The OpenFOAM®
Technology Primer

Version OpenFOAM-v2012

Tomislav Marić
Jens Höpken
Kyle G. Mooney

License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license.

Disclaimer

Acknowledgments for the first edition

To Marija, Kathi, Olivia and our families.

Contents

List of Acronyms

VoF   Volume-of-Fluid

GPL   General Public License

DNS   Direct Numerical Simulations

CFD   Computational Fluid Dynamics

CAD   Computational Aided Design

FVM   Finite Volume Method

FEM   Finite Element Method

STL   Stereolithography

VTK   Visualization Toolkit

PDE   Partial Differential Equation

RANSE   Reynolds Averaged Navier Stokes Equations

RANS   Reynolds Averaged Navier Stokes

RAS   Reynolds Averaged Simulation

LES   Large Eddy Simulation

DES   Detached Eddy Simulation

DNS   Direct Numerical Simulation

CDS   Central Differencing Scheme

BDS2   second-order accurate Backward Differencing Scheme

DSL   domain specific language

IPC   interprocess communication

MPI   message passing interface

RTS   Runtime Selection

GUI   Graphical User Interface

DSL   Domain Specific Language

RVO Return Value Optimization

RAII Resource Acquisition Is Initialization

HTML HyperText Markup Language

IDE Integrated Development Environment

UML Unified Modeling Language

IO Input/Output

HPC High Performance Computing

VCS Version Control System

OOD Object Oriented Design

PISO Pressure-Implicit with Splitting of Operators

SIMPLE Semi-Implicit Method for Pressure-Linked Equations

AMI Arbitrary Mesh Interface

SRP Single Responsibility Principle

IDW Inverse Distance Weighted

SMP Symmetric Multiprocessor

DMP Distributed Memory Parallel

Forward

From its public release over twenty years ago, OpenFOAM has fundamentally disrupted the world of computational fluid dynamics, spreading through academia and research centers worldwide. There are several reasons for its success, largely tied to the project's open-source nature and the revolutionary design of the software. Foremost, OpenFOAM brings down the cost of the CFD software. For this reason, academia, with its wealth of student labor, led much of early OpenFOAM adoption. The flexibility of open source allowed researchers to innovate with unparalleled freedom. The industry was also intrigued by the ability to customize OpenFOAM and to have solvers specifically designed for their in-house problems. Now, OpenFOAM is a standard of CFD that is familiar to everyone in the field. Yet, there was always a weakness for all of OpenFOAM's strengths: lack of documentation and a relatively steep learning curve.

Theoretically, with open source, the code is the documentation. An experienced user could dig down through the layers of C++ and understand the code's basic functionality. However, this expectation represents a huge barrier to getting started with the software for a new user. This difficulty is compounded by the complexity of the C++ language and the templated OpenFOAM code. Even tools such as Doxygen, which attempts to catalog the OpenFOAM class structure, do not sufficiently reduce the obstacles. On one user forum, a frustrated person lamented that digging through the Doxygen output was, for a C++ beginner, like "reading Chinese backward." The OpenFOAM community needed, very badly, a comprehensive and accessible form of documentation that would provide an on-ramp for the OpenFOAM newbies of the world. Because OpenFOAM is free, the learning curve was the fundamental barrier to entry for OpenFOAM users.

One could buy training from experts in OpenFOAM for the right price, but there is no substitute for a first-rate reference book. And for many years, no reference text offered depth and breadth for the new OpenFOAM

user. That all changed in 2014 when Tomislav Marić, Jens Höpken, and Kyle Mooney published the *OpenFOAM Technology Primer*. This book was a windfall to those who wanted a single source that could make sense of OpenFOAM. The target audience was those who had a basic background in CFD but wanted to dive more deeply into the workings and usage of OpenFOAM. Even though I had nearly a decade of experience with OpenFOAM, that text became the bible of my research lab and lived on my desk, within easy reach. But after a few years of great success, the book utterly disappeared. The community longed for the return of this lovely reference–until 2021.

We are now fortunate that the authors have devoted yet more time to update the text and reissue it. One fundamental challenge of document-ing OpenFOAM is that it is evolving software, ceaselessly changing and improving. To keep such a text current is a Sisyphean task, requiring an unending commitment. For these reasons, I am thrilled to see that the authors chose to release the text under the Creative Commons Li-cense. Much like the decision to open-source the original OpenFOAM code, this valuable text will spearhead a new era in OpenFOAM use and documentation.

——

Professor David P. Schmidt
Dept. of Mechanical and Industrial Engineering
University of Massachusetts Amherst

Authors

Tomislav Marić

Tomislav studied Mechanical Engineering at the University of Zagreb, Croatia, and has obtained his Ph.D. degree in 2017 at the Mathematical Modeling and Analysis institute (MMA), lead by Prof. Dr. rer. nat. Dieter Bothe at the Mathematics Department, at TU Darmstadt (Germany). Tomislav is currently working at TU Darmstadt as an Athene Young Investigator (October 2020). Tomislav has been developing unstructured Lagrangian / Eulerian Interface Approximation (LEIA) methods for simulating two-phase flows in the OpenFOAM open-source software since 2008. He co-founded sourceflux with Jens, and as member of the Collaborative Research Center (CRC) 1194 at TU Darmstadt since 2016, Tomislav supports CRC-1194 researchers in scientific software development and research data management.

Jens Höpken

Jens studied Naval Architecture at University of Duisburg-Essen and graduated at the Institute of Ship Technology, Ocean Engineering and Transport Systems (ISMT) of the University Duisburg-Essen, Germany. Jens is working with OpenFOAM since 2007, he is an OpenFOAM expert with over a decade of experience in developing OpenFOAM for Naval Hydrodynamics. Jens co-founded sourceflux with Tomislav.

Kyle G. Mooney

Kyle received his Ph.D. in Mechanical Engineering from The University of Massachusetts Amherst in 2016. His research involved numerical simulation of viscoelastic fluids and spray droplet dynamics. Following graduate school he joined ICON Technology Process & Consulting where he helped innovate automotive aerodynamics simulation processes for Ford Motor Company, Fiat Chrysler Automobiles, and General Motors. After moving to San Francisco he shifted to leading fluid mechanical R&D work along with hardware and software product development at several

startups. He is currently a Senior Application Engineer at Geminus.AI creating multi-fidelity flow system simulation models for industrial applications. In his free time you'd likely find him boxing, skateboarding, hiking the High Sierra, or performing music.

Preface

The OpenFOAM open-source software for Computational Fluid Dynamics (CFD) is widely used in industrial and academic institutions. Compared to using proprietary CFD software, the advantage of using OpenFOAM lies in the *Open-Source* General Public License (GPL), which allows the user to *freely use* and *freely modify* a modern high-end CFD code. An open-source license removes license costs in the product optimization cycle, enables straightforward automation of parameter variations, and accelerates the development of new numerical methods and models. The development and implementation of novel methods are accelerated because they start from existing functionality in OpenFOAM, instead of starting from scratch.

Next to all the advantages mentioned above, there is one disadvantage in working with OpenFOAM. The C++ programming language and modern software design patterns are used in OpenFOAM, so substantial effort is required to learn how to program new methods in OpenFOAM in a modular and sustainable way. Working with OpenFOAM - one might say this for Computational Science in general - additionally requires a combination of knowledge from different backgrounds, involving applied mathematics, physics, software development, the C++ programming language, and high-performance computing (parallel programming and performance measurement).

This book is an effort to describe different aspects of OpenFOAM in a single place to a beginning OpenFOAM user develop into an intermediate OpenFOAM programmer. To achieve this goal, we strongly advise the reader to work through the covered examples. The more advanced aspects of the C++ programming language - required for some core parts of OpenFOAM - are not covered in this book. However, this knowledge is available in books on the C++ programming language and software design patterns.

This book covers two main aspects of working with OpenFOAM: using the applications and developing and extending OpenFOAM applications and libraries. The first part describes the OpenFOAM workflow using a couple of OpenFOAM utilities and applications. The second part of the book covers the sustainable development of new solvers and libraries in OpenFOAM.

Intended audience

This book is intended for anyone interested in open-source Computational Fluid Dynamics (CFD).

However, it was not possible to provide all the background information on the C++ programming language, software design, Computational Fluid Dynamics (CFD), and high-performance computing (HPC) necessary to effectively develop OpenFOAM in a single book; instead, the focus is placed on the usage, design, and development of OpenFOAM solvers and libraries, and the reader is guided to other sources of in-depth information on the topics that are not covered in full detail.

Therefore, some knowledge of object-oriented programming in the C++ programming language is assumed, involving classes (encapsulation, inheritance, and composition), virtual functions (dynamic polymorphism), and operator overloading. The background information on these topics specific to each chapter is provided throughout the book's second part. However, the reader is also expected to learn about these topics independently, using the literature cited at the end of each chapter. The provided examples deliberately avoid verification and validation, as this puts the reader off track from learning OpenFOAM. However, it is not possible to learn and understand OpenFOAM without the knowledge of Computational Fluid Dynamics and, in the case of OpenFOAM, the unstructured Finite Volume Method (FVM), briefly addressed in this book and covered elsewhere in more detail.

What is covered in this book

Chapter 1 provides an overview of the workflow for CFD simulations and an overview of the unstructured Finite Volume Method (FVM)

in OpenFOAM.

Chapter 2 covers the domain discretization (mesh generation and conversion) and domain decomposition.

Chapter 3 describes the structure and the setup of a simulation case: setting initial and boundary conditions, configuring the simulation control parameters, and numerical settings.

Chapter 4 provides an overview of pre-and post-processing utilities and data visualization.

Chapter 5 provides an in-depth overview of an OpenFOAM library compared to chapter 1.

Chapter 6 describes how to program OpenFOAM in a productive and sustainable way: developing and using libraries, using the git version control system, debugging and profiling, and so on.

Chapter 7 provides a brief overview of turbulence modeling: introducing turbulence into a simulation case and configuring the turbulence model.

Chapter 8 covers programming OpenFOAM pre- and post-processing applications.

Chapter 9 describes the background of solver design in OpenFOAM, and shows how to extend an existing solver with new functionality.

Chapter 10 shows the numerical background and software design aspects of boundary conditions in OpenFOAM. An implementation example of a custom boundary condition is provided that uses the principles described in chapter 6.

Chapter 11 covers the numerical background, design, and implementation of transport models in OpenFOAM, using a temperature-dependent viscosity model as an example.

Chapter 12 covers the design and implementation of function objects in OpenFOAM, in comparison to C++ function objects.

Chapter 13 covers dynamic mesh handling in OpenFOAM. The design and usage of the dynamic mesh engine in OpenFOAM is covered, as well as the extending a solver with dynamic mesh handling.

Chapter 14 concludes the book.

How to read this book

For beginning OpenFOAM users, it is recommended to read the book from beginning to end and work indepenently on the examples. The

experienced OpenFOAM users may select a chapter from the second part with the relevant information on how to program a specific part of OpenFOAM.

The OpenFOAM version

Different versions of OpenFOAM are available, such as OpenFOAM Foundation, Foam Extend and OpenFOAM. The differences and similarities between these "forks" are not covered in this book. The book text and the example repository match OpenFOAM-v2012, and the book will follow only the respected new releases of this OpenFOAM version. The information on how to install this version of OpenFOAM is available on this website. There is a choice between installing OpenFOAM as a Linux package, compiling a snapshot of the source code, or compiling the cloned git repository. Since the goal of this book is to address programming OpenFOAM, compiling the source code snapshot or compiling the clone of the OpenFOAM git repository are the recommended options.

Naming and typesetting conventions

The command line is typeset using typewriter with a prepending ?> . An example is shown below:

```
?> ls $FOAM_TUTORIALS
Allclean basic         electromagnetics lagrangian
Allrun   combustion    financial        mesh
Alltest  compressible  heatTransfer     multiphase
DNS      discreteMethods incompressible  stressAnalysis
```

A C++ code block is typeset like this:

```
template<class GeoMesh>
tmp<DimensionedField<scalar, GeoMesh> > stabilise
(
    const DimensionedField<scalar, GeoMesh>&,
    const dimensioned<scalar>&
);
```

The configuration of OpenFOAM simulations relies on so-called *dictionary files*. The dictionary files are text files that store lists of key-value pairs in an OpenFOAM-specific format:

```
ddtSchemes
{
    default       Euler;
}
```

In equations, scalars such as the flux ϕ are typeset without emphasis; vectors are typeset with bold-face (e.g., velocity **U**); tensors are bold-face and underlined (e.g., unit matrix **E**).

Example OpenFOAM repository

The examples covered in the book are available on GitLab: https://git-lab.com/ofbook-/ofprimer.

The latest release is merged into the master branch, all releases match OpenFOAM git tag releases, e.g., OpenFOAM-v2012.

Contributing

Bug reports and topic suggestions are handled using the GitLab Service Desk, and they can be voted on. The topics with the most upvotes will have a higher chance of being addressed in the next edition.

Before submitting a bug report or a feature request, search the existing issues to make sure it has not already been reported.

Part I

Using OpenFOAM

1

Computational Fluid Dynamics in OpenFOAM

In this chapter, an overview of the workflow used to solve a Computational Fluid Dynamics (CFD) problem in OpenFOAM is presented. Additionally, this chapter contains background information about the Finite Volume Method (FVM) on unstructured meshes used by OpenFOAM, as well as the top-level structure of the OpenFOAM platform.

1.1 Understanding The Flow Problem

The goal of any CFD analysis is to obtain a deeper understanding of the problem under consideration. As simulation results are often accompanied by experimental data, the validity of results, and the overall aim of the simulation needs to be taken into account. Furthermore, the simulated physical process needs to be properly mathematically modeled. This, in turn, requires the CFD engineer to make a valid choice of the appropriate solver within the OpenFOAM framework. During the CFD analysis, assumptions may either complicate or simplify the analysis. Analysis and reduction of the simulation domain geometry are often performed, considering the available computational resources, required simulation fidelity, and desired turnaround time.

Pragmatic questions that might be posed before undertaking a CFD analysis project are outlined below. This list is by no means exhaustive.

General considerations

- What should be the conclusion resulting from the CFD analysis?
- How is the degree of accuracy of the results defined?
- With what methods will the results be validated?
- How much time is available for the project?

Thermo-physics

- Is the flow laminar, turbulent, or transitional?
- Is the flow compressible or incompressible?
- Does the flow involve multiple fluid phases or chemical species?
- Does heat transfer play an important role in the problem?
- Are the material properties functions of dependent variables? For example, a shear-thinning fluid.
- Is sufficient information available on boundary conditions and initial conditions? Are they appropriately modeled, and can they be accurately approximated?

Geometry and mesh

- Is an accurate discrete representation of the flow domain possible?
- Will the computational domain be deforming or moving during the simulation?
- Where can the complexity of the domain be reduced without impacting the solution accuracy?

Computational Resources

- How much computational time is available for the simulation?
- What kind of distributed computing resources are available?
- Will one simulation suffice, or is a parameter variation necessary?

These questions contribute to an accurate CFD analysis of any flow problem. Using OpenFOAM or any other CFD simulation software requires a proper understanding of physics, numerical methods, as well as available computational resources. The interdisciplinary nature of CFD greatly contributes to its complexity.

1.2 Stages of a CFD analysis

A CFD analysis of a problem can be described in five steps. Some of these steps must be performed multiple times to obtain high-quality results.

1.2.1 Problem definition

The numerical model should be as simple as possible from the engineering perspective and still accurately describe the real-world engineering system. Neglecting irrelevant aspects of the simulated problem increases the efficiency of a CFD analysis because it simplifies the physical process and thus the mathematical model that describes it. For example, even though the air is compressible, simulating a flow over an airfoil in some flow regimes involves considering the air to be an incompressible fluid.

1.2.2 Mathematical modeling

Once the relevant aspects of the physical process are isolated, the problem requires mathematical description in the form of a mathematical model, which is in CFD usually a set of Partial Differential Equation (PDE)s. A CFD engineer must understand the models used to describe different physical phenomena. Within the OpenFOAM framework, the user has a choice between dozens of solvers. Each solver implements a specific mathematical model, and choosing the correct one is often crucial to obtaining a valid solution to the simulated problem. The incompressibility assumption for a flow of air over an airfoil excludes the solution of the energy equation. As another example, a potential flow is solely governed by the Laplace equation - details can be found in the book by Ferziger and Perić [1]. If more complex physical transport phenomena are taken into account, the complexity of the mathematical model increases. This typically leads to more sophisticated mathematical models, e.g., the Reynolds Averaged Navier Stokes Equations (RANSE), used to model the turbulent flow. The mathematical model describes the flow details, which means that the numerical simulation, which only approximates the solution of the model at best, cannot produce more information about the flow than what is available in the model. More information on turbulence

modeling in OpenFOAM can be found in chapter 7. More details on the particular mathematical models can be found in standard fluid dynamics textbooks.

1.2.3 Pre-processing and mesh generation

The mathematical model defines fields of physical properties as dependent variables of the model equations. In CFD, more often than not, the equations describe a boundary and initial value problem. Therefore, the fields need to be initially set (pre-processed) before the start of the simulation. If the field values are spatially varying, different utility applications (utilities) may be used to compute and pre-process the fields. There are utilities that are distributed along with OpenFOAM (e.g. the `setFields` utility), or are a part of another project (e.g. `funkySetFields` utility of the `swak4Foam` project).

The use of some of the available pre-processing utilities is explained in chapter 8.

INFO

More information on the `swak4Foam` project can be found on the Open-FOAM wiki on `http://openfoamwiki.net/index.php/Contrib/swak4Foam`.

The flow domain must be discretized in order to approximate the model solution numerically. The spatial discretization of the simulation domain consists of separating the flow domain into a *computational mesh* consisted of volumes (cells), often of different shapes.   All of these volumes taken together are referred to as the 'mesh' or the computational grid. Usually, the mesh must be refined in areas of interest: for example in those parts of the domain where large gradients of field values occur. Further on, the accuracy and a proper choice of the mathematical model has to be kept in mind. Resolving flow features in a spatial manner does not compensate for a model that does not account for these features in the first place. On the other hand, increasing the mesh resolution for transient simulations may prohibitively slow down the simulation. This is due to the often small value required for the discrete time step in order to obtain a stable solution, when explicit discretization schemes are used.[1] The

---

[1]More information on this issue is provided in CFD textbooks, such as the book of [1].

mesh is one of the most likely components of the simulation workflow that needs to be changed if the numerical simulation fails to converge. Failing simulations are very frequently caused by a mesh of insufficient quality. OpenFOAM comes with two different mesh generators which are namely `blockMesh` and `snappyHexMesh`. The usage of both is covered in chapter 2.

Additionally, pre-processing covers various other tasks, such as decomposing the computational domain if the simulations are run in parallel on multiple computers or CPU cores.

1.2.4 Solving

Alongside mesh generation, this usually is the most time consuming part of the CFD analysis. The time required highly depends on the mathematical model, the numerical scheme used to approximate its solution, as well as the geometrical and topological nature of the computational mesh. In this step, the differential mathematical model is replaced by a system of (linearised) algebraic equations. In CFD, such algebraic linear equation systems are often *large*, resulting with matrices with millions or billions of coefficients. The algebraic equation systems are solved using algorithms developed specially for this purpose - iterative linear solvers. OpenFOAM framework supports a wide choice of linear solvers although the solver applications generally have preset or ideal choices of linear solvers and parameters. A skilled CFD engineer has the opportunity to modify both the solver and the corresponding parameters. The choice of the linear solver and discretization strategy is important because it impacts both computation speed and stability.

1.2.5 Post-processing

After the simulation completes successfully, the user often ends up having a large amount of data that must be analyzed and discussed. The data must be extracted, plotted, and/or visualized appropriately in order to inspect the details of the flow. By using dedicated tools such as paraView, such data can be discussed fairly easily. For analyzing the simulation results, OpenFOAM provides a wide choice of post-processing applications.

More details on various post-processing tools and methods are provided in chapter 8.

> **INFO**
>
> The standard post-processing tool, that comes with OpenFOAM is paraView. It is an open-ource tool and can be obtained from `www.paraview.org`.

1.2.6 Verification and validation

This is the point where the user must determine whether to trust the results or not. Generally, CFD software is complex, and it relies on configurable parameters, which leaves ample room for error. If a mistake is made in one of the previous steps, it will most likely be discovered during verification and validation.

Verification ensures that a numerical method properly solves the mathematical model it approximates. In other words, verification checks if the solution of a mathematical model is approximated appropriately.

Validation compares simulation results with experimental data: it introduces a more strict safety margin when it comes to confidence in the simulation results. When comparing against experiments, validation ensures that the *right* mathematical model is chosen and that its solution adequately reflects reality. If the simulation results do not satisfy the requirements, previous steps of CFD analysis must be revisited.

1.3 Introducing the Finite Volume Method in OpenFOAM

An overview of the Finite Volume Method (FVM) in OpenFOAM is presented in this section. A more detailed description of the FVM can be found in Ferziger and Perić [1], Versteeg and Malalasekra [13], Weller, Tabor, Jasak, and Fureby [14], Jasak, Jemcov, and Tuković [4], and Moukalled, Mangani, Darwish, et al. [6] for more detailed descriptions.

Steps of the unstructured FVM in OpenFOAM correlate somewhat to the steps of the CFD analysis described in section 1.2. The physical

properties that define the fluid flow, such as pressure, velocity, or temperature/enthalpy are dependant variables in a *mathematical model*: a formal mathematical description of the fluid flow. A mathematical model that describes a fluid flow is defined as a system of PDEs.

Different physical processes are sometimes described using a similar mathematical description, e.g. the conduction of heat as well as diffusion of sugar concentration in water are modelled as *diffusive processes*. The general scalar transport equation, as described in [1], contains the terms (differential operators) that model different physical processes, for example: transport of particles with the fluid velocity (advection term), heat source (source term), and so forth. As it contains often encountered terms, the general scalar transport equation is used to describe the FVM, by discretizing

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi) - \nabla \cdot (D\nabla\phi) = S_\phi. \tag{1.1}$$

Here, ϕ is some transported scalar property, $\mathbf{U}$ the prescribed velocity and D the diffusion coefficient. The terms in equation (1.1) from left to right are: temporal term, convective term, diffusive term and source term. Each term describes a physical process that changes the property ϕ in a different way.

Depending on the nature of the process, some of the terms may be neglected: e.g. for the inviscid fluid flow, the diffusive term (transport) of the momentum is neglected in the momentum equation. In addition, the coefficients that appear in some of the terms may be constant values, spatially/temporally varying fields, or dependent on the physical property. An example of such a dependent coefficient is a temperature-dependent conductivity coefficient for conductive heat transfer: $\nabla \cdot (k(T)\nabla T)$, which makes k a spatially/temporally varying field that would depend on the model solution: the temperature field.

The purpose of any numerical method is to approximate the solution of the mathematical model. An *approximation* of the solution of a complex physical process is necessary because the *exact* solution can be obtained only for very special cases that often lack relevance for technical or engineering applications.

Approximative solution of the mathematical model is obtained by solving a discrete approximation of the governing system of equations. The

Ω

$\Omega \approx \Omega_D = \cup_{c \in C} \Omega_c$

$\circ \mathbf{x}_c$

y

x

Ω_c

$\phi = \phi(\mathbf{x}, t)$

$\phi(\mathbf{x}, t) \doteq \phi(x_c) + \nabla\phi(\mathbf{x}_c)(\mathbf{x} - \mathbf{x}_c)$

(a) Continuous flow domain.

(b) Discretized flow domain.
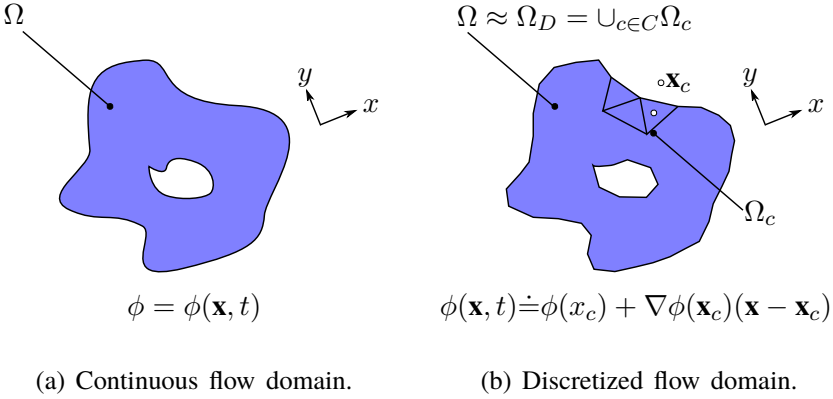
Figure 1.1: Continuous and discretized flow domains and the corresponding continuous and discrete flow variables.

approximation process of the Finite Volume Method (FVM) involves a substitution of the system of PDEs with a corresponding system of linear algebraic equations, that can subsequently be solved using a computer. The generation of an algebraic system of equations by the unstructured Finite Volume Method (FVM) is made up of two major steps: domain discretization and equation discretization.

1.3.1 Domain discretization

The mathematical model uses continuous variables. In order to approximate the solution of the mathematical model, space is discretized into a finite number of volumes (mesh cells). The finite volumes (cells) make up the finite volume mesh. The transition from the continuous representation of the flow with continuous fields filling the flow domain Ω, to a discretized domain Ω_D is shown in figure 1.1. The continuous domain Ω is approximated as a union of mesh cells (finite volumes)

$$\Omega \approx \Omega_D = \cup_{c \in C} \Omega_c, \tag{1.2}$$

where the union Ω_D is the finite volume mesh and C is the set that contains indexes of all cells in the mesh Ω_D. Continuous fields that are defined in each point of the space filled by the fluid in figure 1.1a are approximated linearly inside the finite volumes Ω_c, as shown in figure 1.1b.

Each finite volume stores a volume-averaged value of the physical property (e.g. temperature) that is associated to its centroid $\mathbf{x}_c$. Associating the value to the centroid makes the domain discretization second-order accurate. To see why this is the case, let us assume that the field ϕ can be represented using the Taylor series expansion as

$$\phi(\mathbf{x}) = \phi(\mathbf{x}_c) + \nabla\phi(\mathbf{x}_c)\cdot(\mathbf{x}-\mathbf{x}_c) + \nabla\nabla\phi(\mathbf{x}_c) : (\mathbf{x}-\mathbf{x}_c)\otimes(\mathbf{x}-\mathbf{x}_c) + ... \quad (1.3)$$

Expressing the volume-averaged value of $\phi(\mathbf{x})$ inside the cell Ω_c with the Taylor series expansion in (1.3) introduces quantities that are defined at the point $\mathbf{x}_c$, whih are constant over Ω_c. This leads to

$$\phi_c = \phi(\mathbf{x}_c) + \nabla\phi(\mathbf{x}_c)\cdot\frac{1}{|\Omega_c|}\int_{\Omega_c}(\mathbf{x}-\mathbf{x}_c)\,dV$$

$$+ \nabla\nabla\phi(\mathbf{x}_c) : \int_{\Omega_c}(\mathbf{x}-\mathbf{x}_c)\otimes(\mathbf{x}-\mathbf{x}_c) + ... \, dV. \quad (1.4)$$

The definition of the centroid $\mathbf{x}_c$ of the volume Ω_c gives

$$\int_{\Omega_c}(\mathbf{x}-\mathbf{x}_c)\,dV = 0, \quad (1.5)$$

which, inserted to equation (1.4) results in

$$\phi_c = \phi(\mathbf{x}_c) + \int_{\Omega_c}\nabla\nabla\phi(\mathbf{x}_c) : (\mathbf{x}-\mathbf{x}_c)\otimes(\mathbf{x}-\mathbf{x}_c) + ... \, dV. \quad (1.6)$$

Equation (1.6) shows that the average value of ϕ over the finite volume Ω_c is exactly equal to the value of ϕ at the centroid $\mathbf{x}_c$ of Ω_c for a linear ϕ, because for a linear ϕ the higher-order derivatives are zero. In other words, the cell-average (cell-centered) value at the centroid of finite volumes recovers values of linear fields exactly. A method that exactly recovers values of linear functions is at least second-order accurate.

