 1\text{W}/(\text{mK})$. The fluid is modelled as a perfect gas.

I test two partitioning scenarios where I place the coupling interface downstream of the heated wall segment for the first scenario and then upstream for the second. Figure 5.1 also shows where the pipe was partitioned for both cases. By this, I test that we can couple

the heat flux independent of the flow direction. For velocity and pressure, it was clear in which way the Dirichlet and Neumann boundary conditions are set to the coupling inlet and coupling outlet. For temperature, it seems not inherently obvious in what direction the Dirichlet-Neumann coupling should take place. Therefore, I test both partitioning variants, with `fixedValue` at the coupling inlet and `fixedGradient` at the coupling inlet as well as the other way around.

The boundary conditions used for pressure and velocity correspond to the ffep coupling that I established in Section 4.3. At the coupling outlet I use `fixedValue` for pressure for velocity and `fixedGradient` and at the coupling inlet `fixedValue` for velocity and `fixedFluxExtrapolatedPressure` for pressure.

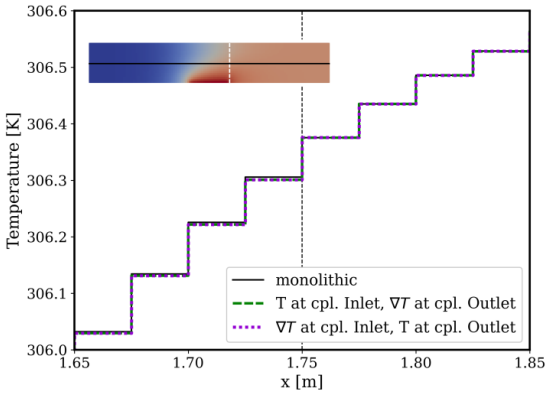
The preCICE coupling setup in `precice-config.xml` is the same as in Listing 4.4 with the addition of `Temperature` and `TemperatureGradient` as exchanged variables, both with a relative convergence measure of 1×10^{-5}. The temperature variable that is sent upstream is additionally added to the Quasi-Newton acceleration.

Results

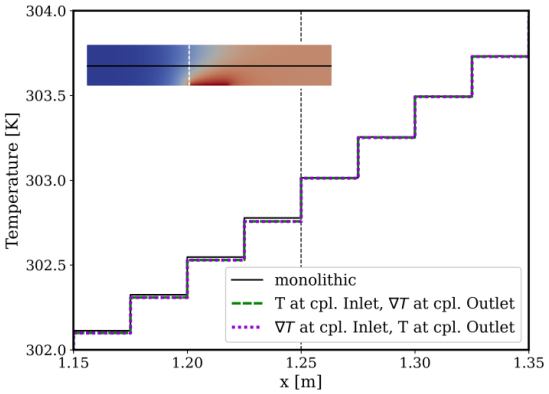
Figure 5.2 shows the temperature graphs along the center of the pipe close to the coupling interface. For both locations of the coupling interface it does not matter in which direction I do the Dirichlet-Neumann coupling. The solutions converge to almost the same results. As we can see from the total number of iterations at $t = 40$s in Table 5.1, the solutions converged faster when I had a `fixedValue` at the coupling outlet and a `fixedGradient` at the coupling inlet. However, experience shows that these iteration numbers are quite sensitive to the coupling scheme setup, which should be optimized for each particular case. From Table 5.1 we can also see that the overall solutions show a slightly lower average relative error of temperature for the setups where I place the coupling interface downstream of the heat segment. This corresponds to the error differences of velocity and because velocity and temperature are tightly coupled, it is hard to predict the errors' origin. Generally, we can assume that it is preferable to place the coupling interface somewhere where we expect a low velocity gradient. This should lead to a lower error in the general flow and therefore to the coupled quantities such as temperature which are also advected by the velocity of the flow.

5.2 Custom inlet-outlet boundary conditions

In more complex CFD simulations, one usually tries to place the inlets and outlets far away from the actual area of interest. This is done because it is easier to model the boundary conditions. At the inlet for example, we can specify a basic uniform profile that has space to develop before reaching the interesting area. At the outlet, we prefer a fully developed flow so that we do not have to worry about any potential backflow that might interact again with our region of interest. However, when we partition our domain to run



(a) Temperature graphs across the coupling interface downstream from the heated wall segment at $x = 1.75\text{m}$. Both coupling variants give the same results, which are close to the monolithic solution.



(b) Temperature graphs across the coupling interface upstream from the heated wall segment at $x = 1.25\text{m}$. Both coupling variants give the same results, which close to the monolithic solution.

Figure 5.2: Temperature graphs across the coupling interface downstream and upstream from the heated wall segment. Two temperature coupling variants are shown, once with `fixedValue` at the coupling Inlet and `fixedGradient` at the coupling Outlet (green) and once with `fixedGradient` at the coupling Inlet and `fixedValue` at the coupling Outlet (violet).

a coupled simulation with preCICE, it might be difficult to find sections where the flow is steady throughout the simulation. Instead, we have to expect recirculations across the interface, meaning that the flow direction at the coupling interface might vary from cell to cell. This is especially important, when we want to couple turbulent flow, as turbulent flow is expected to create vortices on different length scales.

So far, I only looked at unidirectional flow, where we can clearly define a coupling outlet and a coupling inlet. These definitions will no longer hold, when both coupling boundaries might act as an outlet, inlet, or both, depending on the flow characteristics. Furthermore, we can no longer predict in what direction the Dirichlet-Neumann coupling should take place for velocity and pressure. We need special boundary conditions that can act as both a Dirichlet or Neumann boundary condition, depending on the flow direction.

OpenFOAM provides such functionality in boundary conditions named `inletOutlet` and `outletInlet`. These are recommended to be used at outlets, where a backflow might be expected. They can switch between acting as a `fixedValue` and zeroGradient boundary conditions depending on the direction of the face flux phi. However, during the investigation in Chapter 4, I established that we cannot use `zeroGradient` boundary conditions for either velocity or pressure at the coupling interface. Instead, we need a `fixedFluxExtrapolatedPressure` pressure boundary conditions for inflow and a

Coupling setup	Cumulative preCICE iterations	Mean relative error (T)	Mean relative error (U)
downstream ($T \downarrow, \nabla T \uparrow$)	5346	1.895×10^{-6}	2.468×10^{-5}
downstream ($T \uparrow, \nabla T \downarrow$)	4999	1.688×10^{-6}	2.440×10^{-5}
upstream ($T \downarrow, \nabla T \uparrow$)	3063	6.754×10^{-6}	5.899×10^{-5}
upstream ($T \uparrow, \nabla T \downarrow$)	2810	6.747×10^{-6}	5.892×10^{-5}

Table 5.1: Comparison of coupling iterations and relative errors for different coupling setups at $t = 40\text{s}$ ($\Delta t = 0.1\text{s}$). The coupling setup is either downstream or upstream relative to the heated wall segment. The temperature T and temperature gradient ∇T are communicated downstream from the coupling outlet to the coupling inlet ($\downarrow$) or upstream from the coupling inlet to the coupling outlet ($\uparrow$).

`fixedGradient` velocity boundary condition for outflow. These features cannot be used when we set OpenFOAM's native `inletOutlet` and `outletInlet` boundary conditions.

5.2.1 Implementation

To combine the desired functionalities into inlet-outlet style boundary conditions, I implement custom classes for both, velocity and pressure. These boundary conditions inherit from the base class `fvPatchField` from OpenFOAM. The respective C++ and header files are placed inside the fluid-fluid module of the preCICE OpenFOAM adapter and are part of the `libpreciceAdapterFunctionObject.so` library. Because the boundary conditions need to be available before the function objects are loaded, the usage of the preCICE function object changes slightly. Listing 5.1 shows the lines that need to be added to each solver's `controlDict`.