INFO

The domain discretization of the unstructured FVM that assigns cell-average values ϕ_c of ϕ at centroids $\{\mathbf{x}_c\}_{c\in C}$ of the cells $\{\Omega_c\}_{c\in C}$ is second-order accurate.

An interpolation of ϕ based on (1.6),

$$\phi_c \approx \frac{1}{|\Omega_c|}\int_{\Omega_c}\phi_l(\mathbf{x}_c)\,dV + O(\|\mathbf{x}-\mathbf{x}_c\|_2^2), \quad (1.7)$$

is second-order accurate, because the largest term in the truncated part of the Taylor series in equation (1.6) scales with $\|\mathbf{x} - \mathbf{x}_c\|_2^2$.

In (1.3), the gradient $\nabla\phi(\mathbf{x}_c)$ at the cell center must also be approximated. This is covered in the section on equation discretization 1.3.2. Before understanding how equations are discretized, the mesh Ω_D should be defined in more detail, specifically how the cells $\{\Omega_c\}_{c \in C}$ are connected to each other as this connectivity determines which domains can be discretized.

We distinguish here three major types of meshes: structured, block-structured and unstructured meshes. All three impose different require-ments for domain and equation discretization and are different in the way the mesh elements are mutually connected and addressed. The con-nectivity between the mesh elements (mesh connectivity) determines the how neighboring elements are accessed, which is important for equation discretization. This further impacts possible optimizations in accessing mesh elements, and the optimizations reflect in the efficiency of numer-ical operations as well as how those operations can be parallelized. For example, unstructured addressing of mesh cells makes it difficult to ac-cess cells in any specific direction. This complicates the construction of larger stencils on unstructured meshes that needed by higher-order in-terpolation schemes, because they rely on cell-averaged (cell centered) values from a wider neighborhood. The parallelization of such higher-order interpolations that rely on cell-centered values using the domain decomposition and message-passing approach is also complicated and likely inefficient because large messages of variable length have to be communicated across process boundaries. The mesh connectivity there-fore determines what can be computed on the mesh in an efficient way, sometimes to such effect, that the inefficiency caused by choosing the wrong numerical method for a specific mesh renders a method unusable for problems of practical interest, which generally require a larger num-ber of cells. In OpenFOAM, the mesh connectivity also plays a very strong in the way numerical algorithms are implemented: OpenFOAM only supports unstructured meshes.

Structured meshes support direct addressing of an arbitrary cell neighbor as well as direct cell traversal: the cells are labeled with the indexes increasing in the directions of the coordinate axis (see figure 1.2a). Un-structured meshes on the other hand have no apparent direction (see figure 1.3a).

i-1,j+1	i,j+1	i+1,j+1
i-1,j	i,j	i+1,j
i-1,j-1	i,j-1	i+1,j-1
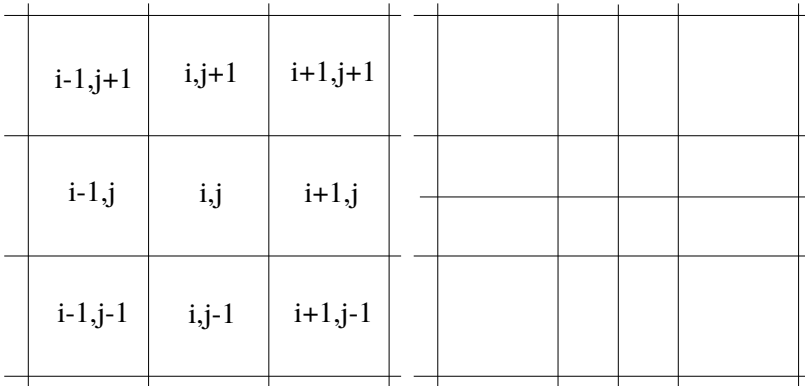
(a) Sub-set of a 2D equilateral Cartesian mesh.

(b) Refined sub-set of a 2D equilateral Cartesian mesh.

Figure 1.2: Structured quadratic mesh.

7	18	4
11	9	6
3	5	10
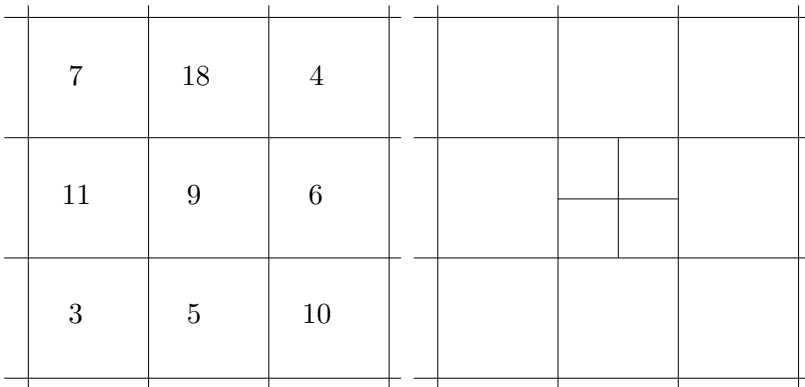
(a) Sub-set of a 2D unstructured quadratic mesh.

(b) Refined sub-set of a 2D unstructured quadratic mesh.

Figure 1.3: Unstructured quadratic mesh.

A structured mesh connectivity increases the absolute accuracy of the interpolations involved in the FVM, but it makes the mesh less flexible when it is used for mesh generation of geometrically complex domains. During the mesh generation process, the user usually desires to generate a dense mesh where large changes in the solution occur and not to computational resources by adding cells in flow regions where the solution is almost constant. All mesh generation steps should happen in the shortest possible time, which is impossible to achieve with structured meshes without substantial enhancements. Local mesh refinements are virtually impossible to achieve, since the refinement has to be propagated into the respective direction through the entire mesh. An example sketch of the difference in topology between the structured and unstructured mesh is given in figure 1.3b.

Figure 1.2a shows a two dimensional schematic of a structured quadratic mesh. This kind of mesh is also called Cartesian, since it is consisted of volumes with volume centers distributed in the direction of the coordinate system axes. Being able to move through the mesh by changing the indexes i, j has one significant advantage: the numerical method working with this kind of mesh may access any neighboring cell by simply incrementing or decrementing the indexes i, j by an integer value of 1. This comes in handy e.g. when the flux through the cell-faces ϕ_f is interpolated from cell centers to the face centre: high-order interpolation stencils can be easily applied, to increase the accuracy of the solution. A high-order (large) interpolation stencil means that an increased number of cells, that need not necessarily be face-neighbors of the current cell, are included into the interpolation.

There is one problem, however: it is impossible to refine the mesh locally, i.e. in a sub-region of the mesh.

This is often the case in a region of extreme changes, when the order of interpolation available on the structured mesh is still insufficiently accurate. For example, abrubpt changes in the values of physical properties are present in shocks or two-phase flow simulations where two immiscible fluids are simulated. An interface is formed between two fluids that separates them, and the values of the physical properties may vary by orders of magnitude, as can be seen in a schematic diagram in figure 1.4.

To resolve such steep changes in values, the mesh is locally refined. This
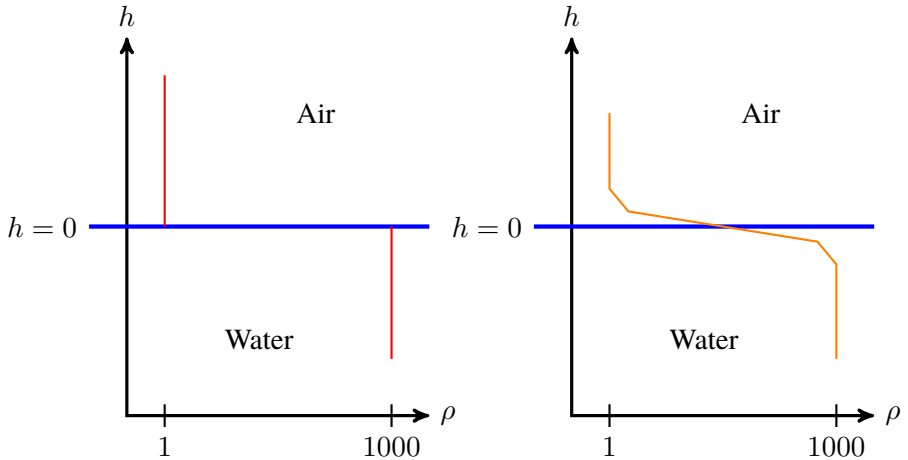
(a) Continuous space with sudden jump of the density ρ

(b) Discretized space with gradual - but still steep - jump of the density ρ

Figure 1.4: Qualitative distribution of the density ρ with respect to the height h over the free surface.

refinement can either be performed during pre-processing or applied during runtime. As mentioned previously, refining a Cartesian mesh cannot be done locally: the topology of a structured mesh forces the mesh to be refined in the complete direction (see figure 1.2b), where the refinement of a single cell in two directions generates refinement throughout the entire mesh. Structured meshes that conform to curved geometries are especially difficult to generate. A parameterization of curved domain boundaries (coordinatization) is necessary in order to maintain the structured mesh connectivity, which is only possible for relatiely simple boundaries.

There are, however, extensions of the structured mesh discretization practice and some of them allow for both local and dynamic mesh refinement. In order to increase the accuracy locally, block structured refinement may be used, which is a process of building a mesh that consists of multiple structured blocks. When such a block structured mesh is assembled, the blocks will have different local mesh densities. Unfortunately this introduces a new complication because the numerical method must be able to deal with non-conforming block patches (hanging nodes). Alternatively,

the block refinement must be carefully crafted such that the points on adjoining blocks of different densities *match perfectly* (patch-conforming block-meshes). Constructing block structured meshes is a complex problem even for simple flow domains. This often makes block-structured meshes a poor choice for many technical applications that involve complex geometries or boundary shapes. Refining block-structured meshes results in refinement regions spreading through the blocks. With standard solvers that rely on patch-conforming block-meshes the refinement complicates the mesh generation further.

Dynamic adaptive local refinement in OpenFOAM is performed by introducing additional data structures that generate and store the information related to the refinement process. An example of such method is an *octree based refinement*, where an octree datastructure is used to split the cells of the structured Cartesian mesh into octants. This requires the mesh (or at least the subset of the mesh undergoing refinement) to consist solely of hexahedral cells. Information stored by the octree data structure is then used by the numerical interpolation procedures (discrete differential operators) taking into account the topological changes resulting from local mesh refinement. The possibility of dealing with more complex geometrical domains can then be added to an octree-refined structured mesh by using an *cut-cell* approach. In that case, the cells which hold the curved domain boundary, are cut by a piecewise-linear approximation of the boundary. Octree-based adaptive mesh refinement may have an advantage in its efficiency depending on the way the topological operations are performed on the underlying structured mesh. However, the logic of the octree based refinement requires the refining domain to be box-shaped. More information about local adaptive mesh refinement procedure can be found in [8].

OpenFOAM implements a FVM of second-order of convergence with support for *arbitrary unstructured meshes*. In addition to the unstructured mesh connectivity, mesh cells of an arbitrary unstructured mesh can be of any shape. This allows the user to discretize flow domains of very high geometrical complexity. The unstructured mesh allows for a very fast, sometimes even *automatic mesh generation* procedure. This is very important for industrial applications, where the time needed to obtain results is of great importance.

Figure 1.3a shows a two-dimensional schematic of a quadratic unstructured mesh. Since the mesh addressing is not structured, the cells have

been labelled solely for the purpose of explaining the mesh connectivity. The unordered cells complicate the possibility to perform operations in a specific direction without executing costly additional searches and re-creating the directional information locally. Another advantage of the unstructured mesh is the ability for a cell to be refined locally and directly, which is sketched in figure 1.3b. The local refinement is more efficient in terms of the increase of the overall mesh density, since it only increases the mesh density where it is required.

INFO

Although the connectivity of the mesh is fully unstructured in Open-FOAM, often the mesh will be generated block-wise for cases with simple domain geometry (`blockMesh` utility): this does not mean that a *block structured* mesh is generated.
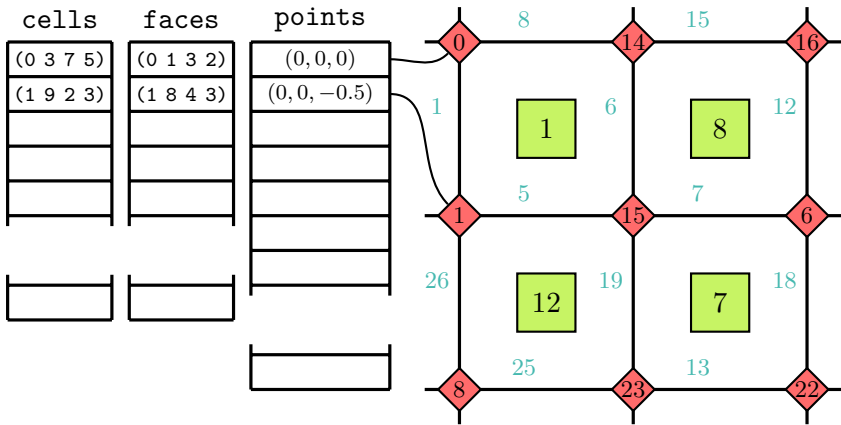
Figure 1.5: Example 2D unstructured mesh connectivity with labels of different shapes used for cells (squares), points (diamonds) and faces.

The way the mesh elements are addressed by the algorithms of the numerical method is determined by the mesh connectivity. OpenFOAM relies on *indirect addressing*, *owner-neighbor addressing* and *boundary mesh addressing* to address mesh elements.

Indirect addressing defines how the mesh is assembled from the mesh points, which are given as a global list of points. Cell faces are index the global mesh point list using *integer labels*. This way, the

coordinates of the points are not repeated to construct the mesh faces, which saves memory and increases computational efficiency. Each cell face is therefore a list of integer indexes to global mesh points. All faces in the mesh are stored in a global list of faces: a *list of lists of indexes*, which is what the name *indirect addressing* implies. Similarly, cells are constructed as lists of indexes that address positions of faces in the global list of faces, shown in figure 1.5. As the faces, the cells are also stored in a global (cell) list. Face 1, as shown in figure 1.5, serves as an example. It is constructed from points 0 and 1, as well as 3 and 2, which are not shown in figure 1.5. This face in turn is used to assemble cell 1. Indirect addressing used for faces and cells avoids copying of mesh points whenever an instance of a face or cell is created. Otherwise one would end up having multiple copies of the same points and faces in memory, which would be a waste of computing capacity and would severely complicate numerical and topological operations.

Owner-neighbor addressing defines which cell owns a certain face and which is its neighbor. In addition it is an optimization of access to mesh faces. When transport equations are discretized using a second-order accurate unstructured FVM, the discretization consists of sums over faces for each cell with interpolation being used at the center of each face. As adjacent cells share faces, the computing the discretization for each face results in two same calculations done at each face shared by two cells. To avoid this, two global lists are introduced into the mesh with owner-neighbor addressing optimization: the *face-owner* and the *face-neighbor* list. Each face of the mesh can be shared by maximally two cells. Since the cells are stored in a global list of cells, they are uniquely identified by their position index in that list (*cell label*). The cell that has the smaller cell label then becomes is *owner cell* of a face shared with the cell that has a larger cell label. The owner cell of a face f is marked with O_f in figure 1.6. The other cell adjacent to face f, but with a larger cell label is called *face-neighbor* and it is marked with N_f in figure 1.6 for face f. The direction of the face area normal vector $\mathbf{S}_f$ is shown as an arrow in figure 1.6. The face area normal vector is directed always from the owner into the neighbor cell, from the cell with a smaller index, into the cell with a larger index. This way, the discretization is computed not over all mesh

cells, but over all faces, only once. The contribution at the face center is then added to the owner and subtracted from the neighbor cell by the discretized differential operator, following a convention of the outer pointing normal area vector.

Boundary addressing takes care of how the faces on a boundary are addressed. By definition, all faces that only have a single owner cell and no neighbor cell, are boundary faces. To increase efficiency, the boundary faces are stored at the end of the global list of mesh faces and are grouped in *patches*. Grouping of boundary faces into boundary patches is related to applying boundary conditions, that will be different for different groups of boundary faces. The complete boundary mesh is therefore defined as a list of boundary patches. This allows an efficient definition of the boundary patches as subsets of the global mesh face list. Such definition of the boundary mesh results in the automatic parallelization of all the top-level code in OpenFOAM that relies on the face-based interpolation practice. Because they have only one cell owner, all the normal vectors of the boundary mesh are directed outwards of the solution domain.

While the indirect addressing and the unstructured mesh connectivity increase the flexibility in handling complex geometries and applying local refinement, the indirection comes at a cost. It decreases the performance of the code especially compared to structured mesh codes.

A sketch of how cell-centred values of the unstructured mesh are addressed by the face owner-neighbor addressing mechanism is shown in figure 1.6. An index pair (O_f, N_f) is defined for each face f of the cell. This index pair contains the indexes N_f, O_f of the cells Ω_{N_f} and Ω_{P_f} adjacent to the face f. The face-adjacent cell that has the lower cell index is named the *owner cell* (owner) O_f, and the cell with the higher index is the so-called *neighbor cell* (neighbor). The index b marks the boundary face of cell 1, and the boundary face only has an owner cell: the cell 1 itself.

Additional addressing, such as cell-cells and point-cells, is also stored by the unstructured mesh in OpenFOAM. The additional addressing can be used to construct numerical methods that are different from the unstructured FVM and require different connectivity of mesh elements.

Time is considered as an additional dimension of the solution domain, and
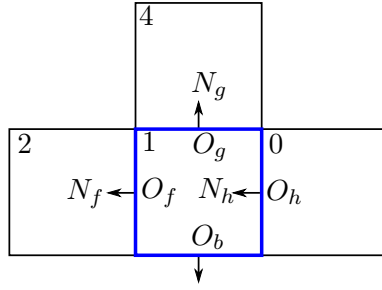
Figure 1.6: Owner-neighbor addressing for an example cell (blue frame) with cell label 1 internal face labels f, g, h, and the boundary face label b.

> **INFO**
>
> Time is discretized as another dimension of the solution domain.

the time interval $[t^0, t^E]$, where t^0, t^E are the start and end of the simulation, is discretized as a partition $[t^0, t^1, t^2, \ldots, t^{n-1}, t^n, t^{n+1}, \ldots, t^E]$, such that $t^{n-1} < t^n < t^{n+1}$ and the difference $t^{n+1} - t^n$ is called a *time step* δt, t^{n-1} is the previous point in the time partition (previous time), t^n is the current time and t^{n+1} is the next time.

1.3.2 Equation discretization

Once the domain is discretized into finite volumes, approximations are applied on the terms in the mathematical model that transform the differential operators ($\partial_t, \nabla, \nabla\cdot$) into discrete differential operators in equation (1.1). The discrete differential operators of the unstructured FVM are represented as linear combinations of average cell-centered values (averaged by equation (1.7)), or in in other words, they are linear algebraic equations with cell-centered values as dependent variables. Because the discretization process of the PDE for a cell Ω_c represents the PDE in the discrete form as a linear combination of the cell-centered values of its neighbors, the neighbors influence the solution in the cell Ω_c. Because this is done for each cell Ω_c, the discretization results in the assembly of a global algebraic linear equation system based on the mesh $\{\Omega_c\}_{c \in C}$ and $\{\phi_c(t^n)\}_{c \in C}$, that is then solved for $\{\phi_c(t^{n+1})\}_{c \in C}$. The next section describes how this is achieved using unstructured FVM equation

discretization.

Alternative descriptions of the unstructured FVM are available in [6] and [2]. Publicly available descriptions of the unstructured FVM in Open-FOAM can be found in [3, 5, 12, 9], among others.

All terms of the equation (1.1) need to be discretized in order to obtain the algebraic equation. The numerical method must be consistent (see [1]): as the sizes of the cells are reduced, the discrete (algebraic) mathematical model must approach the exact mathematical model. Or in other words: as described by Ferziger and Perić [1], refining the computational domain infinitely and solving the discretized model on this spatial discretization leads to the solution of the mathematical model consisting of PDEs. To obtain the discrete model, equation (1.1) is integrated over the cell Ω_c:

$$\int_{\Omega_c} \frac{\partial \phi}{\partial t} \, dV + \int_{\Omega_c} \nabla \cdot (\mathbf{U}\phi) \, dV - \int_{\Omega_c} \nabla \cdot \Gamma \nabla \phi \, dV = \int_{\Omega_c} S(\phi) \, dV \quad (1.8)$$

Recall the second-order accurate volume average associated to the centroid of Ω_c given by equation (1.7)

$$\phi_c \approx \frac{1}{|\Omega_c|} \int_{\Omega_c} \phi_l(\mathbf{x}_c) \, dV + O(||\mathbf{x} - \mathbf{x}_c||_2^2).$$

The temporal term from equation (1.8) is discretized using equation (1.7), and using a shorthand notation $f(\mathbf{x}_c) = f_c$ for cell-centered values, which leads to the *spatialy* second-order accurate discretization

$$\int_{\Omega_c} \frac{\partial \phi}{\partial t} \, dV = \left(\frac{\partial \phi}{\partial t} \right)_c |\Omega_c| + O(||\mathbf{x}_c - \mathbf{x}||_2^2). \quad (1.9)$$

The temporal term $\partial_t \phi_c$ can then be approximated using finite differences. In OpenFOAM, *temporally* first-order accurate backward Euler, or second-order accurate backward difference (BDS2) scheme can be used, namely

$$\left(\frac{\partial \phi_c}{\partial t} \right)_c^{n+1} = \frac{\phi_c^{n+1} - \phi_c^n}{\delta t} + O(\delta t), \quad (1.10)$$

$$\left(\frac{\partial \phi_c}{\partial t} \right)_c^{n+1} = \frac{3\phi_c^{n+1} - 2\phi_c^n + \phi_c^{n-1}}{2\delta t} + O(\delta t^2), \quad (1.11)$$

where $n + 1$ is the new time step, n is the current time step, and $n - 1$ is the previous time step. Finally, using for example the second-order

accurate Backward Differencing Scheme (BDS2) given by equation (1.11), the temporal term is discretized as

$$
\int_{\Omega_c} \left(\frac{\partial \phi}{\partial t} \right)^{n+1} dV = |\Omega_c| \frac{3\phi_c^{n+1} - 2\phi_c^n + \phi_c^{n-1}}{2\delta t} + \\
+ O(\|\mathbf{x}_c - \mathbf{x}\|_2^2) + O(\delta t^2) \tag{1.12}
$$

The divergence (advection) term $\nabla \cdot (\mathbf{U}\phi)$ in equation (1.8) is discretized using the divergence theorem:

$$
\int_{\Omega_c} \nabla \cdot (\mathbf{U}\phi) \, dV = \int_{\partial \Omega_c} \phi \mathbf{U} \cdot \mathbf{n} \, do. \tag{1.13}
$$

The boundary $\partial \Omega_c$ of Ω_c, used on the r.h.s. of equation (1.13) is a union surfaces (*faces*) that are bounded by line segments (*edges*), namely

$$
\partial \Omega_c = \cup_{f \in F_c} S_f, \tag{1.14}
$$

where F_c is the index set of the faces S_f of the cell Ω_c, and the integral on the r.h.s. of equation (1.13) can be written as

$$
\int_{\Omega_c} \nabla \cdot (\mathbf{U}\phi) \, dV = \sum_{f \in F_c} \int_{S_f} \phi \mathbf{U} \cdot \mathbf{n} \, do. \tag{1.15}
$$
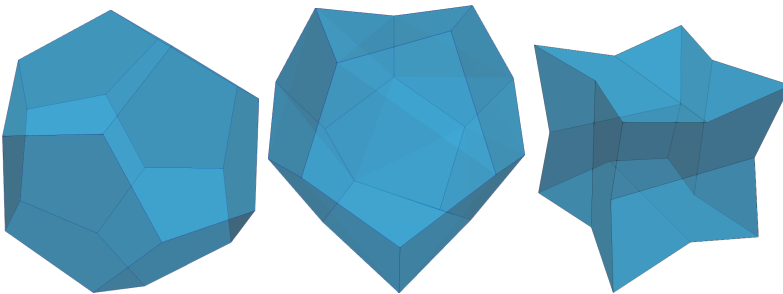
(a) Regular dodecahe- (b) Non-convex dodec- (c) Endo-dodecahedron
dron cell.   ahedron cell.   cell.

Figure 1.7: Example dodecahedron cells: convex dodecahedron, non-convex dodecahedron with non-planar faces, non-convex dodecahedron with planar non-convex faces.

Faces S_f of the cell Ω_c are planar polygons if Ω_c is a convex polyhedron (tetrahedron, cube, rectangular cuboid, etc.), as the dodecahedron shown for example in figure 1.7a. Alternatively, Ω_c can be a "*generalized polyhedron*": a non-convex volume bounded by faces that can either be non-planar, or planar and non-convex. For example, the non-convex dodecahedron in figure 1.7b is non-convex, simply because its faces are not-planar. The endo-dodecahedron shown in figure 1.7c is non-convex because its faces, although planar, are non-convex. Polyhedral mesh generation algorithms often create non-planar polyhedrons such as the one shown in figure 1.7b. The faces S_f of this polyhedron are *nonlinear ruled surfaces* that are bounded by line segments (mesh edges). This is clearly visible for the top face of the non-planar polyhedron in figure 1.7b. Endo-dodecahedron is not that important for polyhedral mesh generation, although it can theoretically appear in a dodecahedral mesh. To avoid introducing special-case handling in the discretization, all cells Ω_c are considered as general polyhedrons in OpenFOAM. Cell-faces are all linearized by using the centroid of each polygonal face to decompose the face into a set of triangles. This of course does not introduce errors when the cell Ω_c really is a convex polyhedron. If Ω_c is a generalized polyhedron with non-planar faces, the triangulation of its faces using the centroid-based triangulation introduces approximation errors on the r.h.s. of equation (1.15).

To further discretize equation (1.15), averaging of ϕ is done for each face S_f, by applying a $2D$ equivalent of the averaging given by equation (1.7), at $\mathbf{x}_f$, the centroid of the face S_f, namely

$$\phi_f = \frac{1}{|S_f|} \int_{S_f} \phi \, ds + O(||\mathbf{x} - \mathbf{x}_f||_2^2), \tag{1.16}$$

where $\phi_f = \phi(\mathbf{x}_f)$. Inserting equation (1.16) into equation (1.15) leads to

$$\int_{\Omega_c} \nabla \cdot (\mathbf{U}\phi) \, dV = \sum_{f \in F_c} \phi_f \mathbf{U}_f \cdot \mathbf{S}_f + O(||\mathbf{x} - \mathbf{x}_f||_2^2), \tag{1.17}$$

a spatially second-order accurate discretization of the divergence term given by equation (1.1). In OpenFOAM, a so-called *collocated* discretization is used: all dependent variables in the linear algebraic equation system are stored at cell centers. Therefore, the face-centered averages on the r.h.s. of equation (1.17) are expressed using the values stored at the centers of two face-adjacent cells for each face S_f.

INFO

Face-centered averages in unstructured FVM are, equivalently to cell-centered values, second-order accurate.
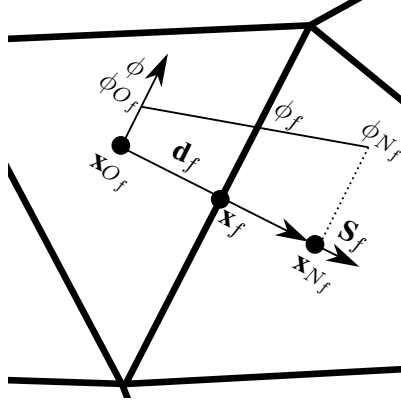
Figure 1.8: Central differencing interpolation scheme applied on an *orthogonal* triangular mesh.

Consider the schematic representation of a face f and its two adjacent owner and neighbor cells O_f and N_f on a $2D$ triangular mesh shown in figure 1.8. If the face-centered value ϕ_f is expressed using the linear interpolation (Central Differencing Scheme (CDS)), we have

$$\phi_f = \phi_{O_f} + \frac{\phi_{N_f} - \phi_{O_f}}{\|\mathbf{d}_f\|_2} \|\mathbf{x}_f - \mathbf{x}_{O_f}\|_2 \tag{1.18}$$

$$= w_f \phi_{N_f} + (1 - w_f)\phi_{O_f}, \tag{1.19}$$

where $\mathbf{d}_f = \mathbf{x}_{N_f} - \mathbf{x}_{O_f}$, $\frac{1}{\|\mathbf{d}_f\|_2}$ is the so-called *delta coefficient* in Open-FOAM, and w_f calculated with the CDS scheme is the *linear* face coefficient

$$w_{f,CDS} = \frac{\|\mathbf{x}_f - \mathbf{x}_{O_f}\|_2}{\|\mathbf{d}_f\|_2}. \tag{1.20}$$

The face coefficient $w_f \in [0, 1]$ can be calculated differently. The calculation of w_f determines the order of accuracy of the discretized divergence term and impacts the numerical stability of the approximated solution as well. Alternatively, w_f can be determined using the upwind scheme, *for the convective term*, which defines w_{N_f}, w_{O_f} using the value stored at

the center of the *upwind* (upstream) cell with respect to f,

$$w_{f,upwind} = \begin{cases} 1 & F_f > 0 \\ 0 & \text{otherwise,} \end{cases}$$

(1.21)

where F_f is the *scalar volumetric flux* at $\mathbf{x}_f$

$$F_f = \mathbf{U}_f \cdot \mathbf{S}_f.$$

(1.22)

If the velocity $\mathbf{U}$ is prescribed, it is not a dependent variable in equation (1.1), and it is expressed in equation (1.17) exactly like ϕ_f, using either a linear interpolation (CDS) or some other combination of the face-neighbor values in equation (1.19).

Computing the face-centered average as a combination of cell-centered values from adjacent cells (e.g. by CDS equation (1.18)), as well as the scalar volumetric flux from equation (1.22), results in the discrete divergence operator $\nabla_c \cdot (\cdot)$ as

$$\nabla_c \cdot (\mathbf{U}\phi) := \int_{\Omega_c} \nabla \cdot (\mathbf{U}\phi)\, dv \approx \sum_{f \in F_c} F_f[w_f \phi_{N_f} + (1 - w_f)\phi_{O_f}]$$

(1.23)

where F_c is the index set of all faces that belong to the cell Ω_c.

Up to this point, no assumption was made about the orientation of normal vectors of the cell boundary $\partial\Omega_c$, or its discrete counterpart, namely the orientation of the $\mathbf{S}_f$ normal area vectors of the faces F_c of the cell Ω_c. However, OpenFOAM does not discretize the divergence/diffusion differential operator terms from equation (1.1) (equation (1.8)) using sums over all faces of the cell (index set F_c), nor are the normal vectors of the cell boundary consistently oriented (only outwards, or only inwards) in OpenFOAM for each cell. For each face center, a unique vector $\mathbf{S}_f$ is defined. This prevents the consistency of the $\mathbf{S}_f$ normals, because for one of the cells adjacent to the face f, $\mathbf{S}_f$ then must be oriented *inward*.

The discrete divergence operator $\nabla_c \cdot (\cdot)$ is implemented in OpenFOAM using owner-neighbor addressing. The $\mathbf{S}_f$ vector shown in figure 1.8 is the surface area normal vector that always points from the owner cell

O_f to the neighbor cell N_f (see section 1.3.1, figure 1.6). This is a very important implementation aspect of the unstructured FVM in OpenFOAM. The divergence theorem assumes *a consistent orientation of the normal* of the boundary $\partial\Omega_c$ in equation (1.13): the normal **n** can either be oriented outward or inward with respect to the cell Ω_c, and the convention on the outward normal vector orientation is used in OpenFOAM. The discretization of the divergence term by equation (1.23) results in a sum over all faces of the cell Ω_c. If the discretization would be performed *for each cell*, enforcing the consistency of the normal orientation would be complicated. For example, for the cell 1 in figure 1.6, a single face area normal vector is oriented inwards because of the owner-neighbor addressing. Defining a unique normal vector $\mathbf{S}_f$ any other way would still make this vector point from one cell and into another cell, effectively pointing into one of the cells. The owner-neighbor addressing is used for the domain discretization covered in section 1.3.2 because it uniquely determines the normal orientation based on the indexes of face-adjacent cells: owner cell has a smaller index than the neighbor cell.

So, the question is: How to perform the discretization of the divergence term given by equation (1.23) if some faces of the cell (cell 1 in figure 1.6) are oriented outward and some inward?

Because the $\mathbf{S}_f$ vector always points from the cell with the lower index O_f (face-owner, owner-cell, owner) into the cell with the higher index N_f (face-neighbor, neighbor-cell, neighbor), the contributions to the sum on the r.h.s. of equation (1.23) *are added to the owner of the face, and subtracted from the neighbor of the face f, for each face f in the mesh..* To achieve this, two index-sets are introduced: the *owner* (owner-cell) and *neighbor* (neighbor-cell) index sets, namely

$$O := \{O_f : O_f < N_f \text{ for each face } f \text{ in the mesh } \{\Omega_c\}_{c\in C} \}, \quad (1.24)$$
$$N := \{N_f : O_f < N_f \text{ for each face } f \text{ in the mesh } \{\Omega_c\}_{c\in C} \}. \quad (1.25)$$

Index sets O, N given by equation (1.24) and (1.25) contain respectively the labels of owner and neighbor cells *for each face in the mesh*, instead of *for each face of the cell* as does the index set F_c in equation (1.23). This makes it possible to perform the same calculation as in equation (1.23) even when the normal vectors $\mathbf{S}_f$ of the cell are not all

pointing outwards or inwards:

for each face index $f \in F$

$$\nabla_{O_f} \cdot (\mathbf{U}\phi) = \nabla_{O_f} \cdot (\mathbf{U}\phi) + [(w_f \phi_{N_f} + (1 - w_f)\phi_{O_f}]F_f$$
$$\nabla_{N_f} \cdot (\mathbf{U}\phi) = \nabla_{N_f} \cdot (\mathbf{U}\phi) - [(w_f \phi_{N_f} + (1 - w_f)\phi_{O_f}]F_f \qquad (1.26)$$

where F is the index set of all faces in the mesh, O_f is the index of the face-owner cell from O given by equation (1.24), N_f is the index of the face-neighbor cell N given by equation (1.25), and $\nabla \cdot$ is the discrete divergence operator given by equation (1.23).

INFO

The discretization of the divergence operator by equation (1.23) is helpful for understanding the unstructured Finite Volume Method, while the actual computation in OpenFOAM is performed using equation (1.26).

INFO

Owner and neighbor index sets can be accessed from the `mesh` using `owner()` and `neighbour()` member functions of the `fvMesh` class.

As described in section 1.3.1 and shown in figure 1.6 for cell 1, boundary faces only have one owner-cell. Because this face has no neighbor-cell, it will not have a contribution $-F_f w_f \phi_{N_f}$ in equation (1.26). To avoid checking if the face belongs to the mesh boundary, boundary faces (face b in figure 1.6) are stored separately from internal faces in OpenFOAM.

The Laplace (diffusion) term from equation (1.1) is discretized similarly to the divergence (advection) term. The discretization starts with the integral over Ω_c and the application of the divergence theorem, using the shorthand notation $\Gamma_f := \Gamma(\mathbf{x}_f), \nabla\phi(\mathbf{x}_f) := (\nabla\phi)_f$, which leads to

$$\int_{\Omega_c} \nabla \cdot \Gamma \nabla \phi \, dV = \sum_{f \in F_c} \Gamma_f (\nabla\phi)_f \cdot \mathbf{S}_f + O(\|\mathbf{x} - \mathbf{x}_f\|_2^2), \qquad (1.27)$$

where the spatial second-order accuracy error term results from the face-centered averaging given by equation (1.16).

The next step in the discretization of the Laplace term is the discretization of the gradient $(\nabla\phi)_f$. The face-centered gradient $(\nabla\phi)_f$

in equation (1.27) is not interpolated from face-adjacent cell centered gradients. Instead, Taylor series from the face center $\mathbf{x}_f$ to the owner, and to the neighbor cell center are used. To simplify the expressions, the following shorthand notation is introduced. For face-centered values, $\mathbf{x}_{O_f} - \mathbf{x}_f := \mathbf{x_f x}_{O_f}$ and $\phi_{O_f} := \phi(\mathbf{x}_{O_f})$ are used (equivalently for N_f, f), in addition to the notation for the tensor product of a vector $\mathbf{a}$:

$$\mathbf{a}^n := \underbrace{\mathbf{a} \otimes \mathbf{a} \otimes \mathbf{a}}_{n}, n \in \mathbb{N}^+.$$

Using this notation, Taylor series from the face center, the face owner and neighbor are respectively given as

$$\phi_{O_f} = \phi_f + (\nabla\phi)_f \cdot \mathbf{x_f x_{O_f}} + \frac{1}{2}(\nabla^2\phi)_f : \mathbf{x_f x_{O_f}}^2 + \frac{1}{6}(\nabla^3\phi)_f :: \mathbf{x_f x_{O_f}}^3 + \ldots,$$
(1.28)

$$\phi_{N_f} = \phi_f + (\nabla\phi)_f \cdot \mathbf{x_f x_{N_f}} + \frac{1}{2}(\nabla^2\phi)_f : \mathbf{x_f x_{N_f}}^2 + \frac{1}{6}(\nabla^3\phi)_f :: \mathbf{x_f x_{N_f}}^3 \ldots.$$
(1.29)

Subtracting equation (1.28) from equation (1.29), leads to

$$(\nabla\phi)_f \cdot (\mathbf{x_f x_{N_f}} - \mathbf{x_f x_{O_f}}) = \phi_{N_f} - \phi_{O_f} - \frac{1}{2}(\nabla^2\phi)_f : (\mathbf{x_f x_{N_f}}^2 - \mathbf{x_f x_{O_f}}^2)$$

$$- \frac{1}{6}(\nabla^3\phi)_f :: (\mathbf{x_f x_{N_f}}^3 - \mathbf{x_f x_{O_f}}^3) + \ldots. \quad (1.30)$$

On *equidistant meshes*, similar to the mesh shown in figure 1.8, face centers $\mathbf{x}_f$ split the vector $\mathbf{d}_f = \mathbf{x}_{N_f} - \mathbf{x}_{O_f}$ into two equal parts, such that

$$\mathbf{x_f x_{N_f}} = -\mathbf{x_f x_{O_f}} = \frac{1}{2}\mathbf{d}_f. \tag{1.31}$$

Inserting equation (1.31) into equation (1.30), multiplying the result with $\cdot\mathbf{d}_f$, and dividing by $\|\mathbf{d}_f\|_2^2$, cancels out the second-order term in equation (1.30), i.e.

$$(\nabla\phi)_f = \frac{\phi_{N_f} - \phi_{O_f}}{\|\mathbf{d}_f\|_2}\hat{\mathbf{d}}_f - \frac{1}{2}(\nabla^2\phi)_f : \underbrace{((0.5\mathbf{d}_f)^2 - (0.5\mathbf{d}_f)^2)}_{0} \cdot \frac{\mathbf{d}_f}{\|\mathbf{d}_f\|_2^2}$$

$$- \frac{1}{24}(\nabla^3\phi)_f :: (\mathbf{d}_f^3) \cdot \frac{\mathbf{d}_f}{\|\mathbf{d}_f\|_2^2} + \ldots, \tag{1.32}$$
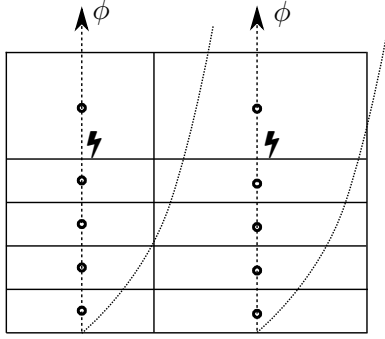
Figure 1.9: Loss in accuracy of the gradient discretization in a boundary layer, caused by the aspect ratio different from 0.5 for faces ⚡ with a strong change in ϕ, in the vertical direction.

which leads to the final discretization of the *face-centered gradient* as

$$(\nabla\phi)_f \approx \frac{\phi_{N_f} - \phi_{O_f}}{\|\mathbf{d}_f\|_2}\hat{\mathbf{d}}_f - \frac{1}{24}(\nabla^3\phi)_f :: (\hat{\mathbf{d}}_f^3) \cdot \hat{\mathbf{d}}_f \|\mathbf{d}_f\|_2^2. \qquad (1.33)$$

Equation (1.33) demonstrates second-order accuracy of the *face-centered gradient* in terms of $\|\mathbf{d}_f\|_2^2$, but *only on equidistant meshes that satisfy equation* (1.31), and for *sufficiently regular* ϕ that can be expanded into Taylor series. If equation (1.31) does not hold, the largest error term in equation (1.33) contains a contribution $\frac{1}{2}(\nabla^2\phi)_f : (\mathbf{x_f x_{N_f}}^2 - \mathbf{x_f x_{O_f}}^2)$ from equation (1.30), which makes the discretization first-order accurate.

INFO

The *aspect ratio* $\frac{\|\mathbf{x_f x_{N_f}}\|_2}{\|\mathbf{x_{O_f} x_{N_f}}\|_2}$ should ideally be 0.5 in regions where strong changes in ϕ are expected, because then equation (1.31) holds and the second-order accurate discretization by equation (1.33) is achieved. Stronger mesh grading can be used where $(\nabla^2\phi)_f$ is expected to be insignificantly small and have virtually no impact on the second-order convergence of the solution.

Consider figure 1.9: ϕ varies strongly in the vertical direction, and a boundary layer captures the vertical change in ϕ. However, the boundary layer ends before ϕ stops to vary strongly, and the gradient discretization
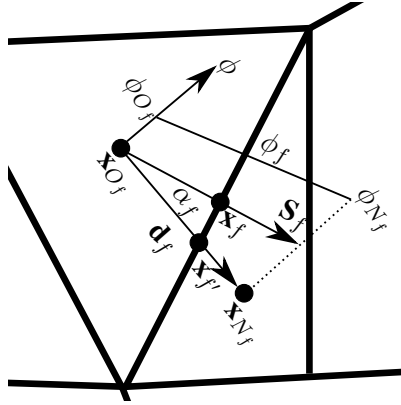
Figure 1.10: A $2D$ *nonorthogonal* mesh.

on faces denoted with ⚡ introduces first-order error terms, because of an aspect ratio that is different than 0.5. On the other hand, ϕ does not change at all horizontally in figure 1.9, so the aspect ratio that differs from 0.5 has virtually no impact on the solution accuracy at these faces.

With the second order accurate discretization of $(\nabla \phi)_f$ using equation (1.33), the discretization of the Laplace term is finalized by inserting equation (1.33) into equation (1.27), wich results in a second-order accurate FVM discretization of the Laplace (diffusion) term:

$$\int_{\Omega_c} \nabla \cdot \Gamma \nabla \phi \, dV \approx \sum_{f \in F_c} \frac{\Gamma_f (\hat{\mathbf{d}}_f \cdot \mathbf{S}_f)}{\|\mathbf{d}_f\|_2} (\phi_{N_f} - \phi_{O_f}) \qquad (1.34)$$

The discretization of the Laplace term by equation (1.34) can be further simplified *if $\hat{\mathbf{d}}_f$ and $\hat{\mathbf{S}}_f$ are collinear* ($\hat{\mathbf{d}}_f \cdot \hat{\mathbf{S}}_f = 1$), or in other words, *if the mesh is orthogonal*, as

$$\int_{\Omega_c} \nabla \cdot \Gamma \nabla \phi \, dV \approx \sum_{f \in F_c} \frac{\Gamma_f \|\mathbf{S}_f\|_2}{\|\mathbf{d}_f\|_2} (\phi_{N_f} - \phi_{O_f}). \qquad (1.35)$$

A *nonorthogonal* mesh is shown in figure 1.10: in a nonorthogonal mesh, at the face S_f, vectors $(\hat{\mathbf{d}}_f, \hat{\mathbf{S}}_f)$ are not collinear and form a so-called *nonorthogonality angle* $\alpha_f := \angle(\hat{\mathbf{d}}_f, \hat{\mathbf{S}}_f)$. Nonorthogonality introduces an

error in equation (1.35), and this error is corrected using a discretization of the cell-centered gradient, based on the Gauss divergence theorem. A gradient at the cell centroid can be discretized as

$$\int_{\Omega_c} \nabla\phi \, dV = \int_{\partial\Omega_c} \phi \cdot \mathbf{n} \, dS = \sum_f \phi_f \cdot \mathbf{S}_f + O(h_c^2) := (\nabla\phi)_c, \quad (1.36)$$

with second-order accuracy given by face-centered averaging in equation (1.16) and linear interpolation in equation (1.18). The gradient discretized this way at the cell centroid is denoted with $(\nabla\phi)_c$. If equation (1.36) is interpolated linearly with equation (1.18) at the face centroid, namely

$$(\nabla\phi)_f = w_f(\nabla\phi)_{N_f} + (1 - w_f)(\nabla\phi)_{O_f} + O(h_f^2), \quad (1.37)$$

the face-centered gradient retains second-order accuracy from the owner and neighbor gradients given by equation (1.36), because linear interpolation is bounded, which means it maintains the bounds of the error $O(h^2)$ from N_f and O_f cells. The second-order accuracy term is simplified in equation (1.36) for brevity as $O(h^2)$: the expression is in fact more complex on general unstructured meshes. The error be roughly estimated by maximal errors, such as $\tilde{h}_c = \max_{f \in F_c} \|\mathbf{d}_f\|_2$ and $h_f = \max_{p \in F_p} \|\mathbf{x}_p - \mathbf{x}_f\|_2$, where F_p is the set of all face points. Finally, even if $h = max(\tilde{h}_c, \max_{f \in F_c}(h_f))$, we can see that the linear interpolation of the cell-centered Gauss gradient retains second-order accuracy under conditions given by equation (1.31).

Face-centered gradient approximated with equation (1.36) is not used in equation (1.27) for the discretization of the Laplace term, because equation (1.37) can cause so-called *checkerboarding*. Checkerboarding is the inability of the discretization to compute $(\nabla\phi)_c$ due to the artificial cancellation present in equation (1.37) for some cases. As a simplest example, consider five $1D$ finite volume cells labeled with $(0, 1, 2, 3, 4)$, with a unit distance between them, and a ϕ distribution of $(50, 100, 50, 100, 50)$ at their cell centers. If cell-centered gradients $(\nabla\phi)_1, (\nabla\phi)_3$ are computed using equation (1.18) and (1.36), with the normal vector orientation based on owner-neighbor addressing (respectively $\mathbf{n}_{0,1} = \mathbf{n}_{1,2} = \mathbf{n}_{2,3} = \mathbf{n}_{3,4} = 1$), the resulting gradients are $(\nabla\phi)_1 = -75 + 75 = 0$ and $(\nabla\phi)_3 = -75 + 75 = 0$. These false zeros, interpolated further by equation (1.37) to compute face-centered gradients $(\nabla\phi)_{(0,1)}, (\nabla\phi)_{1,2}$, remain zero valued. Alternatively, if $(\nabla\phi)_{(0,1)}$

is computed with equation (1.37), $(\nabla\phi)_{(0,1)} = \frac{50-100}{1} = -50 \neq 0$, and the same holds for $(\nabla\phi)_{1,2}$. This example is clearly artificial, however it shows that the discretization disregards oscillations in the solution between two adjacent cells, and such oscillations, although not as regular as in this example, can actually be a part of the solution. An example of such oscillations would be the oscillations in the pressure caused by a sequence of small (under-resolved) vortices.

INFO

Face-centered gradient is not used by equation (1.37) because it can cause checkerboarding.

Linearly interpolated face-centered gradient given by equation (1.37) can be used to correct for nonorthogonality in equation (1.35). Three approaches are usually used for the decomposition of the surface area normal vector: *minimum correction*, *orthogonal correction* and *over relaxed correction* (cf. [3, section 3.3.1.3], [5, section 3.3.2], [6, section 8.6]). All three approaches decompose the surface area normal vector $\mathbf{S}_f$ into the orthogonal part $\mathbf{S}_f^{\perp}$ collinear with $\mathbf{d}_f$ and the non orthogonal part $\mathbf{S}_f^{\not\perp}$, i.e.

$$(\nabla\phi)_f \cdot \mathbf{S}_f = (\nabla\phi)_f^{\perp} \cdot \mathbf{S}_f^{\perp} + (\nabla\phi)_f^{\not\perp} \cdot \mathbf{S}_f^{\not\perp}, \qquad (1.38)$$

such that

$$\mathbf{S}_f = \mathbf{S}_f^{\perp} + \mathbf{S}_f^{\not\perp}. \qquad (1.39)$$

Different approaches for computing $\mathbf{S}_f^{\perp}$ are shown in figure 1.11 and $\mathbf{S}_f^{\not\perp}$ is computed from equation (1.39). The orthogonal contribution to the gradient in equation (1.40), $(\nabla\phi)_f^{\perp}$, is evaluated using equation (1.33), which involves two face-adjacent cell averages and is discretized implicitly. The nonorthogonal contribution $(\nabla\phi)_f^{\not\perp}$ is discretized explicitly.

minimum correction

$$\mathbf{S}_f^{\perp} = (\mathbf{S}_f \cdot \hat{\mathbf{d}}_f)\hat{\mathbf{d}}_f$$

orthogonal correction

$$\mathbf{S}_f^{\perp} = \|\mathbf{S}_f\|_2 \hat{\mathbf{d}}_f$$

over-relaxed correction

$$\mathbf{S}_f^{\perp} = \left(\frac{\|\mathbf{S}_f\|_2}{cos(\alpha_f)}\right)\hat{\mathbf{d}}_f$$
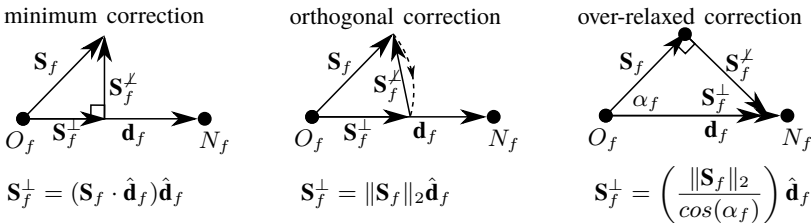
Figure 1.11: Nonorthogonality corrections.

Explicit discretization of $(\nabla\phi)_f^{\not\perp}$ in equation (1.40) results in

$$(\nabla\phi)_f(t^{n+1}) \cdot \mathbf{S}_f = (\nabla\phi)_f^{\perp}(t^{n+1}) \cdot \mathbf{S}_f^{\perp} + (\nabla\phi)_f^{\not\perp}(t^n) \cdot \mathbf{S}_f^{\not\perp}, \qquad (1.40)$$

which does not hold, because the orthogonal and nonorthogonal contributions evaluated at different time steps. To correct the discrepancy between t^{n+1} and t^n in equation (1.40), nonorthogonal corrections in OpenFOAM are performed using an additional fixed number M of additional iterations within each time step, namely

$$(\nabla\phi)_f(t^{n+1})\cdot\mathbf{S}_f = (\nabla\phi)_f^{\perp}(t^{n+1})\cdot\mathbf{S}_f^{\perp}+(\nabla\phi)_f^{\not\perp}(t^m)\cdot\mathbf{S}_f^{\not\perp}, \ m = 1, 2, 3, \ldots M,$$
$$(1.41)$$

in the hope that after M iterations $(\nabla\phi)_f^{\not\perp}(t^M) = (\nabla\phi)_f^{\not\perp}(t^{n+1})$. Iterations for nonorthogonal correction play a vital role in the pressure-velocity coupling algorithm, where the Laplace operator is used in the Poisson equation for the pressure. That is why configuration files for pressure-velocity coupling algorithms in OpenFOAM have an appropriate entry.

> **INFO**
>
> In OpenFOAM, the number M of iterations for the nonorthogonality correction used in the pressure Poisson equation can be set in `system/fvSolution` configuration file.

> **INFO**
>
> Information about the nonorthogonality angle in an unstructured Open-FOAM mesh is reported by the `checkMesh` utility.

> **INFO**
>
> In the general case if the mesh is nonorthogonal, it is also non-equidistant, in which case equation (1.31) does not hold and the gradient approximation at face centers becomes first-order accurate.

Generally, mesh nonorthogonality displaces the point of intersection between $\mathbf{x}_{O_f}\mathbf{x}_{N_f}$ and Sf (point $\mathbf{x}_{f'}$ in figure 1.10) from the face centroid $\mathbf{x}_f$, used for the averaging over S_f in equation (1.16). Equation (1.16) expects the face-average to be associated with the face centroid to ensure second-order accuracy and since $\mathbf{x}_f'$ is where the interpolation actually takes place, the so-called *mesh skewness error* is introduced.

If the face centroid $\mathbf{x}_f$ used for averaging of ϕ_f over the face S_f in equation (1.16) does not correspond to the intersection point $\mathbf{x}_{O_f}\mathbf{x}_{N_f} \cap S_f$, *mesh skewness error is introduced.*

Different methods for addressing the skewness error are proposed in the scientific literature, but no skewness correction method is currently vailable per default in OpenFOAM, so skewness errors are not addressed here in detail. Additional information on nonorthogonality and skewness errors in the context of the unstructured FVM can be found in [3, 5, 6].