The coupledPressure boundary condition

A custom `coupledPressure` boundary condition should behave like a `fixedValue` boundary condition where there is outflow and switch to a `fixedFluxExtrapolatedPressure` boundary condition when there is inflow. The latter gives the first restriction for the custom boundary class. As we saw in Subsection 4.2.4, OpenFOAM explicitly tests, if the pressure boundary condition is of that type in the function `constrainHbyA()` (see Listing 4.1). In order to not modify any existing OpenFOAM files, this implies that the `coupledPressureFvPatchField` class must inherit from `fixedFluxExtrapolatedPressureFvPatchScalarField`. The `outletInlet` class of OpenFOAM inherits from the mixed boundary condition, but the new class cannot do that since it already inherits

```
functions
{
    preCICE_Adapter
    {
        type preciceAdapterFunctionObject;
    }
}

libs ("libpreciceAdapterFunctionObject.so");
```

Listing 5.1: Required setup in the `controlDict` file inside the OpenFOAM fluid solver directory, if the custom inlet-outlet boundary conditions are to be used.

from another boundary class. Therefore, I copy the logic of the mixed boundary condition to the `coupledPressureFvPatchField` class. Similar to OpenFOAM's mixed boundary condition, the `refValue`, `refGrad` and `valueFraction` are stored as private class members. Listing 5.2 shows the implementation for the `evaluate()` and `snGrad()` functions which return the face value and normal gradient respectively. The `evaluate()` function calls `updateCoeffs()`, which sets the value fraction parameter according to flux direction. The `refValue` will be set by preCICE while the gradient is calculated by OpenFOAM in the `constrainPressure()` function (see Listing 4.2).

The coupledVelocity boundary condition

The `coupledVelocity` boundary condition is very similar to the `inletOutlet` class from OpenFOAM. However, I add one safety clause to the `updateCoeffs()` function, as shown in Listing 5.3. This prevents the coupled simulation from crashing when the user cannot predict the correct flow directions for the first time step. As an example, if we consider a partitioned pipe with `coupledVelocity` at the coupling inlet and the user sets `refValue` to zero, then the boundary flux will be evaluated to zero as well and the coupling inlet would be treated wrongly as an outlet. Listing 5.4 shows the necessary entries for the initial `p` and `U` files, where the coupled boundaries need to be set.

```
void Foam::coupledPressureFvPatchField::updateCoeffs() {
    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>(phiName_);
    const fvsPatchField<scalar>& phip =
        patch().patchField<surfaceScalarField, scalar>(phi);

    this->valueFraction() = pos0(phip);
    fixedFluxExtrapolatedPressureFvPatchScalarField::updateCoeffs();
}

void Foam::coupledPressureFvPatchField::evaluate() {
    updateCoeffs();

    scalarField::operator=
    (
        valueFraction_*refValue_
        + (1.0 - valueFraction_)
        *(
            this->patchInternalField()
            + refGrad_/this->patch().deltaCoeffs()
        )
    );
    fvPatchScalarField::evaluate();
}

Foam::tmp<scalarField>
Foam::coupledPressureFvPatchField::snGrad() const {
    return
        valueFraction_
        *(refValue_ - this->patchInternalField())
        * this->patch().deltaCoeffs()
        + (1.0 - valueFraction_)*refGrad_;
}
```

Listing 5.2: `OpenFOAM-Adapter/FF/BoundaryConditions/coupledPressureFvPatchField.C` Excerpt of the `coupledPressureFvPatchField` class. `evaluate()` calculates the new expression for the pressure values at the boundary faces. `snGrad()` returns the boundary face normal gradient.