Temporal discretization combines the discretization of the temporal, convection (divergence), and diffusion (Laplace) term described so far. Reorganizing the discretized scalar transport equation (1.1) using discretized operators described so far, we can write

$$\left(\frac{\partial \phi}{\partial t}\right)_c = \frac{1}{|\Omega_c|}\left[\sum_f D_f(\nabla\phi)_f \cdot \mathbf{S}_f - \sum_f \mathbf{U}_f\phi_f \cdot \mathbf{S}_f\right] + S(\phi_c). \quad (1.42)$$

Expanding ϕ_c in time using Taylor series, with $\delta t = t^{n+1} - t^n$ as the time step, gives

$$\phi_c^{n+1} = \phi_c^n + \left(\frac{\partial \phi}{\partial t}\right)_c^n \delta t + O(\delta t^2), \quad (1.43)$$

$$\left(\frac{\partial \phi}{\partial t}\right)_c^n = \frac{\phi_c^{n+1} - \phi_c^n}{\delta t} - O(\delta t), \quad (1.44)$$

and

$$\phi_c^n = \phi_c^{n+1} - \left(\frac{\partial \phi}{\partial t}\right)_c^{n+1} \delta t + O(\delta t^2), \quad (1.45)$$

$$\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1} = \frac{\phi_c^{n+1} - \phi_c^n}{\delta t} + O(\delta t). \quad (1.46)$$

Discretizations given by equation (1.44) and equation (1.46) are equivalent and equally first-order accurate. However, OpenFOAM uses implicit Euler discretization given by equation (1.46), because it is unconditionally stable for linear equations with smooth initial conditions. Inserting

equation (1.46) into equation (1.42) results in the discretization of the scalar transport equation (1.1) with the often used implicit Euler scheme that is first-order accurate in time and second-order accurate in space:

$$
\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1} = \frac{1}{|\Omega_c|}\Bigg[\sum_f D_f(\nabla \phi)_f^{n+1} \cdot \mathbf{S}_f
$$
$$
- \sum_f \mathbf{U}_f \phi_f^{n+1} \cdot \mathbf{S}_f\Bigg] + S(\phi_c^{n+1}) + O(\delta t) + O(h^2),
$$

(1.47)

where the source term $S(\phi_c)^{n+1}$ is either evaluated at the old time step t^n, or extrapolated in ϕ and time using linearization. Additional information about source term linearization can be found in [7]. Alternatively, inserting equation (1.44) into equation (1.42) and summing the resulting equation with equation (1.47), results in the Crank-Nicolson scheme

$$
\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1} = \frac{0.5}{|\Omega_c|}\Bigg[\sum_f D_f(\nabla \phi)_f^{n+1} \cdot \mathbf{S}_f +
$$
$$
+ \sum_f D_f(\nabla \phi)_f^{n} \cdot \mathbf{S}_f - \sum_f \mathbf{U}_f \phi_f^{n+1} \cdot \mathbf{S}_f
$$
$$
- \sum_f \mathbf{U}_f \phi_f^{n} \cdot \mathbf{S}_f + S(\phi_c^{n}) + S(\phi_c^{n+1})\Bigg]
$$
$$
+ O(\delta t^2) + O(h^2)
$$

(1.48)

that is second-order accurate in time as well, because the terms $-O(\delta t)$ and $+O(\delta t)$ cancel out in the summation of equation (1.44) and equation (1.46), respectively, leaving $O(\delta t^2)$ as the leading truncation term. Verifying this is left to the reader as a short exercise. In OpenFOAM, the coefficient 0.5 in the Crank-Nicolson scheme is made variable by

Listing 1 Example temporal discretization entry in the system/fvSchemes configuration (dictionary) file.

```
ddtSchemes
{
    default       CrankNicolson 0.5;
}
```

grouping together implicit and explicit terms, namely

$$
\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1} = \frac{1}{|\Omega_c|}\left[W_{CN}\left(\sum_f D_f(\nabla\phi)_f^{n+1}\cdot\mathbf{S}_f - \sum_f \mathbf{U}_f\phi_f^{n+1}\cdot\mathbf{S}_f\right.\right.
$$

$$
\left. + S(\phi_c^{n+1})\right) + (1 - W_{CN})\left(\sum_f D_f(\nabla\phi)_f^n\cdot\mathbf{S}_f\right.
$$

$$
\left.\left. - \sum_f \mathbf{U}_f\phi_f^n\cdot\mathbf{S}_f + S(\phi_c^n)\right)\right] + O(\delta t^2) + O(h^2)
$$

$$(1.49)$$

When $W_{CN} = 0.5$, equation (1.48) is recovered from equation (1.49). If equation (1.46) is used for $\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1}$, $W_{CN} = 1$ results in the first-order Euler implicit method given by equation (1.47).

Temporal integration schemes are chosen in the simulation case inside the system/fvSchemes dictionary of a simulation folder, as shown for example in listing 1, where $W_{CN} = 0.5$ is used. Of course, source terms in equation (1.49) are linearized functions of ϕ, that ensure a linear algebraic system is obtained by equation discretization, that can be solved using a linear solver.

A system of linear algebraic equations is constructed by equation discretization, solved for ϕ_c^{n+1} in each cell Ω_c of the mesh Ω_D. Consider, for example the discretization given by equation (1.48) on an orthogonal mesh. The terms $(\nabla\phi)_f^{n+1}$ and ϕ_f^{n+1} will, relying respectively on equation (1.35) and equation (1.23), introduce the values from the face-neighbors of the cell $_c$ into equation (1.48), from the time steps $n, n+1$. Other terms, such as $\left(\frac{\partial \phi}{\partial t}\right)_c^{n+1}$ only contain values ϕ_c from the current n and the new time step $n+1$. Separating contributions from the cell $_c$ and the set of its face-neighbors N_c, results in a linear algebraic equation,

generally written as

$$a_c \phi_c^{n+1} + \sum_{N \in N_c} a_N \phi_N^{n+1} = S_c. \tag{1.50}$$

Implicit discretization, as the one from equation (1.48), generates a linear algebraic equation (1.50) for each Ω_c in the mesh. Equations 1.50 are coupled with each other, because in the equation for each cell Ω_c, there are contributions from the neighboring cells. However, because the contributions are only there from the neighboring cells that share a face with the cell Ω_c, the system of equations will only have a few non-zero coefficients, and the resulting linear system will be sparse. Solving the sparse linear algebraic system results in the calculation of every ϕ_c value in the mesh, for the new time step.

Boundary conditions are required in the discretization whenever a face-centered value belongs to a cell face that is a part of the mesh boundary. The so-called *boundary faces* do not have neighbor cells, whose values take part in the discretization. Instead, what will be defined is often:

- a value is prescribed by a *Dirichlet*, or *fixed-value* boundary condition,
- a gradient is prescribed by a *Neumann* boundary condition,
- or a linear combination of a value and a gradient is prescribed by a *mixed*, or *Robin* boundary condition.

Boundary conditions are necessary at boundary faces regardless of the implicit or explicit nature of the discretization, interpolation scheme used for ϕ_f, or the gradient discretization scheme used for $(\nabla \phi)_f$. Boundary faces are highlighted by the thick line in figure 1.12.

For example, assembling the linear algebraic equation equation (1.50) using equation (1.48), summation used by the discretized operators will encounter boundary faces in some cells. Let us consider the advection operator $\nabla \cdot (\mathbf{U}\phi)$ and assume such a boundary face is marked with b: then it becomes clear that $\phi_b^{n+1}\mathbf{U}_b \cdot \mathbf{S}_b$ should be added to the sum in the discretized advection term. For internal faces, this value would be interpolated between cell-centered values of the cells adjacent to the face b. However, there is only one cell next to a boundary face, and this cell is - by definition - the owner of that face. For the `fixedValue` boundary condition, the procedure is simple: the boundary condition prescribes the
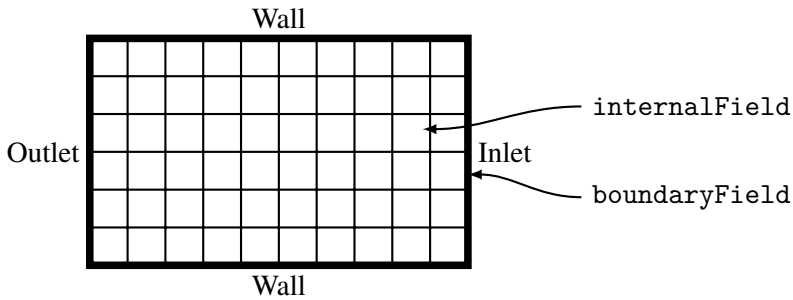
Figure 1.12: Example of a simple 2D channel flow with the inlet on the right side and the outlet on the left hand side. The other remaining two boundaries are assumed to be walls.

value ϕ_b for the property ϕ as well $\mathbf{U}_b$ for the velocity $\mathbf{U}$. This is sufficient for a fixed-value boundary condition, however, it complicates the implementation of boundary conditions significantly. The vast majority of faces in the mesh are internal faces, not boundary faces. Therefore it would make no sense to keep the internal and boundary faces somehow mixed together, and classified, such that the discretization could "ask each face" if it belongs to the boundary or not, and additionally which boundary condition is prescribed for this face. To avoid this complication, the faces of the mesh Ω_D are separated into internal faces, and boundary faces. Boundary faces are further grouped together into *boundary patches* in OpenFOAM: sub-sets of the boundary mesh where the same boundary conditions are applied. This makes sense, because simulated processes usually have surfaces that are exposed to different conditions. Considering heat transfer as an example: some surfaces may be isolated, others may be cooled by streaming air or liquid sprays, some surfaces will be adjacent to bodies made out of different material. Consequently, as shown in figure 1.12, fields in OpenFOAM are separated into *internal* (cell-centered) fields, and *boundary* face-centered fields, separated into *patch fields* that correspond to specific boundary patches (boundary conditions). The discretization in OpenFOAM is significantly simplified by separation of faces and fieds respectively into internal and boundary patches and fields, because the evaluation of sums in the discretized operators can be applied separately for the internal part of the domain and its boundary.

Another boundary condition is the Neumann - or "natural" - boundary

condition, which prescribes a zero gradient of the property at the domain boundary:

$$\nabla \phi_b^{n+1} = 0, \tag{1.51}$$

and this is the condition that is used to compute the value of the property at the boundary face b, using the Taylor series approximation:

$$\phi_c^{n+1} \approx \phi_b^{n+1} + \nabla \phi(\mathbf{x})^{n+1} \cdot b, \tag{1.52}$$

so the value at the boundary face takes on the value from the cell center for the zero-gradien boundary condition, namely

$$\phi_b^{n+1} \approx \phi_c^{n+1}. \tag{1.53}$$

The boundary contribution to the algebraic equation equation (1.50) for a zero-gradient boundary condition on the boundary face b will end up in the coefficient next to the cell value in the new time step: a_c in equation (1.50). In other words, the zero-gradient boundary condition will impact diagonal coefficients of the linear algebraic equation system for all cells next adjacent to the boundary patch where this condition is prescribed. Various boundary conditions are implemented in OpenFOAM and all of them either prescribe the boundary value or the gradient, or a combination of both.

Solving the system of linear algebraic equations (linear system) generated by the unstructured FVM often requires substantial computational effort, because its size is $|\Omega_D|^d$, where $|\Omega_D|$ is the number of cells in the mesh Ω_D and d is the spatial dimension. The coefficients next to $\phi_c^{n+1}, \phi_N^{n+1}$ in equation (1.50) are therefore the coefficients in the $|\Omega_D|^d$-large sparse matrix. OpenFOAM utilizes a specific format for the matrix representation, together with a set of linear solvers that are tightly coupled with the OpenFOAM matrix format. This topic is not covered here, instead, the reader is directed to [6, chapter 10] for the details on the OpenFOAM matrix format and linear solvers. Note that it is rarely necessary to make significant adjustments or modifications to linear solver configurations in OpenFOAM beyond what is configured for many of tutorials. An informative and intuitive derivation of the Conjugate-Gradient linear solver is available in [11]. Detailed background knowledge of iterative solution method methods for sparse linear systems is available in [10].

1.4 Top-level OpenFOAM structure

OpenFOAM consists of many different libraries, solver and utility applications. In order to establish some orientation around this massive and often intimidating code base, we can take a look at the contents of the root OpenFOAM directory.

Contents of the OpenFOAM directory:

applications Source code for solvers, utilities, and auxilary testing functions. Solver code is organized by function such as `/incompressible`, `/lagrangian`, or `/combustion`. Utilities are organized similarly into mesh, pre-processing, and post-processing categories among others.

bin Bash (not C++ binaries) scripts of with a broad array of functions: checking the installation (`foamInstallationTest`), executing a parallel run in debug mode, (`mpirunDebug`), generating an empty source code template (`foamNew`) or case (`foamNewCase`), etc.

doc User's Guide, the Programmer's Guide, and the Doxygen generation files.

etc Compilation and runtime selectable configuration controls flags for the entire library. Numerous installation settings are set in `/etc/bashrc` including which compiler to use, what MPI library to compile against, and where the installation will be placed (user local or system wide).

platforms Compiled binaries stored based on precision, debug flags, and processor architecture. Most installations will only have one or two sub-folders here which will be named according to the compilation type. For example, `linux64GccDPOpt` can be interpreted as follows:

linux operating system type

64 processor architecture

Gcc compiler used (Gcc, Icc, CLang)

DP float precision (double precision (DP) vs. single precision (SP))

Opt Compiler optimization or debug flag. (Optimized (Opt) vs. Debug (Debug) vs. Profile (Prof))

src The bulk of the source code of the toolkit. Contains all of the CFD library sources including finite volume discretizaton, transport models, and the most basic primitve structures such as scalars,

vectors, lists, etc. The main CFD solvers within the `applications` folder use the contents of these libraries to function.

`tutorials` Pre-configured cases for the various available solvers. The tutorials are useful for seeing how cases are set up for each solver. Some cases illustrate more complex pre-processing operations such as multi-region decomposition for solid-fluid heat transfer or Arbitrary Mesh Interface (AMI) setup.

`wmake` The bash based script, `wmake`, is a utility which configures and calls the C++ compiler. When compiling a solver or a library with `wmake`, information from `Make/files` and `Make/options` is used to include headers and link other supporting libraries. A `Make` folder is required to use `wmake`, and thus to compile most OpenFOAM code.

The OpenFOAM library is described in depth from a software design perspective in chapter 5. Different paradigms of the C++ programming language and how they are used to make OpenFOAM such a modular and powerful CFD platform are explained there.

1.5 Summary

The OpenFOAM CFD framework can often seem daunting because it demands a solid understanding of physics, numerics, and engineering from the user. OpenFOAM is open-source, and becuase of that many aspects of the solution process are openly exposed to the user (contrast that with the opacity common in commercial simulation products). Access to the source code gives the user the power to adapt things to her/his needs. However, this power comes at the cost of learning how to use the command line in the Linux Operating system, learning about configuration files that define parameters used by the unstructured Finite Volume Method, etc. Understanding this chapter is the first step towards the successful use of OpenFOAM. Understanding the unstructured Finite Volume Method in OpenFOAM is essential, not only for developing new methods but also for fully understanding why things may have went wrong in certain simulation and how to fix it. All elements of OpenFOAM described throughout the rest of the book, such as boundary conditions, discretization schemes, solver applications, etc., are based on the unstructured Finite Volume Method.

Further reading

[1] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer, 2002.

[2] Charles Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Elsevier, 2007.

[3] Jasak. "Error Analysis and Estimatino for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science, 1996.

[4] Hrvoje Jasak, Aleksandar Jemcov, and Željko Tuković. "Open-FOAM: A C++ Library for Complex Physics Simulations". In: *Proceedings of the International Workshop on Coupled Problems in Numerical Dynamics (CMND 2007)* (2007).

[5] F. Juretić. "Error Analysis in Finite Volume CFD". PhD thesis. Imperial College of Science, 2004.

[6] F Moukalled, L Mangani, M Darwish, et al. *The finite volume method in computational fluid dynamics*. Springer, 2016.

[7] Suhas Patankar. *Numerical heat transfer and fluid flow*. CRC Press, 1980.

[8] Tomasz Plewa, Timur Linde, and V Gregory Weirs. *Adaptive mesh refinement-theory and applications*. Vol. 41. Springer, 2005, pp. 3–5.

[9] Henrik Rusche. "Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions". PhD thesis. Imperial college of Science, Technology and Medicine, London, 2002.

[10] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. 2nd ed. Society for Industrial and Applied Mathematics, Apr. 2003. URL: http://www-users.cs.umn.edu/~saad/PS/all_pdf.zip.

[11] Jonathan Richard Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

[12] O. Ubbink. "Numerical prediction of two fluid system with sharp interfaces". PhD thesis. Imperial College of Science, 1997.

[13]   H. K. Versteeg and W. Malalasekra. *An Introduction to Compu-tational Fluid Dynamics: The Finite Volume Method Approach.* Prentice Hall, 1996.

[14]   H. G. Weller et al. "A tensorial approach to computational contin-uum mechanics using object-oriented techniques". In: *Computers in Physics* 12.6 (1998), pp. 620–631.

2

Geometry Definition, Meshing and Mesh Conversion

Before going into the details of this chapter, some notions have to be put into context. A *geometry* in the CFD context is essentially a three dimensional representation of the flow region. A *mesh* on the other hand can have multiple meanings, but the three dimensional *volume mesh* is typically the one considered here. There are small variations to this, for example, a *surface mesh*, which is the discretization of a surface. As could be expected for surface discretization, planar elements are used as opposed to volume elements for the volume mesh. For complex geometries, proper definition of the surface mesh can be essential.

From a CFD perspective, it is the geometry relevant to each particular flow problem, that is of interest. Thinking of an aerodynamic simulation of the flow around a car, the interior of the car is generally of no interest, as it does not contribute to the overall flow in a significant manner. Therefore, only the details on the outside of the car's body are relevant and need to be resolved sufficiently in the spatial discretization.

This chapter outlines how to create a mesh from scratch, how to convert meshes between formats, and reviews various utilities for manipulating a mesh after creation.

2.1 Geometry Definition

It is important to distinguish between the actual mesh geometry and the geometry that is genereated by a Computational Aided Design (CAD) program. Though some words on the general mesh connectivity have been spent in the previous chapter, an overview of how the actual mesh is stored in the file system is given here. There are three main directories in a standard OpenFOAM case: 0, `constant` and `system`. The first (0) stores the initial conditions for the fields, which are generally not needed for the mesh generation process. The last directory (`system`) stores settings related to the numerics and overall execution of the simulation. For this chapter the `constant` directory is under consideration as it stores the mesh including all spatial and connectivity related data. Additional details on the OpenFOAM case structure are provided in chapter 3.

As long as static meshes are used, the computational grid is *always* stored in the `constant/polyMesh` directory. Static meshes, in this context, are meshes which are not changed over the course of the simulation either though point displacement or connectivity changes. Mesh data is naturally located here because it is assumed to be constant, hence the `constant` folder. From a programming perspective it is described as a `polyMesh` which is a general description of an OpenFOAM mesh with all its features and restrictions. For a given static mesh case the mesh data will be stored in `constant/polyMesh`. The typical mesh data files found here include: `points`, `faces`, `owner`, `neighbour` and `boundary`. Of course, the contained data must be valid, to define a mesh properly. The `pitzDaily` tutorial of the `potentialFoam` solver serves as an example in the following discussion, which can be found by issuing the following commands:

```
?> tut
?> cd basic/potentialFoam/pitzDaily
```

After inspeciting the content of the `polyMesh` directory, it is clear that it does not contain the required mesh data, yet. Solely the `blockMeshDict` is present in this tutorial, but executing `blockMesh` in the case directory generates the mesh and associated connectivity data:

```
?> ls constant/polyMesh
   blockMeshDict boundary
?> blockMesh
?> ls constant/polyMesh
   blockMeshDict boundary faces neighbour owner points
```

Users having worked with CFD codes before, especially with codes that are based upon structured meshes, may be missing the per-cell addressing. Rather than constructing the mesh on a per-cell basis, the unstructured FVM method in OpenFOAM constructs the mesh on a per-face basis. For more information on this, the reader is referred to the *owner-neighbour addressing* section of chapter 1.3. The following list illustrates the purposes of each file in `constant/polyMesh`.

points defines all points of the mesh in a `vectorField`, where their position in space is given in meters. These points are not the cell centres **C**, but rather the corners of the cells. To translate the mesh by 1 m in positive x-direction, each point must be changed accordingly. Altering any other structure in the `polyMesh` subdirectory for this purpose is not required, but this is covered by section 2.4.

A closer look at the `points` can be taken by opening the respective file with a text editor. To keep the output limited, the header is neglected and only the first few lines are shown:

```
?> head -25 constant/polyMesh/points | tail -7
25012   // Number of points
(
(-0.0206 0 -0.0005)        // Point 0
(-0.01901716308 0 -0.0005)  // Point 1
(-0.01749756573 0 -0.0005)
(-0.01603868134 0 -0.0005)
(-0.01463808421 0 -0.0005)
```

The `points` contain nothing more than a list of 25012 points. This list does not require to be ordered in any way although it can be. Additionally, all elements of the list are *unique*, meaning that point coordinates cannot occur multiple times. Accessing and addressing those points is performed via the list position in the `vectorField`, starting with 0. The position is stored as a `label`.

faces composes the faces from the points by their position in the `points` `vectorField` and stores them in a `labelListList`. This is a nested list, containing one element per face. Each of these elements is in turn a `labelList` of its own, storing the labels of the `points` used to construct the face. Figure 2.1 shows a visualization of the `labelListList` construct.

Each face must consist at least of three points and its size is followed by a list of point labels. On a face, every point is connected by a straight edge to its neighbours [4]. Using the points which define the face, the surface area vector $\mathbf{S}_f$ can be calculated where
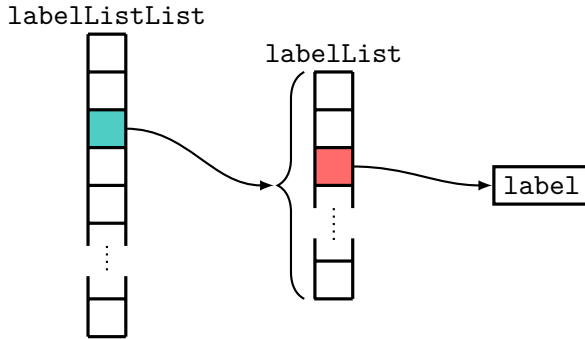
Figure 2.1: Graphical representation of a `labelListList`, used to represent the `faces`

the direction is determined by the right-hand-rule.

Again, only the first few lines of the `faces` are shown to keep things short:

```
?> head -25 constant/polyMesh/faces | tail -7
49180   // Number of faces as labelListList
(
4(1 20 172 153) // Face 0 with it's four point labels as labelList
4(19 171 172 20)
4(2 21 173 154)
4(20 172 173 21)
4(3 22 174 155)
...
)
```

As can be seen from the first line of the output, the mesh consists of 49180 faces of which only a subset is shown above. Similar to the face list's length of 49180, the length of each `labelList` is stated before the lists starts. Hence all faces shown here are built from 4 points, referred to by their position in the `points` list.

owner is again a `labelList` with the same dimension as the list storing the faces. Because the faces are already constructed and stored in the `faces` list, their affiliation to the volume cells must be defined. Per definition, a face can only be shared between two adjacent cells. The `owner` list stores which face is owned by which cell, which in turn is decided based on the cell label. The cell with the lower cell label owns the face and the other remaining face is considered the neighbor. It instructs the code that the first face (index 0 in the list) is owned by the cell with the label that is stored at that position.

A closer look at the owner file reveals that the first two faces are owned by cell 0 and the next two faces are owned by cell 1.

```
?> head -25 constant/polyMesh/owner | tail -7
49180
(
0
0
1
1
```

The ordering of this list is the result of the *owner-neighbour-addressing*, that was presented extensively in the previous chapter.

neighbour has to be regarded in conjunction with the owner list, as it does the opposite of the owner list. Rather than defining which cell owns each particular face, it stores its neighbouring cell. Comparing the owner with the neighbour file reveals a major difference: The neighbour list is significantly shorter. This is due to the fact that boundary faces don't have a neighbouring cell:

```
?> head -25 constant/polyMesh/neighbour | tail -7
24170
(
1
18
2
19
```

Again, the working principle of the *owner-neighbour-addressing* is explained in the previous chapter.

boundary contains all of the information concerning the mesh boundaries in a list of nested subdictionaries. Boundaries are often referred to as *patches* or *boundary patches*. Similar to the previous components of the mesh, only some relevant lines are shown:

```
?> head -25 constant/polyMesh/boundary | tail -8
5
(
    inlet // patch name
    {
        type        patch;
        nFaces      30;
        startFace   24170;
    }
```

For the pitzDaily example used throughout this section, the boundary file contains a list of 5 patches. Each patch is represented by a dictionary, lead by the patch name. The information included in the dictionary includes: the patch type, number of faces and
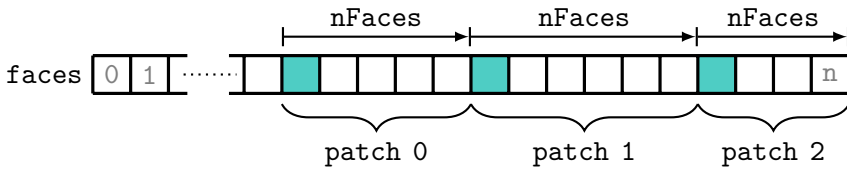
Figure 2.2: Working principle of the boundary addressing in OpenFOAM for a mesh with 3 patches and n faces. Grey elements denote the `startFace` of the particular patch.

starting face. Due to the sorting of the faces list, faces belonging to a certain patch can be addressed quickly and easily using this convention.

The addressing method for the boundary faces is illustrated in Figure 2.2. By design, all faces which don't have a `neighbour` are collected at the end of the `faces` list where they are sorted according to their owner patch. All faces that are boundary faces must be covered by the boundary description.

From a user's perspective, neither `points` nor `faces`, `owner` and `neighbour` will need to be touched or manipulated manually. If they are changed manually, this will most certainly destroy the mesh. The `boundary` file, however, may need to be altered for certain setups depending on your workflow. The most likely reason for changing the `boundary` file is to alter a patch name or type. It may be considerably easier to make this change here rather than re-running the respective mesh generator.

Now that the basic structure of an OpenFOAM mesh is explained, the boundary patch types will be reviewed. There are several patch types that can be assigned to a boundary, some of which are more common than others. It is important to distinguish between the boundary (or patch) and the boundary *conditions* (see figure 2.3).

A patch is an outer boundary of the computational domain and it is specified in the `boundary` file, hence being a *topological* property. The logical connection between the boundaries and the CAD geometry is that the surface of both should be as identical as possible. Expressed in terms of mesh topology, it is a collection of faces that solely have an `owner` cell and no `neighbour` cell. In contrast to the patch, boundary conditions

Boundary Conditions

e.g.: U ,p

Boundary

Computational Grid
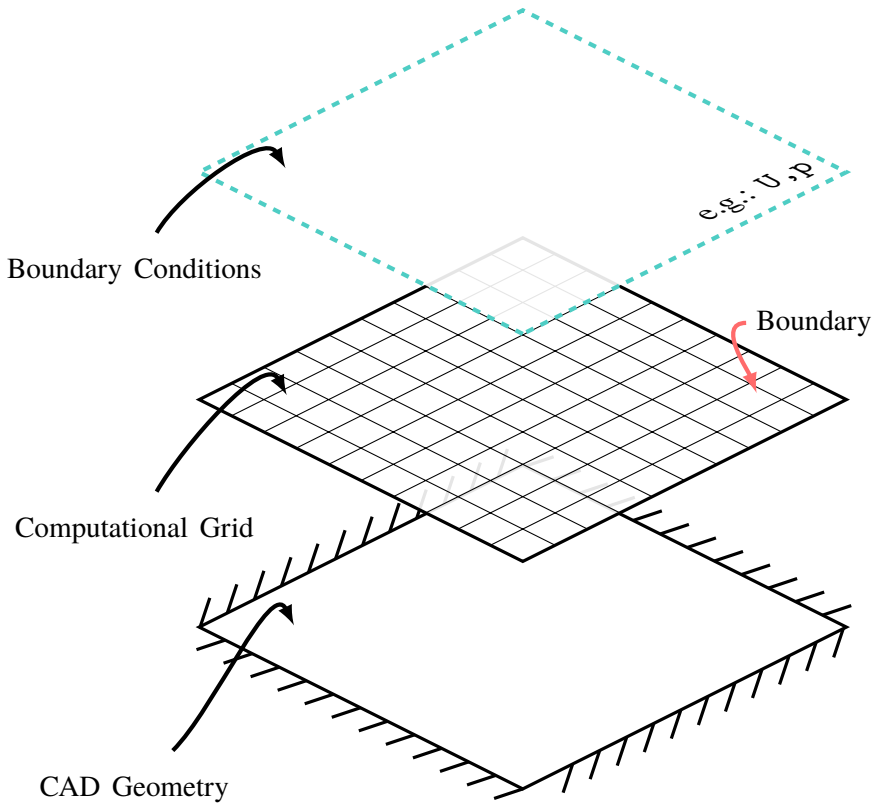
CAD Geometry

Figure 2.3: Illustration of the differences and relations between a CAD geometry, the computational grid and its boundaries, and the boundary conditions applied to those boundaries.

are applied to the patches for each of the fields ($\mathbf{U}$,p, etc...) separately. The patch types are:

patch Most patches (boundaries) can be described by the type patch, as it is the most general description. Any boundary condition of Neumann, Dirichlet or Cauchy can be applied to boundary patches of this type.

wall If a patch is defined as wall, it does not imply that there is no flow through that patch. It solely enables the turbulence models to properly apply wall functions to that patch (see chapter 7). Preventing a flow through the patch of type wall must still be explicitly defined via the velocity boundary condition.

symmetryPlane Setting the patch type to symmetryPlane declares it to act as a symmetry plane. No other boundary conditions can be applied to it except `symmetryPlane`, and must be applied for *all* fields. More information on the various available boundary conditions in OpenFOAM is given in chapter 10.

empty In case of a two-dimensional simulation, this type should be applied to the patches which are "in-plane". Similar to the `symmetryPlane` type, the boundary conditions of those patches have to be set to `empty` for all fields as well. No other boundary conditions will be applied to those patches. It is essential that all cell edges between both empty patches are parallel, otherwise an accurate two-dimensional simulation is not possible.

cyclic If a geometry consists of multiple components that are identical (e.g. a propeller blade or a turbine blade), only one needs to be discretized and treated as if it were located between identical components. For a four bladed propeller this would mean that only one blade is meshed (90° mesh) and by assigning a cyclic patch type to the patches with normals in tangential direction. These patches would then act as being physically coupled.

wedge This boundary type is similar to a cyclic patch, only specifically designed for cyclic patches which form a small (e.g. $\leq 5°$) wedge.

From an execution and compatibility standpoint, it does not matter how the `polyMesh` structure is created, as long as the mesh data in itself is valid. While there are various mesh generation tools packaged with OpenFOAM, external third-party meshers can be used so long as a valid conversion or output can be made.

In addition to the above mentioned *essential* core components of an

OpenFOAM mesh, there are various optional mesh constructs which may only be used for particular applications. Since they are optional, they can be present, regardless of the case setup. An OpenFOAM application reads them as needed and will report back to the user if they are missing.

Sets and Zones

As a user it is fairly easy to get confused by the fact that there are zones and sets in OpenFOAM and both to fairly similar things, when a user is concerned: they select mesh entities. The very short answer to the question which one to use is: use zones, as explained briefly by Hrvoje Jasak via Twitter (cp. Figure 2.4).
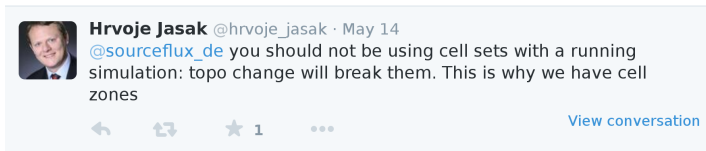
Figure 2.4: Hrvoje Jasak explaining when to use zones in OpenFOAM.

This is however only really relevant for solver applications. If the application is centered around pre- or post-processing, either one is fine. A sets are essentially `labelHashSets`, whereas zones inherit from `labelLists`. Both can store any mesh entity (point, face or cell) in a datastructure that is somewhat similar to a list. The major difference is in the internal handling of the mesh entities, especially in the case of a parallel simulation with topological mesh changes. In this case, the addressing in the list has to be updated accordingly and only the zone provides such a method.

The selection is usually performed by the tools `setSet` or `topoSet`, both of which can select subsets of the mesh and perform boolean operations on them. Both utilities can - generally speaking - convert zones to sets and the other way around. The set or zone may be created for any mesh entity (cell, point or face), but the a `cellSet` and a `cellZone` are the two ones, which are most commonly used. Zones are stored as ordinary dictionaries within the `constant/polyMesh`, whereas sets are stored in the `sets` subdirectory of `constant/polyMesh`. Zones and sets are stored identically on the filesystem: as a long list of labels of the respective mesh entity.

We have published some blog-posts that contain some degree of information on how zones and sets are assembled [2, 1].

2.1.1 CAD Geometry

Importing a geometry that has been generated in an external CAD software is a regular task for any CFD engineer. In OpenFOAM this is commonly performed with `snappyHexMesh`, however, the usage of this mesh generator will be explained later on. The only important concept for now is that this section only deals with the import of Stereolithography (STL) files. Other file types are supported and work in a similar fashion. STL is a file format that can store the surfaces of geometries in a *triangulated* manner. Both binary and ASCII encoded files are possible, but for sake of simplicity we will work with ASCII encoding.

As an example of a STL file, the following snippet shows an STL surface comprised of only one triangle:

```
solid TRIANGLE
   facet normal -8.55322e-19 -0.950743 0.30998
      outer loop
         vertex -0.439394 1.29391e-18 -0.0625
         vertex -0.442762 0.00226415 -0.0555556
         vertex -0.442762 1.29694e-18 -0.0625
      endloop
   endfacet
endsolid TRIANGLE
```

In this example, only one solid is defined which is named `TRIANGLE`. A STL file may contain multiple solids which are defined one after the other. Each of the triangles that compose the surface has a normal vector and three points.

The drawback of using ASCII STL files is that their filesize tends to grow rapidly with increasing resolution of the surface. Edges are not included explicitly because only triangles are stored in the file. Because of this, identifying and extracting feature edges from an STL is sometimes a challenging task.

An advantage of using STL as a file format is that one obtains a triangulated *surface mesh*, which by definition will always have planar surface components (triangles).

2.2 Mesh Generation

There are multiple open source mesh generators designed specifically for OpenFOAM, spread between the two main development branches (vanilla OpenFOAM and foam-extend). This includes `blockMesh`, `snappyHexMesh`, `foamyHexMesh`, `foamyQuadMesh`, and `cfMesh`. There are a few remaining tools, `extrudeMesh` and `extrude2DMesh`, but are not addressed in this section as they are not used by most OpenFOAM users. Furtheron they fall mostly under the mesh utilities, rather than under the core mesh generators, which are the ones that are discussed in this chapter. `blockMesh` and `snappyHexMesh` will be addressed briefly in this section with a review of their usage and their working principles. In a general sense, the purpose of the mesh generators is to generate the `polyMesh` datastructure described in the previous section in a user friendly fashion. Both mesh generators have similar input and output in that they read in a dictionary file and write the final mesh to `constant/polyMesh`.

2.2.1 blockMesh

When calling the executable `blockMesh` the `blockMeshDict` is read automatically from the `constant/polyMesh` directory, where it must be present.

`blockMesh` generates block-structured hexahedral meshes which are then converted into the arbitrary unstructured format of OpenFOAM. Generating grids with `blockMesh` for complex geometries is often a very tedious and difficult task and sometimes impossible. The effort that the user has to spend generating the `blockMeshDict` increases tremendously for complex geometries. Therefore, only simple meshes are typically generated using `blockMesh` and the discretization of the actual geometry is then shifted over to `snappyHexMesh`. This makes `blockMesh` a great tool for generating meshes which either consist of a fairly simple geometry, or to generate background meshes for `snappyHexMesh`.

An example block that `blockMesh` uses to construct the mesh is shown in figure 2.5. Each block consists of 8 corners called *vertices*. The hexahedral block is built from these corners. Edges, as indicated in figure 2.5,
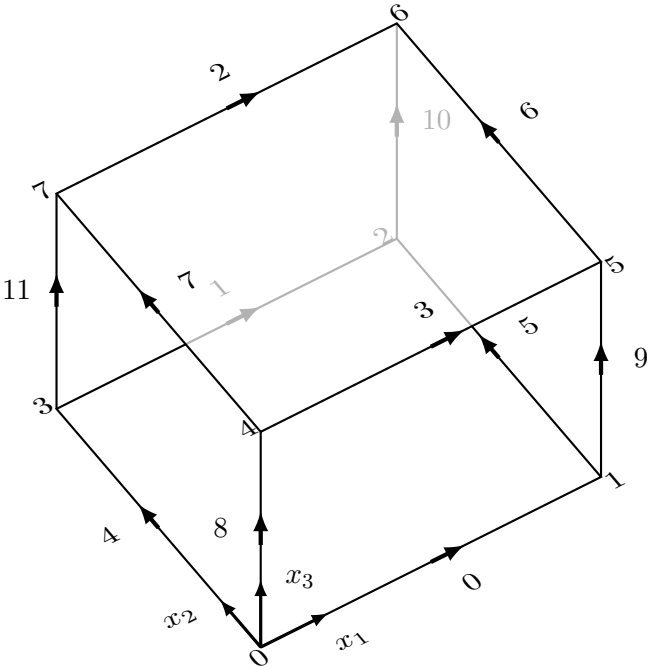
Figure 2.5: A `blockMesh` base block with vertice and edge naming convention.
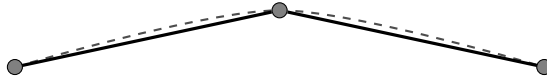
Figure 2.6: The gray dashed arc represents edge of a block, the black line denotes the resulting cell edges, and the gray dots indicate the three nodes on the edge.

connect vertices with each other. Finally, the surface of the block is defined by patches though those have only to be specified explicitly for block boundaries that don't have a neighbouring block. Boundaries between two blocks must not be listed in the patch definition, as they are - by definition - not patches. The length and number of nodes on the particular edges must match in order to stay topologically consistent. Boundary conditions for the actual simulation will be applied later on those patches.

Note that it is possible to generate blocks with less than 8 vertices and to have non-matching nodes on patches (see [4]), however, this is not covered by this guide. The edges of the block are straight lines by default, but can be replaced by different line types such as an arc, a polyline, or a spline. Choosing e.g. an arc does affect the shape of the block edge topology, but the connection between the final mesh points on that edge remain straight lines (see figure 2.6).

Coordinate Systems

The final mesh is constructed in the global (right-handed) coordinate system, which is Cartesian and aligned with the major coordinate axes: x, y, and z. This leads to a problem when blocks must be aligned and positioned arbitrarily in space. To circumvent this issue, each block is assigned its own right-handed coordinate system, which by definition does not require the three axis to be orthogonal. The three axes are labelled x_1, x_2, x_3 (see [4] and figure 2.5). Defining the local coordinate system is performed based on the notation shown in figure 2.5: Vertex 0 defines the origin, the vertex pair $(0, 1)$ represents x_1, while x_2 and x_3 are defined by vertex pairs $(0, 3)$ and $(0, 4)$, respectively.
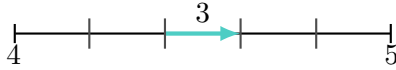
Figure 2.7: Illustration of the expansion ratio, with edge 3 used as an example

Node Distribution

During the meshing process, each block is subdivided into cells. The cells are defined by the nodes on the edges in each of the three coordinate axes of the block's coordinate system and follow the relationship given by:

$$n_{\text{cells}} = n_{\text{nodes}} - 1 \qquad (2.1)$$

It is up to the user to define in the `blockMeshDict` how many cells will appear on a certain edge. Cells on an edge can either be distributed uniformly or on a non-uniform spread based on a grading. There exist two types of gradings: **simpleGrading** and **edgeGrading** based. A **simpleGrading** describes the grading on an edge based on the size ratio of the last to the first cell on that particular edge (see figure 2.7):

$$e_r = \frac{\delta_e}{\delta_s} \qquad (2.2)$$

If $e_r = 1$ all nodes are spaced uniformly on that particular edge, no grading is present. With an expansion ratio $e_r > 1$, the node spacing increases from start to end of the edge. From the C++ sources of `blockMesh` it can be found that the expansion ratio that is defined by the user (e_r) is scaled by the following relation:

$$r = e_r^{\frac{1}{1-n}} \qquad (2.3)$$

Where n represents the number of nodes on that particular edge. By inserting equation (2.3) into equation (2.4), the relative position of the

i-th node on an edge can be calculated.

$$\lambda\left(r, i\right) = \frac{1 - r^i}{1 - r^n} \quad \text{with } \lambda \in [0, 1] \tag{2.4}$$

Even though this might look too laborious to perform for all of the blocks in a `blockMeshDict`, this comes in handy when a smooth transition in the cell sizes between two adjoining blocks is required. In many cases, simple try and error usually suffices.

Defining the dictionary for a minimal example

As a small example on how the `blockMeshDict` is set up correctly, a cube of $1\,\text{m}^3$ in volume is discretized. A prepared case can be found in `chapter2/blockMesh` directory of the example case repository. The dictionary itself consists of one keyword and four sub-dictionaries. The first keyword is `convertToMeters` which is usually 1. All point locations are scaled by this factor, which comes in handy if the geometry is very large or very small. In any of those cases we would end up typing a lot of leading or tailing zeroes, which is a tedious task. By setting `convert-ToMeters` accordingly, we can save some typing. The first relevant line of the `blockMeshDict` is:

```
convertToMeters    1;
```

Secondly the vertices must be defined. It is important to remember that vertices in `blockMesh` are different from the points of the created `poly-Mesh`, though their definition is fairly similar. For the unit cube example the vertices definition is

```
vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 1)
    (1 0 1)
    (1 1 1)
    (0 1 1)
);
```

From having a glance at the above definition, it is clear that the syntax is a list and similar to the list of points in the `polyMesh` definition. This is due to the round brackets that indicate a list in OpenFOAM, whereas curly brackets would define a dictionary. The first four lines define all four vertices in the $x_3 = 0$ plane and the following do the same for the $x_3 = 1$ plane. Similar to the points in `polyMesh`, each element is accessed by its position in the list and not by the coordinates. Please note, that each vertex must be unique and hence only occur once in the list.

As a next step, the blocks must be defined. Figure 2.5 can be used as a reference. An example block definition for the unit cube could look like

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 10) simpleGrading (1 1 1);
);
```

Again this is a list which contains blocks and is not a dictionary, due to the round brackets. The definition might appear odd at first glance, but is actually quite straight forward. The first word `hex` and the first set of round brackets containing eight numbers tells `blockMesh` to generate a hexahedron out of the vertices 0 to 7. These vertices are exactly those specified in the `vertices` section above and are accessed by their labels. Their order is not arbitrary, but defined by the *local block* coordinate system as follows:

1. For the local $x_3 = 0$ plane list all four vertex labels starting at the origin and moving according to the right-handed coordinate system.
2. Do the same for the local $x_3 \neq 0$ plane

It is possible to obtain a valid block definition by messing up the order of the vertex list in the particular block definition. The resulting block will either look twisted or uses incorrect global coordinate orientation. As soon as `blockMesh` and `checkMesh` are executed and the mesh is analyzed in a post-processor (e.g. `paraView`), this is detected.

> INFO
>
> `checkMesh` is a nativ OpenFOAM tool to check the mesh integrity and quality, based on various criteria. If the output of `checkMesh` states that the mesh is not ok, it *must* get improved.

The second set of round brackets defines how many cells are distributed in each particular direction of the block. In this case, the block possesses 10 cells for each direction. Changing the cell count to 2 cells in x_1, 20 cells in x_2 and 1337 cells in x_3, the block definition would look like this:

```
hex (0 1 2 3 4 5 6 7) (2 20 1337) simpleGrading (1 1 1);
```

The last remaining bit is the `simpleGrading` part in conjunction with the last set of numbers in the round brackets. This is the easiest way of defining a grading (or expansion ratio) as described before. The keyword `simpleGrading` in this case, defines the grading for all four edges in each of the three local coordinate system's axis directions to be identical. Hence each of the three numbers stated in the brackets after `simple-Grading` defines the grading for x_1, x_2 and x_3 direction, respectively. Sometimes this is not versatile enough, though. This is where `edgeGrading` can be used. This more advanced grading approach is essentially the same as `simpleGrading`, but the grading for each of the 12 edges on a hexahedron can be specified explicitly. Therefore the last set of brackets would not list 3 numbers, but 3 times 4. Now, each edge can be set individually.

Saving the `blockMeshDict` and executing `blockMesh` afterwards, results in a valid mesh that looks similar to what is defined in the `blockMesh-Dict`. But `blockMesh` warns about undefined patches that are all put into the *defaultFaces* patch by default.

Specifying patches manually is done by defining them inside the list called `patches` and for the example patch 0:

```
patches
(
    XMIN
    {
        type patch;
        faces
        (
            (4 7 3 0)
        );
    }
);
```

This instructs `blockMesh` to generate a patch of type `patch` named *XMIN*, based on the face that is constructed from the vertices 4, 7, 3 and 0. Internally, the patch name is defined as `word` and this data type
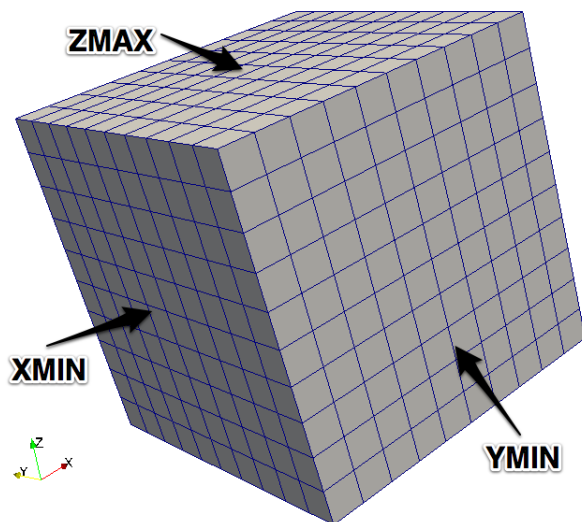
Figure 2.8: Unit cube meshed with `blockMesh` and an edge resolution of 10 cells in each direction

shows up in error messages regularly. How the vertices are ordered is not arbitrary, though. They need to be specified in a clockwise orientation, looking from *inside* the block. An image of the unit cube of our minimal example, consisting of 1000 small cubes is shown in figure 2.8, with highlighted `XMIN`, `YMIN` and `ZMAX` patches. The files used to generate this mesh can be found in the example repository under `chapter2/blockMesh`.

As stated earlier, the edges of a block are lines by default and thus the list containing the edge definitions is optional. Quite similarly, to the above defined blocks and patches, connecting two vertices by an arc instead of the default line would look like this:

```
edges
(
    arc 0 1 (0.5 -0.5 0)
);
```

Each item of the list containing the edge definitions starts with a keyword which indicates the type of edge, followed by the labels of the start and end vertex. In this example, the line is closed by the third point that is required to construct an arc. For any other edge shape (e.g. *polyLine* or *spline*), this point would be replaced by a list of supporting points.
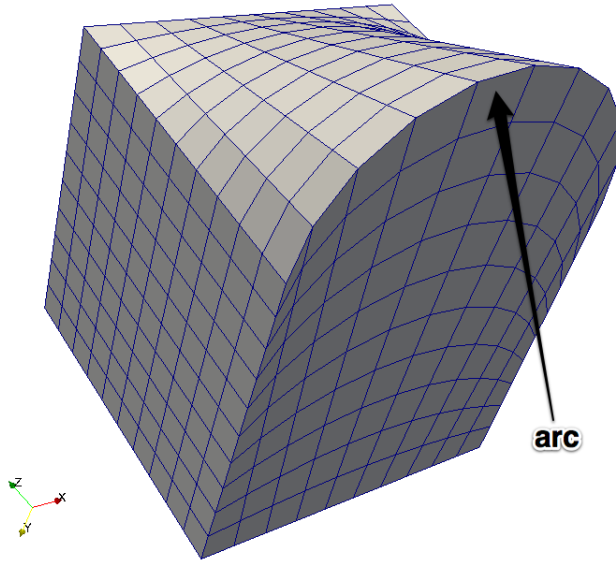
Figure 2.9: Unit cube meshed with blockMesh and an arc as one edge

An example of how inserting the above listed code alters the shape of the unit cube (see figure 2.8) is presented in figure 2.9.

To proceed with the `snappyHexMesh` section, you need to generate a unit cube consisting of 50 cells in each direction.

2.2.2 snappyHexMesh

Compared to `blockMesh`, `snappyHexMesh` may not require as much tedious work, such as adding and connecting blocks. On the other hand, one has less control over the final mesh. With `snappyHexMesh`, hexa-dominant meshes can be generated easily, needing only two things: a hexahedral background mesh and secondly one or more geometries in a compatible surface format. `SnappyHexMesh` supports local mesh refinements defined by various volumetric shapes (see table 2.1), application of boundary layer cells (prisms and polyhedras) and parallel execution.

`SnappyHexMesh` is a complex program and is controlled by a multitude of controlling parameters. Describing all of them in detail is beyond the scope of this book. Please read the [4] in conjunction with this book, for
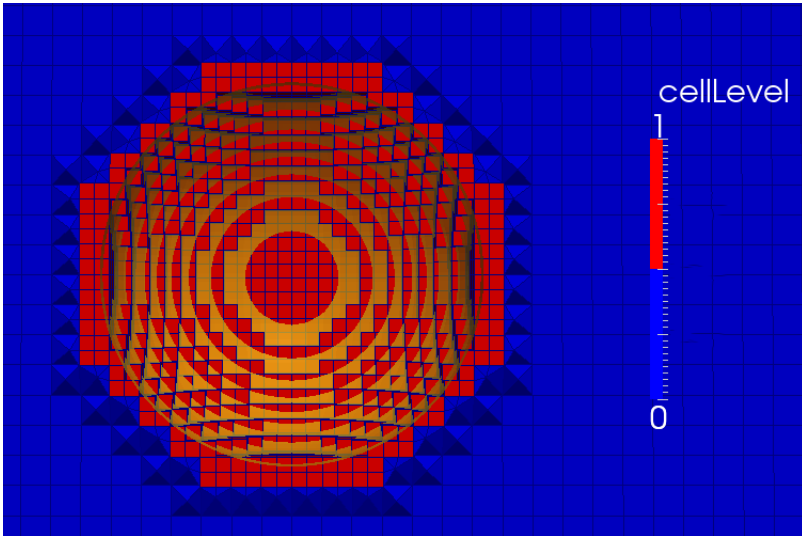
Figure 2.10: A STL sphere ($D = 0.25\,\text{m}$) meshed with `snappyHexMesh` after the first meshing step. The hexahedrons are not aligned with the body surface, yet.

a more thorough discussion of `snappyHexMesh`. Additional information can be found here: [5].

The execution flow of `snappyHexMesh` can be split into three major steps which are executed successively. Each of these steps can be disabled by setting the respective keywords to `false` at the beginning of the `snappyHexMeshDict`. These three steps can be summarized as follows:

castellatedMesh This is the first stage and performs two main operations. First, it adds the geometry to the grid and removes the cells which are not inside the flow domain. Second, the existing cells are split and refined according to the user's specifications. The result is a mesh which consists only of hexahedrons that more or less resembles the geometry. However, the majority of mesh points which are supposed to be placed on the geometry's surface are not aligned with it. A screenshot of a later example at this stage of the meshing process is shown in figure 2.10.

snap By performing the snapping step, the mesh points in the vicinity of the surface are moved onto the surface. This can be seen in figure 2.11. During this process, the topology of those cells may get
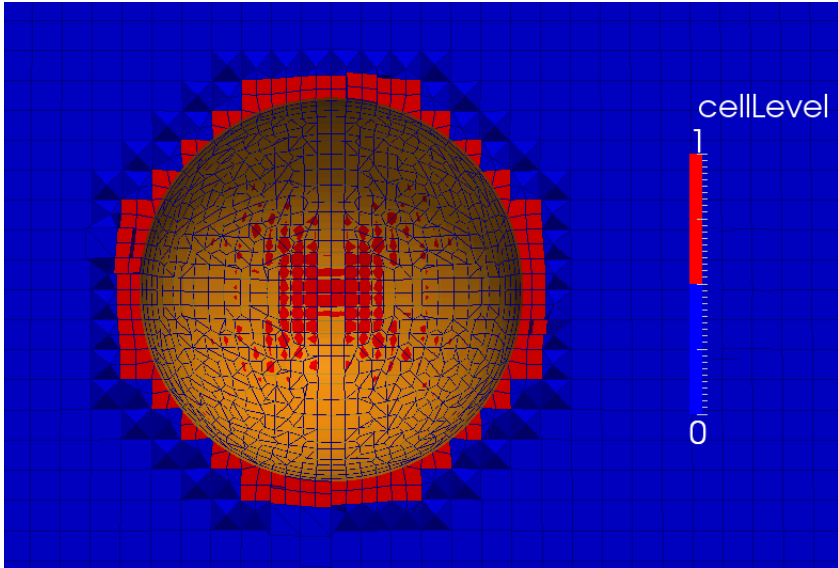
Figure 2.11: The same sphere as above but after the snapping process. All points are aligned with the body surface.

changed from hexahedrons to polyhedrons. Cells near the surface may be deleted or merged together.

addLayers Finally, additional cells are introduced on the geometry surface, that are usually used to refine the near wall flow (see figure 2.12). The pre-existing cells are moved away from the geometry in order to create space for the additional cells. Those cells are likely to be prisms.

All of the above settings and many more are defined in `system/snappyHexMeshDict`, the dictionary which contains all of the parameters required by `snappyHexMesh`. Several helpful tutorials can be found in the OpenFOAM tutorials directory under `meshing/snappyHexMesh`. Compared to other OpenFOAM dictionaries, the `snappyHexMeshDict` is very long and consists of many hierarchy levels which are represented by nested subdictionaries. One time step is written to the case directory, for each of the above mentioned steps (assuming you have a standard configuration). Each of the three steps will be addressed individually in the following section.
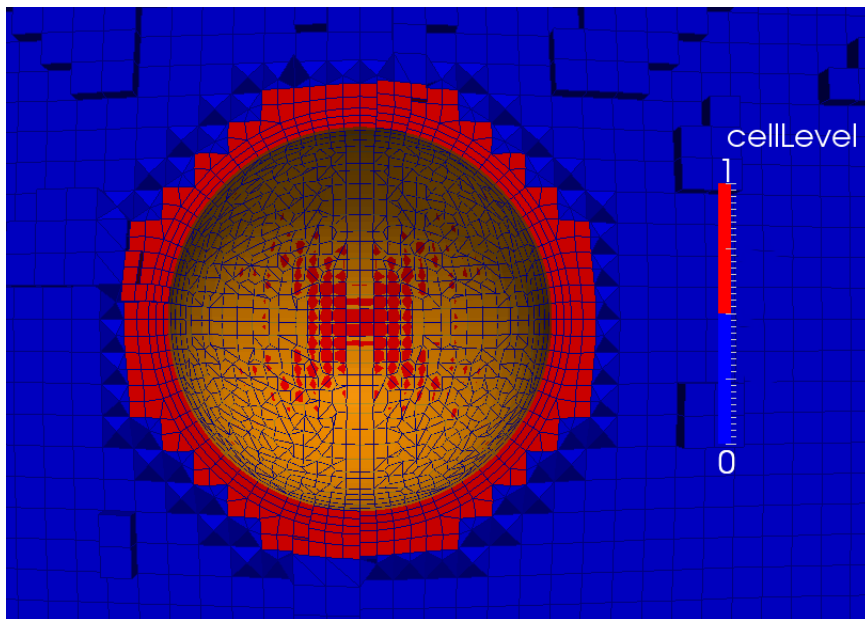
Figure 2.12: Prism layers are applied to the sphere surface, by extruding the surface.

Cell levels

Cell levels are used to describe the refinement status of a background mesh cell. When `snappyHexMesh` is started, the background mesh is read and all cells are assigned cell level 0 (blue cells in figure 2.12). If a cell is refined by one level, each of the edges are halved, generating eight cells from the previous one 'parent' cell. This method of refining is based on octrees and is only applicable for hexahedrons which is why a hexahedral background mesh is required by `snappyHexMesh`. With `snappyHexMesh` it is impossible to refine cells in only one direction, as this cannot be covered by octrees. Therefore they are refined - by definition - uniformly in all three spatial directions.

Defining the geometry

Before starting the meshing process, the geometry must be defined in the *geometry* subdictionary in the `snappyHexMeshDict`. Without the need to

define anything in the `snappyHexMeshDict`, the existing mesh in constant/polyMesh is read automatically and serves as background mesh. If there is no such mesh available, or if it is not purely based on hexahedrons, `snappyHexMesh` will not run. For external flow simulations, the outer boundaries defined by the background mesh are not as important as for internal flows. They can hence be kept as defined by the background mesh, without spending further work on them. For internal flow simulations on the other hand, the outer shape of the background mesh is of no interest as it is defined by the actual geometry.

> **INFO**
>
> STL geometries can be generated using almost any CAD program. Paraview may be used to generate an STL representation of basic shapes such as cylinders, spheres, or cones. Under the *sources* menu, various shapes are available which can be exported using the *save data* entry, under *file* menu.
>
> For real world geometries there are of course various methods to generate the surface mesh and store it as STL. However, bear in in mind that the quality of the surface mesh is essential to obtain a good volume mesh.

As a simple example, the unit cube mesh which was prepared in the previous section is reused and a sphere is inserted into it. The sphere is generated using a STL file, instead of the shapes listed in table 2.1. Loading a STL geometry can be done in a straight forward manner, by simply copying the geometry to `constant/triSurface` of the case and adding the following geometry subdictionary in the `snappyHexMeshDict`. An example of this looks like the following:

```
geometry
{
    sphere.stl // Name of the STL file
    {
        type    triSurfaceMesh; // Type that deals with STL import
        name    SPHERE; // Name access the geometry from now on
    }
}
```

The lines above tell `snappyHexMesh` to read `sphere.stl` from constant/triSurface as a triSurfaceMesh and refer to the geometry contained in that STL as SPHERE. Some simple geometry objects can be constructed without the need to open any CAD program, right inside

Shape	Name	Parameters
Box	searchableBox	min, max
Cylinder	searchableCylinder	point1, point2, radius
Disk	searchableDisk	origin, normal, radius
Plane	searchablePlane	point, normal
Plate	searchablePlate	origin, span
Sphere	searchableSphere	centre, radius
Collection	searchableSurfaceCollection	geometries
SurfaceWithGaps	searchableSurfaceWithGaps	gap, geometries

Table 2.1: List of cell selection shapes

snappyHexMesh. A list of these geometrical shapes is compiled in table 2.1.

Any of the shapes listed in table 2.1 can be constructed in the geometry subdictionary by simply appending to the existing subdictionary. As an example, a box is to be added to the geometry subdictionary, which is constructed from a minimum and maximum point. When using this approach not that it is impossible to rotate the box straight away and it will always be aligned with the coordinate axis.