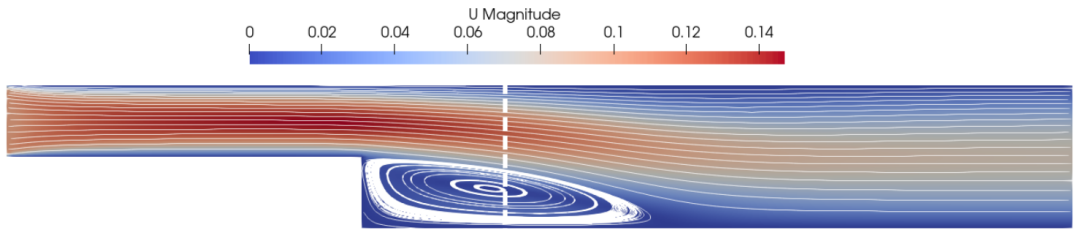
```
1  void Foam::coupledVelocityFvPatchField::updateCoeffs()
2  {
3      const surfaceScalarField& phi =
4          db().lookupObject<surfaceScalarField>(phiName_);
5      const fvsPatchField<scalar>& phip =
6          patch().patchField<surfaceScalarField, scalar>(phi);
7
8      const vectorField& Sf = this->patch().Sf();
9
10     int t0 = patch().boundaryMesh().mesh().time().startTimeIndex();
11     int t = patch().boundaryMesh().mesh().time().timeIndex();
12
13     // if it is the first timestep
14     if (t - t0 == 1)
15     {
16         // evaluate value fraction from coupled velocity values
17         this->valueFraction() = 1 - pos0(refValue_ & Sf);
18     }
19     else
20     {
21     this->valueFraction() = 1 - pos0(phip);
22     }
23
24     fvPatchVectorField::updateCoeffs();
25 }
```

Listing 5.3: `OpenFOAM-Adapter/FF/BoundaryConditions/coupledVelocityFvPatchField.C`
Excerpt of the `coupledVelocityFvPatchField` class. The `updateCoeffs()` function sets the value fraction of the `coupledVelocity` boundary according to the flux field `phi`. In the first time step the coupled velocity values are used to determine the flow direction instead.

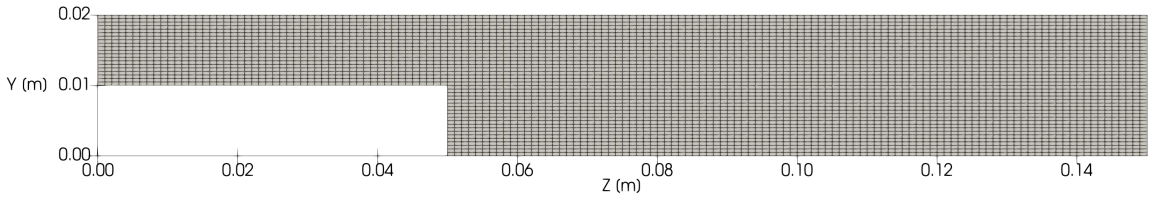
```
// in file 0/p
boundaryField
{
    interface
    {
        type        coupledPressure;
        refValue    uniform 0;
        refGradient uniform 0;
    }
}

// in file 0/U
boundaryField
{
    interface
    {
        type        coupledVelocity;
        refValue    uniform (0 0 0);
        refGradient uniform (0 0 0);
    }
}
```

Listing 5.4: Usage of the `coupledPressure` and `coupledVelocity` boundary conditions. For both the reference value and gradient need to be specified.

5.2.2 Laminar flow over backwards facing step



(a) Streamlines over velocity field



(b) Mesh and dimensions

Figure 5.3: Setup and solution for the flow over a backwards facing step scenario.

To validate my implementation of the inlet-outlet style boundary conditions, I use the scenario of a laminar flow over a backward facing step. The fluid forms a recirculation area behind the step. I place the coupling interface at $z = 0.07\mathrm{m}$, which crosses through the recirculation that forms after a couple of time steps. The majority of the flow goes through the interface in the positive z-direction. But at the bottom interface faces, the fluid flows in negative z-direction. Furthermore, the vortex will take some time steps to develop, which means that the coupled boundary conditions have to dynamically adjust their behavior.

I use the pimpleFoam solver and set a laminar flow. In preCICE, I exchange the quantities $U, p, \nabla U$ in both directions. This creates a little communication overhead, but that seems necessary since it cannot be known what values are needed on each side of the coupling interface. The time step size is set to 0.005 seconds and I simulate until 5 seconds, where the case is converged.

The results of the coupled simulation match the monolithic solutions very well at all time steps, the mean relative error for velocity is around 2×10^{-3}. Figure 5.4 shows the mean relative error for velocity over the simulation time. We can distinguish a few error spikes within the first simulated second. These correspond to the time steps, when new faces at the interface switch their behavior due to the change in flow direction. The cause of these error spikes is still unclear and needs to be investigated, since they cause instabilities when using larger time steps.

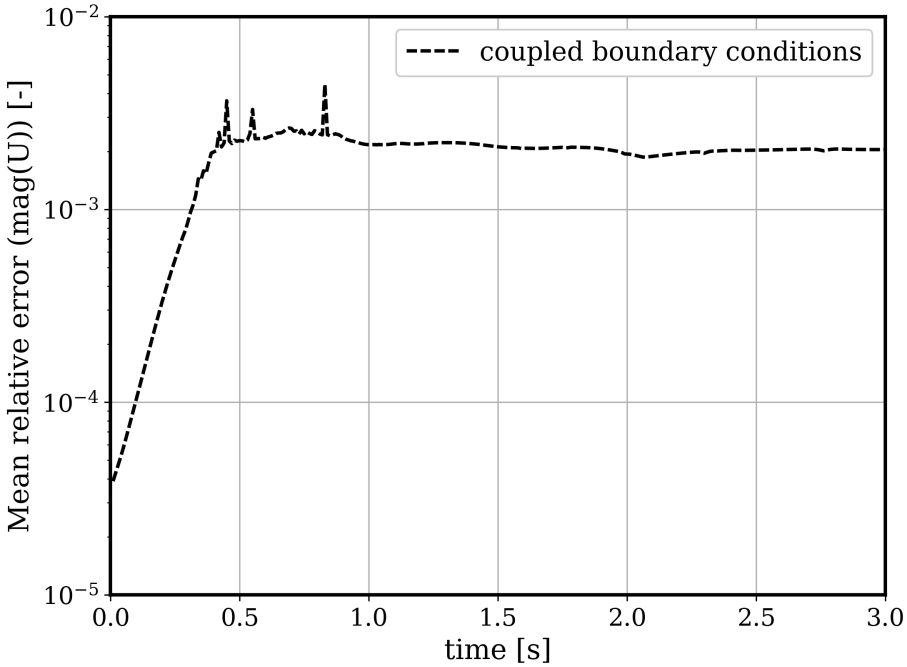


Figure 5.4: Mean relative error of velocity magnitude over time for the backwards facing step case. The error spikes occur when the flow direction across coupled faces changes.

6 Recommendations

In this chapter I want to summarize some concrete tips to fluid-fluid coupling users and developers, based on my discoveries:

Set the fixedFluxExtrapolatedPressure boundary condition at the coupling inlet

When coupling unidirectional flow, the pressure boundary condition at the coupling inlet should be set to `fixedFluxExtrapolatedPressure`. In Subsection 4.2.4 I figured out that OpenFOAM assumes a zero Neumann condition for pressure when we specify an inlet velocity. However, that is physically not true at most coupling interfaces and the `fixedFluxExtrapolatedPressure` boundary condition turned out to be the only available pressure condition that sets a non-zero gradient and doesn't violate mass conservation at the inlet. Using this boundary condition comes with the additional advantage that the pressure gradient does not need to be part of the exchanged coupling variables.