```
smallerBox
{
    type    searchableBox;
    min     (0.2 0.2 0.2);
    max     (0.8 0.8 0.8);
}
```

Similar to the STL definition, the leading string of the subdictionary that defines the searchableBox is the name that is used to access that geometry later on. Sometimes it is desirable to compose a geometry out of the shapes listed in Table 2.1, but treat it as one single geometry rather than multiple. This is where the searchableSurfaceCollection can be used. By using this approach on geometry components that already exist, surfaces can be combined, rotated, translated and scaled. In any case, combining SPHERE and smallerBox into one and scaling the fancybox up by a factor of 2 would look as shown below:

```
geometry
{
    ...
    fancyBox
```

Adding refinements Removing cells inside geometry
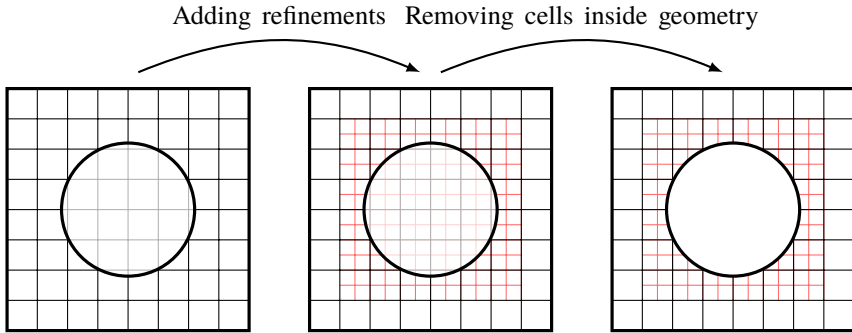
Figure 2.13: Schematic of the actions performed during the `castellat-edMesh` step.

```
{
    type searchableSurfaceCollection;
    mergeSubRegions true;
    SPHERE2
    {
        surface SPHERE
        scale  (1 1 1);
    }
    smallerBox2
    {
        surface smallerBox;
        scale (2 2 2);
    }
}
}
```

Setting up the castellatedMesh step

This is the first out of three steps during the execution of `snappy-HexMesh`. It includes the following two main steps: splitting the cells according to the user specifications and deleting cells that are outside the meshed region. A schematic of this process is given in figure 2.13.

The existing background mesh (black in figure 2.13) is read from `con-stant/polyMesh`. Based on the parameters in the `castellatedMesh-Controls` subdictionary of the `snappyHexMeshDict`, the mesh is refined. It is important to distinguish between refinements that are defined by *geometry surfaces* and *volumetric refinement*. The *surface refinement* ensures

that the boundary faces that represent the geometry are refined up the defined level. It is important to note that this does not only affect the cells owning the particular cells, but the adjecent cells as well. Therefore, surface refinement may appear somewhat similar to a volumetric refinement, however, it is distinctly different. Applying such a surface refinement to the `SPHERE` is controlled by entries in the `castellatedMeshControls`, which could look like the following:

```
castellatedMeshControls
{
    ...
    refinementSurfaces
    {
        SPHERE // Name of the surface
        {
            level (1 1); // Min and max refinement level
        }
    }
    ...
}
```

This refines the surface of the `SPHERE` to level 1. The two numbers between the round brackets define a minimum and maximum level of refinement for this surface. `snappyHexMesh` chooses between both, depending on the surface curvature: highly curved surface areas are refined to the higher level, lesser curved ones to the lower level.

Refinements in `snappyHexMesh` are not limited to surface definitions. Any geometry defined in the geometry subdictionary can serve as defining shape for a *volumetric refinement* as well. These volumetric refinements are called `refinementRegions` and are defined in a subdictionary with the same name, in the `castellatedMesh` controls. Compared to `refinementSurfaces`, `refinementRegions` offer a higher grade of versatility and thus require more options to be defined.

The `mode` has three options: `inside`, `outside` and `distance`. As the names suggests, inside only affects cells inside the selected geometry whereas outside does exactly the opposite. The third option, distance, is a combination of both and is calculated in outward *and* inward normal direction of the surface. In addition to `modes`, there is a `levels` option, which is more complex than for `refinementSurfaces`. It can already be guessed from the name, it does support an arbitrary number of levels. Each level must be defined in conjunction with a distance. With increasing position in the list, the levels must decrease and the distances must

increase. Refining anything inside the `smallerBox` to level 1 can be done by adding the following lines to the `castellatedMeshControls`:

```
castellatedMeshControls
{
   ...
   refinementRegions
   {
      smallerBox // Geometry name
      {
         mode    inside; // inside, outside, distance
         levels  ((1E15 1)); // distance and level
      }
   }
   ...
}
```

The above code uses a distance of 1×10^{15} m, in order to safely select all cells inside the geometry.

Without specifying a point located inside the volume of the final mesh, it is impossible for `snappyHexMesh` to decide which side of the sphere the user wants to discretize. That is why the `locationInMesh` keyword must be defined in the `castellatedMeshControls` subdictionary, as well. This point must not be placed on a face of the background mesh. For the unit cube example, this point is defined as

```
locationInMesh (0.987654 0.987654 0.987654);
```

The next step is to adjust the parameters of the `snap` subdictionary in the `snappyHexMeshDict`.

Setting up the snapping step

Compared to the other two steps of `snappyHexMesh`, this does not require extensive user input. This step is responsible for aligning the purely hex-ahedral mesh faces with the geometry by introducing new points into the mesh and displacing them (see figure 2.11). This is a highly iterative process, which is the reason why there is not much user interaction required. An example `snapControls` subdictionary of the `snappyHexMeshDict` read:

```
snapControls
{
   nSmoothPatch 3;
   tolerance 2.0;
```

```
    nSolveIter 30;
    nRelaxIter 5;

    // Feature snapping
    nFeatureSnapIter 10;
    implicitFeatureSnap false;
    explicitFeatureSnap true;
    multiRegionFeatureSnap false;
}
```

There are only iteration counters, tolerances and flags defined. Half of the parameters deal with snapping to the edges of the geometry, which is not part of this description. A description of this can be found on [3], however. Depending on the specifics of the case, increasing the iteration counters usually leads to a higher quality mesh, but also increases the meshing time significantly.

> **INFO**
>
> All of the parameters are explained in further detail in the `snappy-HexMeshDicts`, provided with OpenFOAM.

Setting up the addLayers step

All settings for the `addLayers` step are defined in the `addLayersControls` subdictionary of the `snappyHexMeshDict`. Any surface can be used to extrude prism layers from, regardless of its type. Firstly, the number of cell layers to be extruded per boundary needs to be specified via the `layers` subdictionary. An example entry looks like the following:

```
addLayersControls
{
    ...
    layers
    {
        "SPHERE_.*" // Patch name with regular expressions
        {
            nSurfaceLayers 3; // Number of cell layers
        }
    }
    ...
}
```

Each patch name is followed by a subdictionary that contains the `nSurfaceLayers` keyword. This keyword defines the number of cell layers

that get extruded and is thus followed by an integer, denoting the number of cell layers to be extruded. In the above example, regular expressions are employed to match any patch names which start with SPHERE_. In this case it is only the sphere itself however the use of wildcard characters in this manner can greatly reduce setup time. A cross-section of the final mesh is shown in figure 2.12.

Various parameters of snappyHexMesh, especially those related to the layer extrusion, require adjustment in order to obtain a mesh that meets your requirements. A few of those are explained briefly in the following.

relativeSizes can switch from absolute to relative dimensioning for the following values. By default it is `true`.

expansionRatio defines the expansion factor from one cell layer to the next one.

finalLayerThickness is the thickness of the last cell layer (furthest away from the wall), with respect to the next cell of the mesh or in absolute meters, depending on your choice for the `relativeSizes` parameter.

minThickness if a layer cannot be thicker than `minThickness`, it is not extruded.

In the example, the settings shown below were employed.

```
relativeSizes      true;
expansionRatio     1.0;
finalLayerThickness 0.5;
minThickness       0.25;
```

Finally snappyHexMesh must be executed in the case directory to begin the meshing process. Each step generates a new time step directory which contains the mesh at that particular stage. If you choose to make changes to your mesh by adjusting the parameters in the snappyHexMeshDict, remember to delete the old time steps before rerunning snappyHexMesh.

> INFO
>
> Another high quality mesh generator for OpenFOAM is *enGrid* and can be obtained freely from `http://engits.eu/en/engrid`.

> **INFO**
>
> The application `cfMesh` is a distributed memory parallel OpenFOAM meshing tool which, like `snappyHexMesh`, takes in STL surfaces as an input. This package is developed by Creative Fields Ltd. and can be downloaded along with documentation at `www.c-fields.com`.

2.2.3 cfMesh

The `cfMesh` library is a cross-platform library for automatic mesh generation that is built on top of OpenFOAM. It is compatible with all recent versions of OpenFOAM and foam-extend and is licensed under General Public License (GPL). The library was developed by Dr. Franjo Juretić and it is distributed by *Creative Fields*, Ltd and is available for download from `http://www.c-fields.com`.

This section covers a brief overview of the library, options that govern the mesh generation process, as well as some of the utilities distributed with `cfMesh`. It is in no way a complete documentation of the project. More information is available on the aforementioned project web page.

The `cfMesh` library supports various 3D and 2D workflows, built by using components from the main library, which are extensible and can be combined into various meshing workflows. The core library is based on the concept of mesh modifiers, which allows for efficient parallelisation using both Symmetric Multiprocessor (SMP) and Distributed Memory Parallel (DMP) using message passing interface (MPI). In addition, special care has been taken on memory usage, which is kept low by implementing data containers (lists, graphs, etc.) that do not require many dynamic memory allocation operations during the meshing process.

The meshing process in `cfMesh` is automatic, and it requires an input triangulation and dictionary with the various meshing parameters (settings). Once the surface mesh and the settings are given, the meshing process is started from the console, and it runs automatically without any user intervention. The library is optimised such that the meshing workflows require a small number of settings, and they have a simple syntax. Currently, `cfMesh` can create volume meshes inside a manifold, see figure 2.14, which does not need to be watertight.

(a) Manifold domain   (b) Manifold domain with a hole
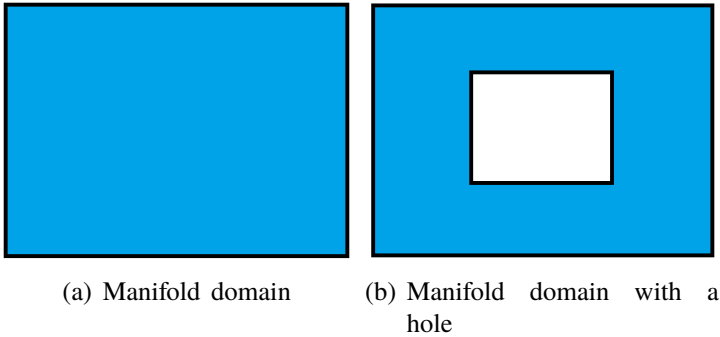
Figure 2.14: Allowed types of geometries in cfMesh

Available meshing workflows

All workflows are parallelised for shared memory machines and use all available CPU cores while running. The number of utilised cores can be controlled by the `OMP_NUM_THREADS` environment variable which can be set to the desired number of cores.

The available meshing workflows start the meshing process by creating a so-called mesh template from the input geometry and the user-specified settings. The template is later on adjusted to match the input geometry. The process of fitting the template to the input geometry is designed to be tolerant to poor quality input data which does not need to be watertight. The available workflows differ by the type of cells generated in the template.

Cartesian workflow generates 3D meshes consisting of predominantly hexahedral cells with polyhedra in the transition regions between the cells of different size. It is started by typing `cartesianMesh` in a shell window. By default, it generates one boundary layer which can be further refined on user request. In addition, this workflow can be run using MPI parallelisation, which is intended for generation of large meshes which do not fit into the memory of a single available computer.

The workflow generates 2D cartersian meshes. The mesh generator is started by typing `cartesian2DMesh` in the console. By default, it generates one boundary layer, which can be further refined. This meshing workflow requires the geometry in a form of a ribbon, as shown in
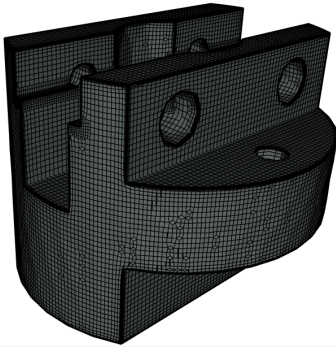
(a) Socket
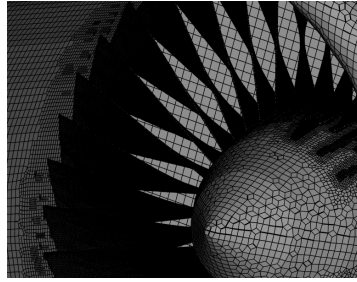
(b) Engine of an aeroplane
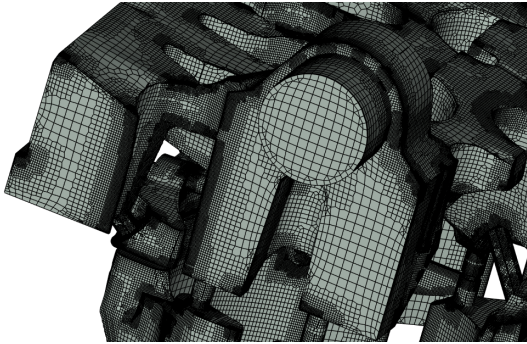
(c) Cooling jacket

Figure 2.15: 3D cartesian meshing

figure 2.16a, which span in the x-y plane and is extruded in the z direction.

Tetrahedral workflow generates meshes consisting of tetrahedral cells, figure 2.17, and is started by typing `tetMesh` in a console. By default, it does not generate any boundary layers, and they can be added and refined on user request.

Input geometry

Geoemtries used by `cfMesh` are required to be defined in the form of a surface triangulation. For 2D cases, the geometry is given in a form of a ribbon of triangles with boundary edges in the x-y plane (other

(a) Input surface mesh　　　(b) 2D volume mesh
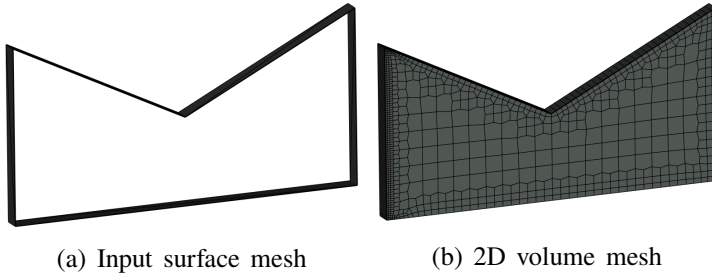
Figure 2.16: 2D cartesian meshing

(b) Cut through
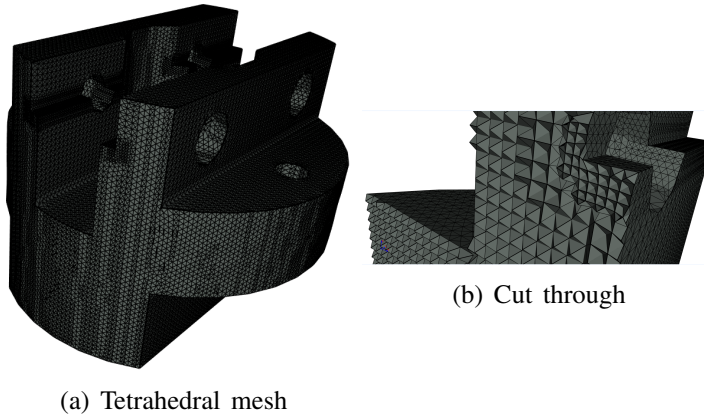
(a) Tetrahedral mesh

Figure 2.17: Tetrahedral meshing

orientations are not supported). The geometry consists of the following entities:

List of points contains all points in the surface triangulation.

List of triangles contains all triangles in the surface mesh.

Patches are entities which are transferred onto the volume mesh in the meshing process. Every triangle in the surface is assigned to a single patch, and cannot be assigned to more than one patch. Each patch is identified by its name and type. By default, all patch names and types are transferred to the volume mesh, and are readily available for definition of boundary conditions for the simulation.

Facet subsets are entities which are not transferred onto the volume mesh in the meshing process. They are used for definition of meshing settings. Each face subset contains indices of triangles in the surface mesh. Please note that a triangle in the surface mesh can be

contained in more than one subset. Facet subsets can be generated by cfSuite, a commercial application developed by Creative Fields, Ltd.

Feature edges re treated as constraints in the meshing process. Surface points where three or more feature edges meet are treated as corners. Feature edges can be generated by the `surfaceFeatureEdges` utility or `cfSuite`.

Figure 2.18 shows a surface mesh with highlighted patches.
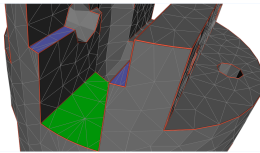
Figure 2.18: An example of geometry with patches and subsets.

All sharp features transferred by `cfMesh` must be defined by the user prior to the meshing process. The edges at the border between the two patches, figure 2.19a, and the feature edges are handled as sharp features in the meshing process, see figure 2.19b. Other edges in the triangulation are not constrained.
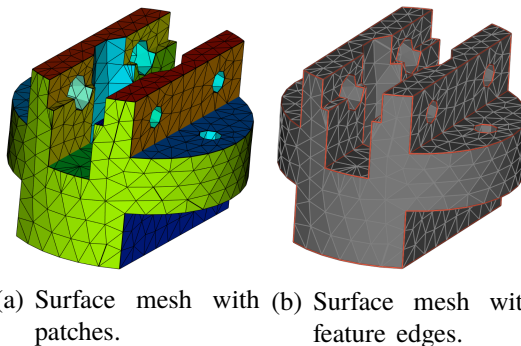
(a) Surface mesh with patches.

(b) Surface mesh with feature edges.

Figure 2.19: Possible ways of capturing feature edges.

The file formats suggested for meshing are: `fms`, `ftr`, and `stl`. In addition, the geometry can be imported in all formats supported by the `surfaceConvert` utility which comes with OpenFOAM. However, the three suggested formats support definition of patches which are transferred onto the volume mesh by default. Other formats can also be used

for meshing but they do not support definition of patches in the input geometry and all faces at the boundary of the resulting volume mesh end up in a single patch.

The preferred format for `cfMesh` is `fms`, designed to hold all relevant information for setting up a meshing job. It stores patches, subsets, and feature edges in a single file. In addition, it is the only format which can store all geometric entities into a single file, and the users are strongly encouraged to use it.

Dictionaries and available settings

The meshing process is steered by the settings provided in a `meshDict` dictionary located in the `system` directory of the case. For parallel meshing using MPI, a `decomposeParDict` located in the `system` directory of a case is required, and the number of nodes used for the parallel run must match the `numberOfSubdomains` entry in `decomposeParDict`. Other entries in `decomposeParDict` are not required. The resulting volume mesh is written in the `constant/polyMesh` directory. The settings available in `meshDict` will be explained in more detail in the remainder of this section.

The `cfMesh` library requires only two mandatory settings to start a meshing process:

surfaceFile points to a geometry file. The path to the geometry file is relative to the path of the case directory.

maxCellSize represent the default cell size used for the meshing job. It is the maximum cell size generated in the domain.

Refinement settings

When a uniform cell size is not satisfactory, there are many options for local refinement sources in `cfMesh`.

boundaryCellSize option is used for refinement of cells at the boundary. It is a global option and the requested cell size is applied everywhere at the boundary.

minCellSize is a global option which activates automatic refinement of the mesh template. This option performs refinement in regions where the cells are larger than the estimated feature size. The scalar value provided with this setting specifies the smallest cell size which can be generated by this procedure. This option is useful for quick simulation because it can generate meshes in complex geometry with low user effort. However, if high mesh quality is required, it provides hints where some mesh refinement is needed.

localRefinement allows for local refinement regions at the boundary. It is a dictionary of dictionaries, and each dictionary inside the main `localRefinement` dictionary is named by a patch or facet subset in the geometry which is used for refinement, The requested cell size for an entity is controlled by the `cellSize` keyword and a scalar value, or by specifying `additionalRefinementLevels` keyword and the desired number of refinements relative to the `maxCellSize`.
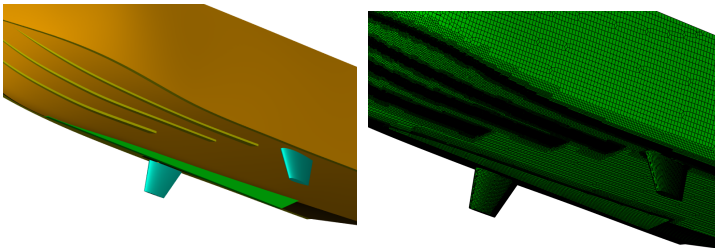
(a) Refinement regions at chines (b) Mesh with local refinement
 and stabilisers

Figure 2.20: Local refinement via patches/subsets

objectRefinement is used for specifying refinement zones inside the volume. The supported objects which can be used for refinement are: lines, spheres, boxes, and truncated cones. It is specified as a dictionary of dictionaries, where each dictionary inside the `objectRefinement` dictionary represents the name of the object used for refinement.

Meshing workflows implemented in `cfMesh` are based on inside-out meshing, and the meshing process starts by generating the so-called mesh template based on the user-specified cell size. However, if the cell size is locally larger than the geometry feature size it may result with gaps in
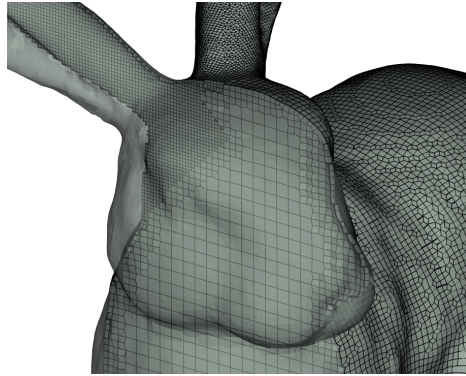
(a) Refinement inside a volume of a Stanford bunny.

Figure 2.21: Local refinement via primitive objects.

the geometry being filled by the mesh. On the contrary, the mesh in thin parts of the geometry can be lost if the specified cell size is larger than the local feature size.

The `keepCellsIntersectingBoundary` option is a global option which ensures that all cells in the template which are intersected by the boundary remain part of the template. By default, all meshing workflows keep only cells in the template which are completely inside the geometry. The `keepCellsIntersectingBoundary` keyword must be followed by either 1 (active) or 0 (inactive). Activation of this option can cause locally connected mesh over a gap, and the problem can be remedied by the `checkForGluedMesh` option which also must be followed by either 1 (active) or 0 (inactive).

The `keepCellsIntersectingPatches` option is an option which preserves cells in the template in the regions specified by the user. It is a dictionary of dictionaries, and each dictionary inside the main dictionary is named by patch or facet subset. This option is not active when the `keepCellsIntersectingBoundary` option is switched on.

The `removeCellsIntersectingPatches` option is an option which removes cells from the template in the regions specified by the user. It is a dictionary of dictionaries, and each dictionary inside the main dictionary is named by a patch or a facet subset. The option is active when the `keepCellsIntersectingBoundary` option is switched on.

(a) Selected regions at the sur- (b) Resolved gap in the volume
    face mesh.   mesh.
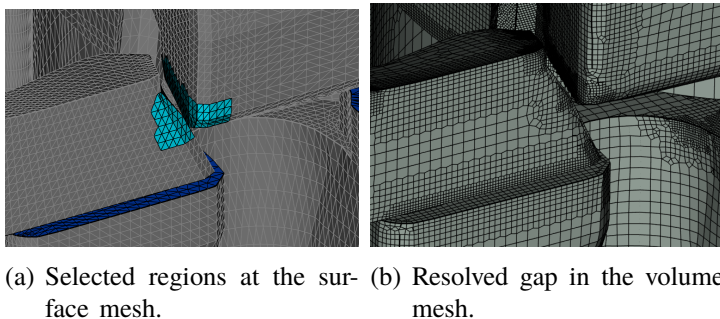
Figure 2.22: Remove cells intersected by the patches/subsets.

Boundary layers in `cfMesh` are extruded from the boundary faces of the volume mesh towards the interior, and cannot be extruded prior to the meshing process. In addition, their thickness is controlled by the cell size specified at the boundary and the mesh generator tends to produce layers of similar thickness to the cell size. Layers in `cfMesh` can span over multiple patches if they share concave edges or corners with valence greater than three. Furthermore, `cfMesh` never breask the topology of a boundary layer, and its final geometry depends on the smoothing procedure. All boundary layer settings are provided inside a `boundaryLayers` dictionary. The options are:

nLayers specifies the number of layers which will be generated in the mesh. It is not mandatory. In case it is not specified the meshing workflow generates the default number of layers, which is either one or zero.

thicknessRatio is a ratio between the thickness of the two successive layers. It is not mandatory. The ratio must be greater or equal to 1.

maxFirstLayerThickness ensures that the thickness of the first boundary layer never exceeds the specified value. It is not mandatory.

patchBoundaryLayers setting is a dictionary which is used for specifying local properties of boundary layers for individual patches.

It is possible to specify `nLayers`, `thicknessRatio` and `maxFirstLay-erThickness` options for each patch individually within a dictionary with the name equal to the patch name. By default, the number of layers generated at a patch is governed by the global number of layers, or the maximum number of layers specified at any of the patches which form a continuous layer together with the existing patch. `allowDiscontinuity`

option ensures that the number of layers required for a patch shall not spread to other patches in the same layer.

(a) Layers at a keel of a speed-boat.

(b) Example of `allowDiscontinuity` option.
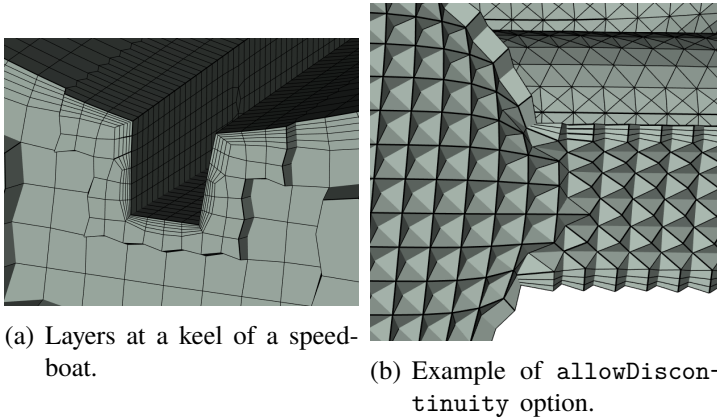
Figure 2.23: Boundary layers.

The settings presented in this section are used for changing of patch names and patch types during the meshing process. The settings are provided inside a renameBoundary dictionary with the following options:

newPatchNames is a dictionary inside the renameBoundary dictionary. It contains dictionaries with names of patches which shall be renamed. For each patch it is possible to specify the new name or the new patch type with the settings:

    newName keyword is followed by the new name for the given patch. The setting is not mandatory.

    type keyword is followed by the new type for the given patch. The setting is not mandatory.

defaultName is a new name for all patches expect the ones specified in newPatchNames dictionary. The setting is not mandatory.

defaultType sets the new type for all patches except the ones specified in newPatchNames directory. The setting is not mandatory.

Various utilities in `cfMesh`

Currently, following utilities are provided by the `cfMesh` project:

FLMAToSurface converts geometry from AVL's `flma` format into the format readable by `cfMesh`. Cell selections defined in the input file are transferred as facet subsets.

FPMAToMesh is a utility for importing volume meshes from AVL's fpma format. Selections defined on the input mesh are transferred as subsets.

copySurfaceParts copies surface facets in a specified facet subset into a new surface mesh.

extrudeEdgesInto2DSurface extrudes edges written as feature edges in the geometry into a ribbon of triangles required for 2D mesh generation. Generated triangles are stored in a single patch.

meshToFPMA converts the mesh into the AVL's fpma format.

patchesToSubsets converts patches in the geometry into facet subsets.

preparePar creates processor directories required for MPI parallelisation. The number of processor directories is dependent on the `number-OfSubdmains` specified in `decomposeParDict`.

removeSurfaceFacets is a utility for removing facets from the surface mesh. The facets which shall be removed are given by a patch name or a facet subset.

subsetToPatch creates a patch in the surface mesh consisting of facets in the given facet subset.

surfaceFeatureEdges is used for generating feature edges in the geometry. In case the output is a fms file, generated edges are stored as feature edges. Otherwise it generates patches bounded by the selected feature edges.

surfaceGenerateBoundingBox generates a box around the geometry. It does not resolve self-intersections in case the box intersects with the rest of the geometry.

2.3 Mesh Conversion from other Sources

While `blockMesh` and `snappyHexMesh` are powerful mesh generation tools, users may often use third party meshing packages for defining and discretizing a more complex flow domain.

2.3.1 Conversion from Thirdparty Meshing Packages

Many advanced external meshing utilities offer the user additional levels of control during mesh generation. This includes selectable element types, fitted boundary layer meshes, and length scale control to name a few. Some mesh generators can export directly to a functional OpenFOAM mesh format. Listed below is a compilation of the mesh formats supported for conversion in OpenFOAM-3.0:

- Ansys
- CFX
- Fluent
- GMSH

- Gambit
- Ideas
- Kiva
- Netgen

- Plot3D
- Star-CD
- tetgen
- KIVA

The capabilities of the importing utilities vary strongly, as well as the nomenclature used by them. The Fluent import tool converts *internal boundaries* to `faceSets`, whereas other tools ignore such features completely.

> **INFO**
>
> Due to licencing issues, the mesh conversion tool used to import meshes from StarCMM+ and libraries related to it need to be downloaded and compiled manually, rather than via a `Allrun` script.

If your particular meshing software is not mentioned in the above list, it is more than likely that it is capable of exporting a mesh into a supported intermediate format.

> **INFO**
>
> The source code for all of the above mentioned conversion utilities are found here: `$WM_PROJECT_DIR/applications/utilities/mesh/-conversion/`.

Users also have the option of converting OpenFOAM meshes into Fluent or Star-CD mesh formats using the `foamMeshToFluent` and `foam-ToStarMesh` utilities. This could be especially useful for exporting meshes generated from the `snappyHexMesh` utility mentioned previously.

The mesh conversion process is typically very straightforward with very little syntax changes between the different conversion utilities. For that

reason, only one example will be given and will be based on the `flu-entMeshToFoam` conversion utility. To begin the process, copy a tutorial case to a directory of choice, this tutorial is based upon the existing mesh conversion tutorial for the `icoFoam` solver.

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/elbow/ meshConversionTest
?> cd meshConversionTest
```