Correct the coupled velocity values for the face flux

Mass conservation is one of the most important aspects when it comes to fluid-fluid coupling. In Subsection 4.4.1, we have seen how that can become problematic when we have non-orthogonal cells at the coupling interface. Fortunately, OpenFOAM handles mass conservation on each subdomain for the face flux field `phi`. Listing 4.5 shows the corrections that can be applied to the exchanged velocity values using the boundary values of `phi`. This correction term should be considered when coupling OpenFOAM fluid solvers on skewed meshes.

Use the provided coupled boundary conditions

I added custom `coupledVelocity` and `coupledPressure` boundary conditions to the preCICE OpenFOAM adapter. These are flexible boundary conditions that can be applied on both sides of the coupling interface and they incorporate the functionalities of the previous two recommendations. They adjust their behavior dynamically for each boundary face depending on the flow direction. For the user, it offers simplicity, as the same boundary condition can be set on both sides of the interface. For the developer, it is easier to further adjust the behavior, if needed. The coupled boundary conditions are specified by the same dictionary entries as the OpenFOAM `mixed` boundary condition.

Place the coupling interface wisely

In this thesis, I wanted to find the limits of our fluid-fluid coupling approach and tested various scenarios, which did not always result in perfect solutions. However, we saw that the coupling error is highly dependent on the magnitude of the velocity gradient that is experienced at the coupling interface. For a fully developed flow, where the velocity gradient is zero, the coupling results match the monolithic solutions very well. In practical scenarios, it is therefore desirable to place the coupling interface as far away from the regions of interest as possible, at locations, where only small changes in velocity are expected.

Refine the mesh next to the interface

The accuracy of a finite volume simulation depends on the discretization in space. A finer mesh is computational more intensive but produces more accurate results. The same applies at the coupling interface. The coupling error can thus be reduced if the mesh is refined next to the interface. It should be enough to refine only the cells next to the coupling interface, because that reduces the error of the extrapolated values that are obtained at the interface. However, it is important to keep an eye on the time step size constraint due to the Courant number limit.

Optimize acceleration and convergence measures

Generally, the best stability for coupled fluid simulations was obtained by using an implicit coupling scheme with Quasi-Newton acceleration for the velocity gradient. I generally used quite strict relative convergence measures in the order of 10^{-5}, which often lead to many required implicit coupling iterations. Further optimization of the implicit coupling configuration can drastically reduce the number of iterations and as a result the runtime. Therefore, the user should think about the necessary convergence measures for each particular case and play around with the variables and setting of the acceleration.

7 Conclusion

In this thesis, I validated the accuracy of the fluid-fluid coupling module of the preCICE OpenFOAM adapter for laminar flows. I explored several validation cases that helped to understand the challenges that arise from a common surface coupling. Additionally, I extended the adapter functionality for flows that transport heat and for reverse flow, which paves the path towards a versatile adapter that supports most major OpenFOAM fluid solvers.

The first validation case was a partitioned laminar flow through a straight pipe. The coupling interface was set in a region where the pipe flow was already fully developed. The standard Dirichlet-Neumann coupling introduced mass flux differences at the coupling interfaces. I discovered that the only viable pressure boundary condition that sets a non-zero gradient without violating the momentum equation at the coupling inlet boundary is `fixedFluxExtrapolatedPressure` (ffep). The ffep boundary condition automatically calculates the pressure gradient according to the inlet velocity and as a result, I needed to exchange only velocity and pressure values via preCICE to obtain coupled results without significant errors compared to the monolithic solution.

The next validation case was designed to have a changing velocity across the interface. Thus, I had to add the velocity gradient to the communicated variables to match the gradient at the interface. The coupled simulation returned average relative errors in the order of 10^{-3} and maximum errors of 10^{-2} in the cells next to the interface. The error depends on the mesh size and the velocity gradient magnitude at the interface. The velocity error $\epsilon(U)$ can generally be quantified as $\epsilon(U) < abs(U_\mathrm{N} - U_\mathrm{P})$, where U_N and U_P are the velocity values in the cells adjacent to the interface in the monolithic solution. I also tried to improve the coupled solutions by manipulating OpenFOAM source code such that the correctness of the momentum equations is enforced at both sides of the coupling interface. This improved the converged coupled solutions, but gave worse results for the first time steps. Ultimately, I realized, that a perfectly coupled simulation needs additional communication hooks that are specific to the numerics of OpenFOAM solvers. Such an approach is not feasible with the adapter implementation as an OpenFOAM function object, but would have to be more invasive.

Lastly, I looked at a widening pipe scenario with non-orthogonal cells. Due to the non-orthogonality, the exchanged velocity quantities resulted in an inconsistent mass flux across the interface. I corrected the velocity values that are read from the coupling outlet for the face flux, which ensured the conservation of mass across the coupling interface. I further noticed that the non-orthogonal correction terms for laplacian schemes are not applied at boundaries, which is another error source in the case of skewed meshes.

After gaining significant knowledge on fluid coupling, I extended the existing fluid-fluid module. I added classes for temperature and temperature gradient coupling and validated those with a laminar, heat-transporting flow. I used Dirichlet-Neumann coupling for temperature on top of the velocity and pressure coupling. The direction of temperature coupling did not affect the accuracy, which was of the order 10^{-6} for the mean temperature error.

Finally, I implemented custom inlet-outlet boundary conditions for velocity and pressure. These combine the flexibility to be used on both sides of the coupling interface and their behavior changes in respect to the flow direction. I validated the boundary conditions with a backwards facing step scenario. The coupling interface was placed in a recirculation area and the boundary conditions switched their functionality for each face according to the flow direction.

7.1 Outlook

Regarding the fluid-fluid coupling module of the preCICE OpenFOAM adapter, it is the goal to support most, if not all, OpenFOAM fluid solvers. For that, the adapter needs to be further extended to support the exchange of all fluid parameters in more complex flows. The next step will be the coupling of *turbulent flows*. For this, it seems desirable to communicate all variables that are used by the considered turbulence model.

Another important aspect to make fluid-fluid coupling practical is the *optimization of implicit coupling*. So far, the implicit coupling takes too many iteration to converge, resulting in total runtimes around twenty times higher compared to the monolithic simulation. Of course, this number will be lower when the size of each subdomain is larger compared to the coupling interface. Still, this is an important issue for future research, which not only concerns fluid-fluid coupling, but all partitioned multiphysics simulations.

The investigation results in this thesis let us conclude that Dirichlet-Neumann coupling is a valid approach for flow partitioning. The remaining errors were not due to the coupling, but rather due to how OpenFOAM solvers handle the boundaries for fields that are created as intermediate steps of the solution algorithms. Instead of zeroth order extrapolation, it might be possible to use a *higher order extrapolation for those boundary field values*. It is possible, however, that such an implementation results in other problems.

Lastly, OpenFOAM is an extremely versatile toolbox for fluid simulations and users have already expressed wishes to couple solvers from different OpenFOAM versions. Therefore, it might be worth to look into a *specific OpenFOAM-to-OpenFOAM preCICE adapter*. Such an adapter would need to be more invasive than the current function object implementation and utilize additional communication hooks to exchange finite volume related variables at several points during each time step, but would enable flow coupling with significantly higher accuracy.

Bibliography

[1] Gerasimos Chourdakis. A general OpenFOAM adapter for the coupling library preCICE. Master's thesis, Technical University of Munich, Oct 2017.

[2] Gerasimos Chourdakis, Benjamin Uekermann, Gertjan van Zwieten, and Harald van Brummelen. Coupling openfoam to different solvers, physics, models, and dimensions using precice. In *14th OpenFOAM Workshop*, Duisburg, Germany, Jul 2019.