Converting the mesh is as simple as running the conversion utility and passing the mesh file as the argument, which must be present in the directory. During conversion the utlity will output patch names and mesh statistics to the console. The files contained in the `polyMesh` directory will be updated accordingly.

```
?> fluentMeshToFoam elbow.msh
```

It is important to keep in mind that the imported mesh is only as good as the exported. In case of the Fluent mesh, 2D meshes are not possible to import, since OpenFOAM only supports three-dimensional meshes. After the import is complete, the case will need to be updated to reflect the new patch names in the initial and boundary condition files. All existing patches can either be gathered from the output of the import tool, or looked up manually by opening `constant/polyMesh/boundary` with an editor. For this tutorial the `U` and `p` fields were pre-configured for this particular mesh.

Scaling the mesh during import is as simple as adding the option and scaling factor to the command. For the sake of this tutorial, the mesh should be scaled down by one order of magnitude.

```
?> fluentMeshToFoam -scale 0.1 elbow.msh
```

When constructing a mesh in many third-party meshing utilities, users can often assign boundary condition types such as inlet, outlet, wall, etc. to each particular patch. The conversion process will attempt to match certain boundary condition formats to a corresponding OpenFOAM format but there is no guarantee of success or accuracy in boundary condition conversion. It is crucial to check that the conversion correctly parsed the flow information. To check this, inspect `constant/polyMesh/boundary` and run `checkMesh` on the newly converted mesh.
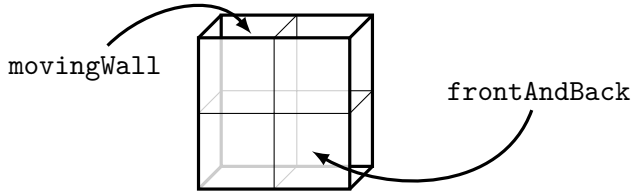
Figure 2.24: Simplified sketch of the `cavity` example

2.3.2 Converting from 2D to Axisymmetric Meshes

In order to convert a mesh to an axisymmetric one, the following require-
ments have to be fulfilled. The mesh must already be a *valid* OpenFOAM
mesh and it must only be one cell "thick". The latter requirement is valid
for all 2D meshes in OpenFOAM. As the `cavity` example of `icoFoam`
satisfies all these requriements, it is used for this tutorial. It is located
at `$FOAM_TUTORIALS/incompressible/icoFoam/cavity`.

With the mesh being shaped as a rectangle and having only one cell,
a very basic geometry can be created. In OpenFOAM an axisymmetric
mesh has the following properties: the mesh is one cell "thick" and is
rotated about an symmetry axis to form a 5° wedge shape. The two
angled boundaries of the wedge are considered separate patches of type
`wedge`.

> **INFO**
>
> The sources of `makeAxialMesh` are available on the OpenFOAM wiki:
> `http://openfoamwiki.net/index.php/Contrib_MakeAxialMesh`.
> Follow the instructions there to download and compile the utility.

The next steps are to create a copy of the case folder to a working
directory of your choice, renaming the directory to avoid any future
confusion and creating the 2D base mesh.

```
?> cp -r $FOAM_TUTORIALS/utilities/incompressible/icoFoam/cavity .
?> mv cavity axiSymCavity
?> cd axiSymCavity
?> blockMesh
```

For the axisymmetric mesh, the `movingWall` patch is used as symmetry
axis (see figure 2.24). In addition, the single `frontAndBack` patch will

be split and act as the two boundaries of the wedge (`frontAndBack_neg` `frontAndBack_pos`). The parameters entered into the command line reflect this:

```
?> makeAxialMesh -axis movingWall -wedge frontAndBack
```

The utility creates a new time directory (in this case `0.005`) to store the transformed mesh. If the creation did not work as expected, only this directory needs to be deleted and the basic mesh is restored again. The case directory should now contain the folders shown below:

```
?> ls
0 0.005 constant system
```

At this point the mesh has been warped into a 5° wedge shape, as shown in figure 2.25. However, the faces from the `movingWall` patch are present, however, they are now crushed into faces of near zero face area. `makeAxialMesh` transforms the point positions but does not alter the mesh connectivity. Because of this, the symmetry patch has no faces assigned to it (`nFaces = 0`) and must be removed. Using the `collapseEdges` tool is advisable in this case. It takes two mandatory command line arguments: edge length and merge angle:

```
?> collapseEdges <edge length [m]> <merge angle [degrees]>
```

For many applications an edge length of 1×10^{-8} m and merge angle of 179° will correctly identify and remove the recently collapsed faces. In some instances where the mesh edge length scale is extremely small, a smaller edge length may be required to avoid false positives and the inadvertant removal of valid edges. Executing `collapseEdges` with the parameters as shown works without issues for this example.

```
?> collapseEdges -latestTime 1e-8 179
```

For some final housekeeping it is advisable to remove the now empty patches from the `boundary` list. Open `constant/polyMesh/boundary` and delete the `movingWall` and `frontAndBack` entries. Note that they are listed as containing zero faces: `nFaces 0;`. Change the boundary list size to 3 to reflect these two deletions. The `boundary` file should now look similar to:
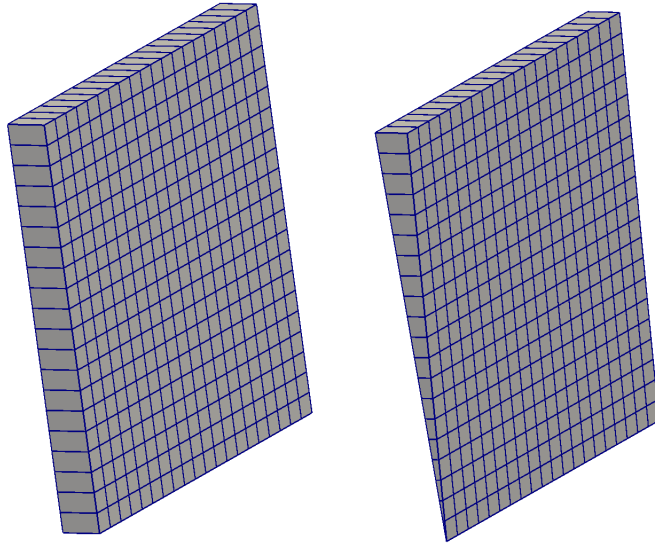
Figure 2.25: The 2D cavity mesh before and after the `makeAxialMesh` wedge transformation

```
3
(
    fixedWalls
    {
        type            wall;
        nFaces          60;
        startFace       760;
    }
    frontAndBack_pos
    {
        type            wedge;
        nFaces          400;
        startFace       820;
    }
    frontAndBack_neg
    {
        type            wedge;
        nFaces          400;
        startFace       1220;
    }
)
```

At this point, the `fixedWalls` patch can be split into 3 separate patches using the `autoPatch` utility. This will look at a contiguious patch and try to identify appropriate places to split it based on a given feature angle.

In this case, any patch edges that form an angle greater than 30° can be split to form a new patch. This provides more flexibility, when it comes to the assignment of boundary conditions.

```
?> autoPatch -latestTime 30
```

The patches will be renamed after the split. The `-latestTime` flag will only read the latest time step available. Instead of overwriting the time step, the split mesh is stored in yet another time step directory. Finally the mesh should be checked for errors, using the `checkMesh` tool, which should be considered a general rule of best practices: always run `checkMesh` when the mesh was changed.

2.4 Mesh Utilities in OpenFOAM

The utility applications (or just short *utilities*) which deal with mesh operations can be found in the directory `$WM_PROJECT_DIR/application-s/utilities/mesh`. The mesh utilities are grouped in the following categories: generation, manipulation, advanced and conversion. This categorization has not changed over the most recent versions at all. Generating the mesh and converting it from different formats into the OpenFOAM format has been described in section 2.2 and section 2.3. This section covers manipulating the mesh as well as advanced operations like mesh refinement once a base mesh has already been generated.

2.4.1 Refining the Mesh by a Specified Criterion

In this example, the mesh refinement application `refineHexMesh` is employed to refine the mesh of the `damBreak` tutorial of the `interFoam` solver. The purpose of this is to refine the area around the initial free surface, where the gradient of the two-phase marker field (α_{water}) is lower than zero ($\nabla(\alpha_{\text{water}}) < 0$).

To start, a local copy of the `damBreak` tutorial in the working directory of your choice must be generated. All utilities executed by the `Allrun` script must be executed, only the solver is not started. This generates the mesh and initializes the α_{water} field.

```
?> cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak .
?> cd damBreak
?> blockMesh
?> setFields
```

The mesh is now generated with the `blockMesh` and the α_{water} field is set using the `setFields` pre-processing utility. The `setFields` utility is described section 3.2. The basic calculator utility `foamCalc` can be used to compute and store the gradient of the α_{water} field.

```
?> foamCalc magGrad alpha.water
```

This will store the cell-centred scalar field of the gradient magnitude in the initial time directory 0 named `magGradalphaWater`. To refine the mesh based on the gradient magnitude using the `refineMesh` application, the configuration dictionary file for this utility must be copied into the `system` directory of the `damBreak` case.

```
?> cp $FOAM_APP/utilities/mesh/manipulation/refineMesh/refineMeshDict system/
?> ls system/
controlDict      fvSchemes   refineMeshDict
decomposeParDict fvSolution  setFieldsDict
```

In the configuration now available in the `system` directory, `refine-HexMesh` refines all cells in a certain cellSet.

```
// Cells to refine; name of cell set
set c0;
```

This `cellSet`, when created, will be stored in the `constant/polyMesh` and `refineHexMesh` will use it to refine the cells.

In this case, `topoSet` is used to generate the cellSet. This requires the `topoSetDict` to be present in the `system` directory and configured properly. Therefore an existing one is copied and changed afterwards.

```
?> cp $FOAM_APP/utilities/mesh/manipulation/topoSet/\
   topoSetDict system/
```

The example `actions` subdictionary of `topoSetDict` must be replace by the contents of:

```
actions
(
   {
       name   c0;
       type   cellSet;
       action new;
```

```
            source fieldToCell;
            sourceInfo
            {
                fieldName magGradalpha1;
                min 20;
                max 100;
            }
        }
    );
```

Now the `cellSet` can be generated, based on the definitions in sys-tem/topoSetDict:

```
    ?> topoSet
    ?> refineHexMesh c0
```

When the mesh is now viewed using `Paraview`, the area of the free surface should now have additional resolution.

2.4.2 transformPoints

In the OpenFOAM mesh format, the only information pertaining to scale and location of the mesh is in the point position vectors. All of the re-maining stored mesh information is purely connectivity based as discussed previously. With that said, the mesh size, position and orientation can be altered by transforming the point locations alone. For this purpose, the `transformPoints` mesh utility comes with OpenFOAM. Because this utility is relatively straight forward, only the required syntax is shown. The most often used options when transforming a mesh are the `-rotate`, `-translate` and `-scale` options.

> INFO
>
> It is also possible to use double quotation marks instead of single quotation marks around the brackets. The order in which the tasks are performed is hard coded and cannot be changed by the user. If you want to make sure that the scaling is performed before the translation, run `transformPoints` twice.

scale scales the points of the mesh in any or all cardinal directions by a specified scalar ammount. `-scale '(1.0 1.0 1.0)'` does not change the point locations, while `-scale '(2.0 2.0 2.0)'`

doubles the point positions in all directions uniformly. Any non-unform scaling will stretch or compress your mesh in your given direction(s).

translate moves the mesh by the given vector, effectivly adding this vector to every point position vector in the mesh.

rotate rotate the mesh. The rotation is defined by input vectors. The mesh will undergo the rotation required to orient the first vector with the second. When rotating a mesh, any initial or boundary vector and tensor values can be rotated as well by adding the `-rotateFields` option.

Syntax for these three point transformations are shown below.

```
?> transformPoints -scale '(x y z)'
?> transformPoints -translate '(x y z)'
?> transformPoints -rotateFields -rotate '( (x0 y0 z0) (x1 y1 z1) )'
```

2.4.3 mirrorMesh

Sometimes it is easier to generate a mesh with symmetry planes then performing mirror reflections than meshing the entire geometry in one step. The `mirrorMesh` utility is doing exactly that. All parameters, with regards to the mirroring process itself, are read from a dictionary, which is located in `system/mirrorMeshDict`.

In order to mirror a mesh sucessfully, the mirroring plane must be planar. For this example a quater mesh is mirrored into a full domain. First, the following solid analysis case must be copied into the directory of choice and renamed to prevent later confusion. The `mirrorMeshDict` needs to be copied from an existing case into the case `system` directory.

```
?> cp -r $FOAM_TUTORIALS/stressAnalysis/solidDisplacementFoam/plateHole .
?> mv plateHole mirrorMeshExample
?> cd mirrorMeshExample
?> cp -r $FOAM_APP/utilities/mesh/manipulation/mirrorMesh/mirrorMeshDict \
system/
```

The next step is to define the plane which will act the the mirror-plane.Such a plane can be defined by an origin and a normal vector, in the `mirrorMeshDict` as shown below. Patches about which the reflection is taking place are automatically removed.
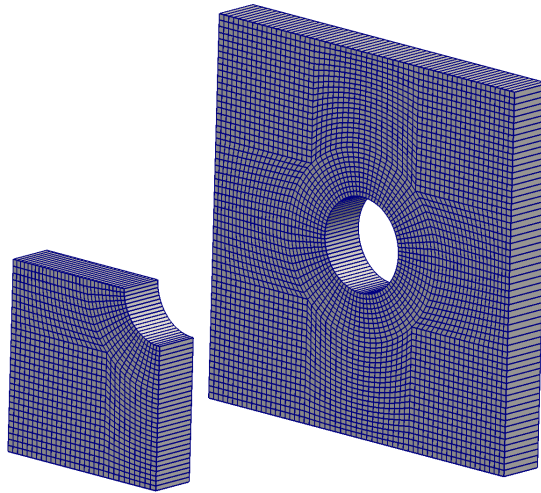
Figure 2.26: A 1/4 mesh before and after mirroring

```
pointAndNormalDict
{
    basePoint      (0 0 0);
    normalVector   (0 -1 0);
}
```

After this is defined properly, `mirrorMesh` can be executed and the mesh should be checked for errors:

```
?> mirrorMesh
?> checkMesh
```

This gives a half mesh. For the second mirroring, the dictionary must be changed to account for the different mirror plane:

```
pointAndNormalDict
{
    basePoint      (0 0 0);
    normalVector   (-1 0 0);
}
```

Again, run `mirrorMesh` and use `checkMesh` to check for any errors in the mesh itself:

```
?> mirrorMesh
?> checkMesh
```

This results in a full domain mesh, instead of a symmetric fraction as shown in figure 2.26.

2.5 Summary

There is a wide choice of open source applications available for designing the domain geometry, as well as for generating the mesh. On the other side, there are commercial solutions for the same purpose. Geometry definition and mesh generation sometimes involve a tedious workflow until the geometry is reduced to a level acceptable for simulation, and the generated mesh has sufficient quality. For simple cases, e.g. for validation, `blockMesh` is still a preferred mesh generation tool for many users. Automated mesh generation solutions such as `snappyHexMesh`, `cfMesh`, `foamHexMesh`, and `engrid` reduce the complexity and duration in mesh generation, but still involve handling different mesh generation parameters. Mentioned applications are distributed along with their own detailed documentation, so this chapter represents an effort to provide information on the way the mesh in OpenFOAM is composed, as well as additional information on mesh generation with `blockMesh` and `snappyHexMesh` to extened the currently available documentation.

Further reading

[1] Sept. 2014. URL: http://www.sourceflux.de/blog/adding-source-terms-equations-fvoptions/.

[2] May 2015. URL: http://www.sourceflux.de/blog/a-changing-cell-set-in-openfoam/.

[3] J. Höpken and T. Maric. *Feature handling in snappyHexMesh*. 2013. URL: www.sourceflux.de/blog/snappyhexmesh-snapping-edges.

[4] *OpenFOAM User Guide*. OpenCFD limited. 2016.

[5] E. de Villiers. *7th OpenFOAM Workshop: snappyHexMesh Training*. URL: http://www.openfoamworkshop.org/2012/downloads/Training/EugenedeVilliers/EugenedeVilliers-TrainingSlides.tgz (visited on 12/2013).

3

OpenFOAM Case setup

In this chapter, the structure of OpenFOAM simulations and the definition of boundary and initial conditions are covered.

> **INFO**
>
> Simulation test cases from this chapter are available in the example case repository, in the `chapter3` sub-directory.

3.1 The OpenFOAM simulation case structure

An OpenFOAM simulation is a directory with a set of files stored in different sub-directories. Some files are used to configure and control the simulation, others are used to store resulting simulation data. In general, the file-based organization of an OpenFOAM simulation case is relatively straightforward to use: the configuration files can easily be edited using a text editor. Using a file-based organization has one more advantage: the possibility to parameterize simulations easily.

The standard composition of an OpenFOAM case (simulation) is explained using the `cavity` tutorial case used with the `icoFoam` incompressible Navier-Stokes solver. Figure 3.1 shows a sketch of the `cavity` case setup. Other input files are present in the case directory, which
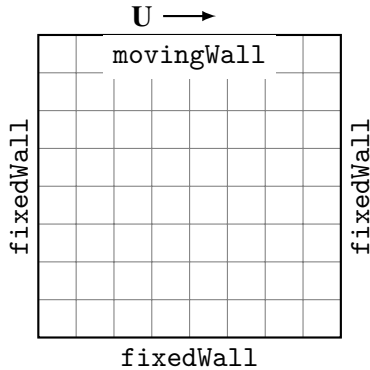
Figure 3.1: Sketch of the cavity

are not required by `icoFoam`, they are used by the pre-processing utilities. Generally, the number of required input dictionaries increases with increasing complexity of the solver.

Listing the sub-directories of the `cavity` case shows the way the simulation files are organized:

```
?> cd $FOAM_TUTORIALS/incompressible/icoFoam/cavity
?> ls *
0:
    p  U
constant:
    polyMesh  transportProperties
system:
    controlDict  fvSchemes  fvSolution
```

The 0, `constant`, and `system` directories are standard directories of OpenFOAM simulation cases that have to be present. The 0 directory holds the initial and boundary conditions applied on the fields. Each field used by the particular solver is represented by a text file, named after the field. For the `cavity` tutorial case, the fields used in the simulation are the pressure field p and the velocity field $\mathbf{U}$. How these fields are defined is shown in section 3.2.

As the simulation progresses, the solver application will write the resulting simulation data to new sub-directories of the case directory. Those directories are the so-called *time step* directories, and they are named based on the simulation time values. They not only contain those fields

already defined in the 0/ directory, but also auxiliary fields of the solver, such as the volumetric flux phi.

> **TIP**
>
> Although the icoFoam solver starts the simulation with the initial values for the pressure p and the velocity U fields, the FVM utilises the volumetric flux fields (phi) in the process of equation discretization (see chapter 1 for more details). As a result, the time step directories will hold the volumetric flux field phi computed by the solver application.

In addition to the time step directories, other simulation data may get written:

- by the solver application (e.g, the volumetric flux phi),
- by a function object running alongside the solver (see chapter 12),

- after the simulation has finished, as a result of post-processing (chapter 4).

The constant directory stores the simulation data that remains constant throughout the simulation. This typically includes the mesh data in the polyMesh sub-directory, as well as various configuration files:

- transportProperties - transport properties (described in chapter 11),
- turbulenceProperties - turbulence modeling (described in chapter 7),
- dynamicMeshDict - dynamic mesh controls (described in chapter 13),
- etc.

Not all aforementioned configuration files are present in the constant directory of the cavity case. One that is missing is the turbulenceProperties dictionary, which is not required by icoFoam. Which additional dictionaries have to be present, depend on the selected solver application. In case a solver is executed in a case directory, without the available necessary input data, the user is prompted with an informative error message, that notifies the user which dictionary, dictionary parameter or field is missing or ill defined.

The system directory holds all dictionaries relevant to the numerical methodology and the control of iterative solvers. A basic overview of

TIP

If a solver application uses the dynamic mesh feature, where point positions or mesh topology is changing, a new `polyMesh` folder is written to each time step directory as well. The dynamic mesh feature of OpenFOAM is discussed in chapter 13.

TIP

Configuration files in OpenFOAM are usually referred to as *dictionary files* or even shorter as *dictionaries*. This naming is due to their usage in the source code, which is based on the `IOdictionary` class.

those methods is provided in chapter 1. It may also hold dictionaries used to configure different pre- and post-processing applications such as `setFields`. The most important dictionary contained in the `system` directory, is the `controlDict`. It controls all parameters that relate to the run time of the solver and the frequency with which solution data is written to the case directory. All parameters defined in the `controlDict` are independant of the solver used for the simulation. More information on using the `controlDict` to control the simulation run is presented in section 3.4. A quite extensive discussion of the parameters in the `controlDict` is given in [5] and some of them are explained in section 3.4.1.