[3] Gerasimos Chourdakis, Kyle Davis, Benjamin Rodenberg, Miriam Schulte, Fré déric Simonis, Benjamin Uekermann, Georg Abrams, Hans-Joachim Bungartz, Lucia Cheung Yau, Ishaan Desai, Konrad Eder, Richard Hertrich, Florian Lindner, Alexander Rusch, Dmytro Sashko, David Schneider, Amin Totounferoush, Dominik Volland, Peter Vollmer, and Oguz Ziya Koseomur. preCICE v2: A sustainable and user-friendly coupling library. *Open Research Europe*, 2:51, Apr 2022. doi: 10.12688/openreseurope.14445.1.

[4] Joris Degroote. Partitioned simulation of fluid-structure interaction. *Archives of computational methods in engineering*, 20(3):185–238, 2013. doi: 10.1007/s11831-013-9085-5.

[5] M. Darwish F. Moukalled, L. Mangani. *The Finite Volume Method in Computational Fluid Dynamics*, volume 113. Springer Cham, 1st edition, 2016. doi: 10.1007/978-3-319-16874-6.

[6] Miguel A. Fernández, Jean-Frédéric Gerbeau, and Saverio Smaldone. Explicit coupling schemes for a fluid-fluid interaction problem arising in hemodynamics. *SIAM Journal on Scientific Computing*, 36(6):A2557–A2583, 2014. doi: 10.1137/130948653.

[7] Christopher Greenshields. *OpenFOAM v10 User Guide*. The OpenFOAM Foundation, London, UK, 2022. URL https://doc.cfd.direct/openfoam/user-guide-v10.

[8] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. CFD Direct Ltd, Reading, UK, 2022. URL https://doc.cfd.direct/notes/cfd-general-principles/.

[9] Leopold Grinberg and George Em. Karniadakis. A scalable domain decomposition method for ultra-parallel arterial flow simulations. *Communications in Computational Physics*, 4(5):1151–1169, 2008.

[10] Joel Guerrero. A crash introduction to the finite volume method and discretization schemes in openfoam®. 15th OpenFOAM Workshop, 2020. URL http://www.wolfdynamics.com/training/OF_WS2020/traning_session2020.pdf.

[11] Hrvoje Jasak. Handling parallelisation in OpenFOAM. In *Cyprus Advanced HPC Workshop*. FSB Basel, Switzerland, 2012.

[12] OpenCFD Limited. OpenFOAM v2112 user guide. https://www.openfoam.com/documentation/user-guide.

[13] OpenCFD Limited. OpenFOAM v2112 programmer's guide, Dec 2021. https://sourceforge.net/projects/openfoam/files/v2112/.

[14] Göran Florian Mintgen. *Coupling of Shallow and Non-Shallow Flow Solvers – An Open Source Framework*. Dissertation, Technische Universität München, München, 2018.

[15] Marta Camps Santasmasas. *Hybrid GPU/CPU Navier-Stokes lattice Boltzmann method for urban wind flow*. The University of Manchester (United Kingdom), 2021.

[16] Benjamin Uekermann. *Partitioned fluid-structure interaction on massively parallel systems*. PhD thesis, Technische Universität München, 2016.

[17] S.P. Venkateshan. *Heat Transfer*. Springer International Publishing, 2021. ISBN 9783030583378.

[18] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An Introduction to Computational Fluid Dynamics - The Finite Volume Method*. Pearson Education, Amsterdam, 2007. ISBN 978-0-131-27498-3.

[19] Tom Verstraete and Sebastian Scholl. Stability analysis of partitioned methods for predicting conjugate heat transfer. *International Journal of Heat and Mass Transfer*, 101: 852–869, 2016. ISSN 0017-9310. doi: 10.1016/j.ijheatmasstransfer.2016.05.041.

[20] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in Physics*, 12(6): 620–631, 1998. doi: 10.1063/1.168744.

[21] Helene Wittenberg and Philipp Neumann. Transient two-way molecular-continuum coupling with openfoam and mamico: A sensitivity study. *Computation*, 9(12), 2021. ISSN 2079-3197. doi: 10.3390/computation9120128.