3.2 Boundary Conditions and Initial Conditions

OpenFOAM simulation is a directory that contains different sub-directories and files used to configure the simulation. This file structure of an OpenFOAM simulation makes setting both boundary and initial conditions very straightforward. Every physical quantity (pressure, temperature, velocity field, etc.) important for the simulation has its file, stored in the 0 directory: directory that belongs to the first time-step of the simulation. This section covers how boundary and initial conditions from section 1.3 are applied in practice in these files. Depending on the tensor rank of the field (scalar, vector, tensor), their respective values are set using a slightly different syntax. This discussion will be limited to case configuration; the numerical and design details of boundary conditions are discussed respectively in section 1.3 and chapter 10. Boundary conditions, as their name

implies, define the field values at mesh boundaries. Initial conditions refer to the initial values of the internal field. A sketch for the differentiation between internal and boundary values is shown in figure 1.12.

> **INFO**
>
> More information on the computational mesh can be found in chapter 2.1. The basics of the numerical method have been discussed in chapter 1. Both topics should be known to the reader, to a certain extend.

Before having a closer look at how boundary condition files are defined, the `cavity` case designed to be simulated with `icoFoam` has to be copied to a location of choice. To set a basic boundary condition for the `cavity` simulation case, the simulation case directory needs to be copied and renamed:

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity \
   cavityOscillatingU
?> cd cavityOscillatingU
```

Listing the `0/` directory reveals that there are two different fields defined: p and $\mathbf{U}$. Both files can be edited using any text editor, straight from the command line. The relevant lines of the pressure boundary condition file can be printed to screen using the following command:

```
?> cat 0/p | tail -n 23 | head -21
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            zeroGradient;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

This reveals three top level entries of the boundary condition file: *dimensions*, *internalField* and *boundaryField*. The first one is a set of scalar dimensions (`dimensionSet`), that is used to define the dimensions of the field. Each scalar corresponds to the power of the particlar SI unit, as defined in the declaration source file for the dimension set (`dimensionSet.H`):

```
//- Define an enumeration for the names of the dimension exponents
enum dimensionType
{
    MASS,               // kilogram   kg
    LENGTH,             // metre      m
    TIME,               // second     s
    TEMPERATURE,         // Kelvin     K
    MOLES,              // mole       mol
    CURRENT,            // Ampere     A
    LUMINOUS_INTENSITY  // Candela    Cd
};
```

The next entry is *internalField*, which defines the initial conditions for the field. Note that this does not include the boundaries, which are defined by the last subdictionary: *boundaryField*. In this example, all cell values are set to 0. It is also possible to define initial values on per-cell basis, which in turn requires the user to compose a list with the desired value for each cell. This list must have as many elements as cells are present in the mesh and its composition is explained in [5]. Lastly, the boundary conditions are defined, inside the *boundaryField* subdicitonary. Boundary conditions must be specified for each patch and each field. Thus the boundary condition for each patch is defined in a subdictionary of the `boundaryField` dictionary.

3.2.1 Setting Boundary Conditions

Already the official release of OpenFOAM comes with a multitude of boundary conditions. The `foamHelp` command, when executed within a case directory, provides a list of all the boundary conditions available. It accounts for the field type and only shows those boundary conditions that are compatible with the respective field. Some boundary conditions shown in the resulting list are very general, while others are very problem-specific. For the velocity field U, it can be called in the following way:

```
?> foamHelp boundary -field U
```

The `foamHelp` requies the mesh to be generated when checking available boundary conditions. As a small example for defining boundary conditions, the previously copied `cavityOscillatingU` case is used. For this example case the `movingWall` boundary condition for the velocity field U is to be changed to a boundary condition that ensures the velocity that oscillates in time (figure 3.1). To set the oscillating velocity we'll use the `codedFixedValue` boundary condition. This boundary condition allows its user to program C++ code that defines how the boundary condition behaves. This C++ code is then compiled by OpenFOAM into a library in the simulation case, stored in the `dynamicCode` sub-folder. The library is linked dynamically to the solver application and used in the simulation. Since it is C++ code that implements the boundary condition, the user of the boundary condition should know how to perform basic operations in OpenFOAM with scalars, vectors or tensors for their respective fields. The C++ programming knowledge is covered in the second part of the book. Generally, the information about the usage of a boundary condition, function object, and other OpenFOAM components is available in the Extended Code Guide, also covered in chapter 5. Alternatively, reading the header file of the boundary condition in the OpenFOAM source code is usually faster. The declaration of the `codedFixedValue` boundary condition in the file must be opened with a text editor. This file is located at:

```
$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/\
codedFixedValue/codedFixedValueFvPatchField.H
```

The description section of the large comment section at the top of the header file contains a description of usage for the boundary condition, that we adapt here for the `movingWall` patch:

```
movingWall
{
    type    codedFixedValue;
    value   uniform (0 0 0);
    name    oscillatingFixedValue;   // name of generated BC

    code
    #{
        operator==(Foam::cos(this->db().time().value() * M_PI)*vector(
            1,
            0,
            0
        ));
    #};
}
```

In the above C++ code snippet of the coded boundary condition, we have overloaded the OpenFOAM field assignment operator, to set the velocity vector to $cos(\pi t)(1, 0, 0)$, making it switch direction once per second. In addition to the above changes to the U field, the simulation time has to be extended so that several velocity oscillation cycles can be visualized. In order to do this, the system/controlDict dictionary has to be changed: the entry endTime is modified from 0.5 to 4.0. All required adjustments have been done and the changed boundary condition can be tested using the icoFoam solver. Before doing so, the mesh must be generated with the blockMesh utility:

```
?> blockMesh
?> icoFoam
```

Visualizing the velocity field can be done using paraView.

> **INFO**
>
> To visualize OpenFOAM cases with ParaView, create an empty file named case.foam (actually, any name with *.foam extension works) and open this file with ParaView.

The internal flow pulsates driven by the oscillating boundary condition.

Careful consideration is always required when selecting boundary condition, especially when comparing results to experiments (validation) or exact solutions (verification). The OpenFOAM tutorial cases in $FOAM_TU-TORIALS contains example simulation cases that use different boundary conditions. The tutorial cases can be a useful source of information about the usage of different bouncary conditions.

3.2.2 Setting Initial Conditions

In this section, an overview of how to set initial conditions of a simulation is covered. Initial conditions define *internal* field values at the beginning of the simulation. There is a variety of different tools that initialize the fields according to the user specifications. Chapter 8 provides information on how to develop new pre-processing applications.

> **INFO**
>
> Setting initial conditions (IC) is known as *pre-processing*.

The initial conditions can be trivial, like in the cavity case: the initial internal velocity and pressure are set to a uniform vector value of **0**. After the first timestep, the computed flow solution is usually quite different than the one initially set. For a stationary single-phase incompressible flow, the initial conditions can be considered as an initial guess that speeds up convergence to the steady state. Should this guess be excessively far from the appropriate solution, solver divergence may occur.

In other situations, the initial conditions are crucial, because they determine how the field evolves in time from the initial values. Compressible flow simulations rely heavily on the initial pressure, temperature, and/or density to properly compute an equation of state. Incompressible multiphase simulations require very accurately initial values for the phase indicator field that separates the fluid phases. Large errors in the initial phase indicator cause even larger errors in curvature approximation, leading to strong numerical instability and likely catastrophic failure.

As stated previously, the `cavity` example case is relatively tolerant towards the definition of the initial conditions. A case that has special requirements on the initial conditions is the `damBreak` case, simulated with the `interFoam` solver. The solver `interFoam` is a two-phase flow solver, using an algebraic Volume-of-Fluid method to distinguish between the two immiscible and incompressible fluid phases. For this purpose a new scalar field is introduced: `alpha.water`. The gas and the liquid phase in this example will have `alpha.water` values of 0 and 1, respectively. For the first example, a water droplet will be added to the `damBreak` case, as shown in figure 3.2.

First, a copy of the `damBreak` tutorial case is made and the case is renamed:

```
?> run
?> cp -r $FOAM_TUTORIALS/multiphase/interFoam/\
laminar/damBreak/damBreak damBreakWithDrop
```

Original field values are kept usually in a sub-folder 0, or in `*.orig` field files in the 0 folder. To reset `alpha.water` initial values, `0/alpha.water` file is copied to `0/alpha.water`.

```
?> cp 0/alpha.water 0/alpha.water
```

Inspecting the current state of the `0/alpha.water` field at this point, shows that the entire internal field has a uniform value of 0:

```
internalField   uniform 0;
```

> ```
> INFO
> ```
> Some tutorials will have field files such as `phi` available in the 0
> directory. Storing field files in their original state enables the user to
> quickly reset initial field values. Alternatively, a version control system
> might be used for the simulation case directory - more information on
> this can be found in chapter 6.

The `setFields` utility can be used to create more complex, non-uniform
field values. This utility is controlled by the `system/setFieldsDict`
dictionary. The template dictionary file for the `setFields` application is
stored in the application source directory: `$FOAM_APP/utilities/pre-`
`Processing/setFields`. As this file is fairly long, the contents are not
shown. However, it is worthwhile to investigate the contents of `set-`
`FieldsDict`, as it stores all the available specifications that can be used
by the `setFields` application to pre-process non-uniform fields.

The contents of the `setFieldsDict` for the example test case are shown
below, together with the added `sphereToCell` sub-dictionary entry used
to initialize a small droplet, i.e. a circle with the `alpha.water` value of
1:

```
defaultFieldValues
(
    volScalarFieldValue alpha.water 0
);

regions
(
    boxToCell
    {
        box (0 0 -1) (0.1461 0.292 1);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
    sphereToCell
    {
        origin (0.4 0.4 0);
        radius 0.05;
        fieldValues
        (
            volScalarFieldValue alpha.water 1
            volVectorFieldValue U (-1 0 0)
        );
    }
);
```
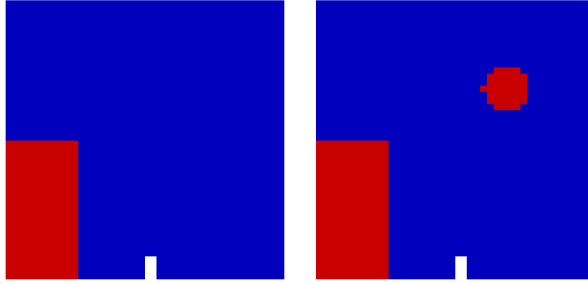
Figure 3.2: A side by side of the initial conditions before and after adding the droplet.

The `defaultFieldValues` will set the internal field to the provided default value 0, before proceeding to process the `regions` sub-dictionary. The syntax of `setFieldsDict` is mostly self-explanatory for simple shape-volume selections like those shown above. For a chosen volume, all cells whose cell center is within this volume will have the given fields set accordingly. For example, the `sphereToCell` source selects cells whose centers are within the sphere of a specified center and radius, and sets the field values $\alpha_{\text{water}} = 1$ and $\mathbf{U} = (-1, 0, 0)$.

In order to pre-process the `alpha.water` field and run the simulation, the following steps need to be executed in the command line:

```
?> blockMesh
?> setFields
?> interFoam
```

An illustration of the initial conditions which were changed due to our additions to the `setFieldDict` is shown below in figure 3.2.

3.3 Discretization Schemes and Solver Control

Choosing a discretization scheme as well as adjusting control parameters for the linear solvers are as important selecting boundary conditions. Discretization schemes are defined in `system/fvSchemes` and the solver is controlled by the `system/fvSolution` dictionary.

3.3.1 Numerical Schemes (fvSchemes)

From a user perspective, all definitions related to unstructured Finite Volume discretization and interpolation are defined in the `system/fvSchemes` dictionary. The settings required differ from solver to solver and are dependent on the formulation of the particular terms of the mathematical model. Discretization and interpolation schemes are used within the framework of the FVM to discretize the terms of the mathematical model. In OpenFOAM, the mathematical model is defined in the solver application, using the Domain Specific Language (DSL). DSL has been developed as the abstraction level the algorithmic implementation became higher and higher with time. In other sources of information regarding OpenFOAM, the OpenFOAM DSL is often referred to as *equation mimicking*. Using algorithms in a higher level of abstraction with equation mimicking allows a very human-readable definition of mathematical models, as well as a trivial modification to the mathematical model. For example, the following source code snippet from the `icoFoam` solver application shows the OpenFOAM implementation of the momentum conservation law equation:

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);

solve(UEqn == -fvc::grad(p));
```

The different terms of the momentum equation are easily recognized, added, deleted or modified. For each of these terms an equivalent discretization needs to be defined in `system/fvSchemes`.

> **INFO**
>
> A Domain Specific Language (DSL) is developed on purpose for some applications, and for others, it is a natural consequence of software development with a clear separation of abstraction layers. Thinking in terms of discretized equations, matrices, and source terms, and *not* in terms of e.g. iteration loops, variable pointers and functions, represents the foundation of equation mimicking / DSL in OpenFOAM. Separating levels of abstraction is a sign of good software development practices.

The discretization and interpolation schemes are the building blocks of operations listed in the source code above. They convert differential local equations into algebraic equations, with properties averaged in finite volumes. Those terms, that take part in the assembly of the algebraic system are named *implicit terms*. Terms that are explicitly evaluated (located on the r.h.s. of the equation), are named *explicit terms*. Explicit terms are evaluated with the discrete operators from the `fvc` (*finite volume calculus*) namespace, whereas implicit terms are evaluated using the operators from the `fvm` (*finite volume method*) namespace. It is important to memorize this difference, since the discretization scheme defined for a term in the `system/fvSchemes` dictionary file will then be used as default for both `fvc::` and `fvm::` operators in the application code.

To get a list of the supported schemes for a specific term, simply replace the existing scheme in the `fvSchemes` dictionary file by *any word* and execute the solver. The error returned by the solver complains that a scheme by the name chosen on purpose, does not exists. But this is followed by a large list, showing all schemes available. In order to get the information about the scheme parameters (scheme-specific keywords and values), the user needs to browse through the source files related to the scheme implementation.

The equations implemented in the solver application are defined at *compile-time*. However, the choice of the schemes used to discretize the terms of those equations is performed at *runtime*. This allows the user to modify the way the mathematical model is discretized: choosing different schemes for different cases, by modifying the entries in `system/fvSchemes`.

> **TIP**
>
> Discretization and interpolation schemes are generic algorithms in OpenFOAM. To distinguish between the implicit algorithms (matrix assembly) and the explicit algorithms (source terms), the algorithms are categorized into C++ *namespaces*. A namespace is a programming language construct that is used to avoid name lookup clashes (more details given by [6])

The `system/fvSchemes` dictionary of the `cavityOscillating` is

```
ddtSchemes
{
    default      Euler;
}
```

```
gradSchemes
{
    default        Gauss linear;
    grad(p)        Gauss linear;
}

divSchemes
{
    default        none;
    div(phi,U)     Gauss linear;
}

laplacianSchemes
{
    default        Gauss linear orthogonal;
}

interpolationSchemes
{
    default        linear;
}

snGradSchemes
{
    default        orthogonal;
}
```

Every `fvSchemes` dictionary has the same 7 subdictionaries and most likely a default parameter defined at the beginning of each subdictionary. Comparing the code snippet of the momentum equation as it is implemented in `icoFoam` to the above listing shows that in this configuration, `div(phi,U)` is discretized using the `linear` method. In the following, an overview of different discretization scheme categories, and characteristics of selected schemes is provided.

> **INFO**
>
> If the following description of selected schemes is not clear after the first read, it doesn't impact the understanding of the rest of the book. Those interested may return to this part of the book later and read it again alongside chapter 1. Describing all of the available schemes in OpenFOAM is outside of scope for this book, so only a few selected schemes were addressed.

The schemes are sorted into categories that correlate to the terms of the mathematical model as well as the entries in `system/fvSchemes`:

- `ddtSchemes`

- `gradSchemes`
- `divSchemes`
- `laplacianSchemes`
- `interpolationSchemes`
- `snGradSchemes`

INFO

The so-called *Run-Time Selection* in OpenFOAM enables the selection different schemes in `system/fvSchemes`. This module outputs a list of all available schemes if the entry in `system/fvSchemes` is wrong. This can be used to learn what schemes are available: entering a name of a non-existing scheme prompts the RTS system to provide a list of available schemes.

ddtSchemes

The `ddtSchemes` are schemes used for temporal discretization. The Euler first-order temporal discretization scheme is set as a default value for transient problems, and it happens to be the scheme used to show the discretization practice of the FVM in chapter 1. Alternative choices of the temporal discretization schemes are:

- `CoEuler`,
- `CrankNicolson`,
- `Euler`,
- `SLTS`,
- `backward`,
- `bounded`,
- `localEuler`, and
- `steadyState`.

Of the available schemes for the temporal discretization, `CoEuler` and `backward` are explained more thoroughly in the following:

CoEuler scheme is a first order temporal discretization scheme that can be used for both implicit and explicit discretization. It automatically adjusts the time step locally so that the local Courant number is limited by a user specified value. The scheme computes a local time step field using the current time step value and a scaling coefficient

field computed from the local Courant number field. The reciprocal of the Courant number limited local time step is calculated as

$$\delta t_{fCo}^{-1} = max(Co_f/maxCo, 1)\delta t^{-1}, \qquad (3.1)$$

where the local face centered Courant number is computed from the volumetric flux F, the face area normal vector $\mathbf{S}_f$, and the distance between the cell centers of the cells connected at the face $\mathbf{d}$:

$$Co_f = \frac{\parallel F \parallel}{\parallel \mathbf{S}_f \parallel \parallel \mathbf{d} \parallel}\delta t. \qquad (3.2)$$

In case the mass flux is provided, the density field needs to be used to compute the local face-centered Courant number using

$$Co_f = \frac{\parallel F_\rho \parallel}{\rho_f \parallel \mathbf{S}_f \parallel \parallel \mathbf{d} \parallel}\delta t, \qquad (3.3)$$

where F_ρ is the mass flux stored at the face center. Once the face centered Courant number is computed, the reciprocal of the time step is computed using equation 3.1. For a face-centered field ϕ_f, the reciprocal of the face-centered time step is then used to compute the first-order temporal derivative:

$$\delta t_{fCo}^{-1}(\phi_f^n - \phi_f^o). \qquad (3.4)$$

Since the temporal schemes operate mostly on cell centered fields, a cell-centered value for the reciprocal of the time step is computed as a maximum of the face values, i.e.

$$\delta t_c^{-1} = max_f(\delta t_{fCo}^{-1}), \qquad (3.5)$$

which is then used to compute the temporal derivative of the cell-centered field ϕ_c:

$$\delta t_{cCo}^{-1}(\phi_c^n - \phi_c^o). \qquad (3.6)$$

This way, using a locally computed field for the reciprocal of the time step δt^{-1}, the temporal derivative is evaluated locally. Local estimation of the temporal derivative based on the local Courant number allows for larger time steps to be applied in those parts of the flow domain, where the Courant numbers are lower. Thus the simulation speed is increased (numerical time is artificially going faster in those parts of the domain, since the flow there is expected to experience a smaller amount of change). Limiting is enforced based on the local Courant number, because by doing so, the numerical stability of the solution is enforced.

backward scheme or Backward Differencing Scheme of second order (BDS2), uses the field values from the current and the two successive old time step fields to assemble a second order convergent temporal derivative. The derivative is computed using a Taylor series expansion starting at the current time, going back two time steps:

$$\phi^o = \phi(t - \delta t) = \phi(t) - \phi'(t)\delta t + \frac{1}{2}\phi''\delta t^2 - \frac{1}{6}\phi'''\delta t^3 + \ldots \quad (3.7)$$

$$\phi^{oo} = \phi(t - 2\delta t) = \phi(t) - 2\phi'(t)\delta t + \frac{1}{2}\phi''4\delta t^2 - \frac{1}{6}\phi'''\delta 8t^3 + \ldots \quad (3.8)$$

Multiplying equation 3.7 with 4 and subtracting equation 3.8 from that, results in the BDS2 temporal discretization of a cell-centered field ϕ_c:

$$\phi_c' \approx \frac{\frac{3}{2}\phi_c - 2\phi_c^o + \frac{1}{2}\phi_c^{oo}}{\delta t}. \quad (3.9)$$

The calculation of the derivative using old values is done in Open-FOAM without requiring the client code to store old field and mesh values. All volumetric and face centered fields have the ability store the values from the old (o) and the old-old (oo) time step values automatically. So does the mesh itself store information needed for the temporal discretization of higher order (e.g. old and old-old mesh volume fields). The `backward` scheme computes a second order temporal derivative which may be explicitly evaluated, since the o and oo field values are known. The actual implementation takes into account the possible adjustment of the time step, resulting in the derivative of the cell-centered fields as

$$\phi_c' \approx \frac{c_t\phi_c - 2c_{t^o} + c_{t^{oo}}\phi_c^{oo}}{\delta t}, \quad (3.10)$$

where c coefficients are defined in the following way:

$$c_t = 1 + \frac{\delta t}{\delta t + \delta t^o}, \quad (3.11)$$

$$c_{t^{oo}} = \frac{\delta t^2}{\delta t^o(\delta_t + \delta_t^o)}, \quad (3.12)$$

$$c_{t^o} = c_t + c_{t^{oo}}, \quad (3.13)$$

and in case of the two sub-sequent time steps being equal, the c coefficients correspond to the discretization using a constant time step, compare equation (3.9).

gradSchemes

The `gradSchemes` determine which gradient evaluation schemes are used for the terms defined in the solver. There are many examples in practice where the choice of the gradient scheme may lead to a better solution. For example, the gradient is used to compute the surface tension force in two-phase flow simulations, so it plays an important role in flows driven by surface tension. Whenever steep gradients are present, or different mesh topology is applied (e.g. tetrahedral meshes), a gradient scheme using a wider stencil of cells may provide better solutions.

Available gradient schemes in OpenFOAM are:

- `Gauss`
- `cellLimited`
- `cellMDLimited`
- `edgeCellsLeastSquares`
- `faceLimited`
- `faceMDLimited`
- `fourth`
- `leastSquares`
- `pointCellsLeastSquares`

Three of these schemes are selected and described in the following: `Gauss`, `cellLimited` and `pointCellLeastSquares`.

Gauss is the most often used discretization scheme for the gradient, the divergence (convective), and laplacian (diffusive) terms. It is also described in chapter 1. This scheme expects face centered values in order to compute the cell-centered gradient.

cellLimited extends the functionality of the standard gradient scheme. It computes a limiter for the cell centered gradient value, computed in the standard way and then scales the gradient value with this limiter ($0 < l \leq 1$). The limiter is calculated using the maximal and minimal values (ϕ_{cmax}, ϕ_{cmin}) of a cell centered property ϕ_c, found by indirectly searching through the face adjacent cells as

$$\phi_{cmax} = \max_C(\phi_c), \tag{3.14}$$

$$\phi_{cmin} = \min_C(\phi_c), \tag{3.15}$$

where ϕ is the cell-centered property and C is the set of all face-adjacent cells of a cell c. Once the minimal and maximal values in the face-connected cell stencil are found, the maximal and minimal values are used to compute the value differences using the original cell centered value ϕ_c:

$$\Delta\phi_{max} = \phi_{cmax} - \phi_c, \tag{3.16}$$

$$\Delta\phi_{min} = \phi_{cmin} - \phi_c. \tag{3.17}$$

The differences are then increased with the user specified coefficient $(0 < k < 1)$ as

$$\Delta\phi_{max} = \Delta\phi_{max} + \left(\frac{1}{k} - 1\right)(\Delta\phi_{max} - \Delta\phi_{min}) \tag{3.18}$$

and

$$\Delta\phi_{min} = \Delta\phi_{min} - \left(\frac{1}{k} - 1\right)(\Delta\phi_{max} - \Delta\phi_{min}). \tag{3.19}$$

Note that the differences are not increased if the user specified coefficient k is set to 1.

TIP

The equations in this section rely on the cell-to-cell connectivity and they also use cell-based stencils. The actual implementation makes use of the owner-neighbor addressing, for reasons described in chapter 1.

The limiter is initially set to 1, but as it is computed by looping over the mesh faces using h, its value will be updated. This is why the min and max functions are comparing the ratio of the property difference given my minimal and maximal values with the *extrapolated difference given by the gradient*:

$$\Delta(\phi_c)_{grad} = \nabla(\phi)_c \cdot \mathbf{cf}, \tag{3.20}$$

where $\mathbf{cf}$ is the vector connecting the cell and the face center in question. The increased differences compared to the minimal/maximal values are then used to compute the limiter (l):

$$l = \begin{cases} min(l, \frac{\Delta\phi_{max}}{\Delta(\phi_c)_{grad}}) & : \Delta\phi_{max} < \Delta(\phi_c)_{grad} \\ min(l, \frac{\Delta(\phi_c)_{grad}}{\Delta\phi_{min}}) & : \Delta\phi_{min} > \Delta(\phi_c)_{grad} \end{cases} \tag{3.21}$$

Determining the final gradient in the cell center, the gradient computed by the standard scheme is scaled with the limiter

$$\nabla(\phi)_c = \nabla(\phi)_c \cdot l. \tag{3.22}$$

pointCellsLeastSquares provides a larger stencil for computing the gradient. It introduces the neighboring cells, that are adjacent to the cell in question, not just across cell faces, but also across points. On an unstructured hexahedral mesh, this stencil would contain 3^3 cells. Although the numerical schemes implemented in OpenFOAM are *second order convergent*, when sharp jumps are present in the flow domain, *absolute accuracy* becomes another very important aspect of the numerical approximation. Using a wider stencil, this scheme provides an estimation of the gradient with lower absolute errors. If a linear Taylor expansion for a property ϕ around the cell center c is used:

$$\phi(\mathbf{x}) = \phi_c - \nabla(\phi)_c \cdot (\mathbf{x} - \mathbf{x_c}) + O(\| \mathbf{x} - \mathbf{x}_c \|^2) \tag{3.23}$$

three values are needed to compute the three unknown variables: the components of the gradient $\nabla(\phi)_c$. The least squares gradient involves expanding the Taylor series (equation 3.23) from the center of the cell where the gradient is computed to neighboring cells. The number of expansions is directly proportional to the size of the cell stencil. For two-dimensional triangular meshes, the gradient can be computed directly from the surrounding three cells from the face-neighbor stencil. However, including more cells in the stencil increases the absolute accuracy of the gradient. When more than three point values are known, the system becomes overdetermined. Consequently the gradient is computed using a minimization of the squared gradient error defined as

$$E = \sum_{cc} w_{cc}^2 E_{cc}^2, \tag{3.24}$$

where cc are the cells of the stencil. w_{cc} is the weighting factor relating the cell in question and the cell of the stencil. Usually an inversed distance between two cells is taken as the weight and E_{cc}^2 is the squared error of the Taylor expansion from the cell for which the gradient is computed to the cell of the stencil:

$$E_{cc} = \frac{1}{2} \nabla\nabla(\phi)_c : (\Delta\mathbf{x}\Delta\mathbf{x}). \tag{3.25}$$

After some algebraic manipulation ([4]), the minimization of the squared error results in a linear algebraic system for the gradient components. The linear algebraic system size is assembled and solved for each cell, since the system dimensions are 3×3. Furthermore, the solution is obtained by inverting the coefficient matrix per cell directly since the coefficient matrix will be a symmetric tensor. As a motivation for using this gradient scheme, consider figure 3.3, where the gradient of the volume fraction field is computed using the standard `Gauss linear` and the new `pointCellsLeastSquares` gradient scheme. The volume fraction field is set as a circle of radius $R = 2cm$ centered in the domain of 6×6 cm, discretized using a mesh with 30×30 volumes ([1]). The `pointCellsLeastSquares` computes a more *uniform* gradient of a circular droplet compared to the `Gauss linear`, since the point neighbors are involved in the gradient calculation. Involvement of point neighbors increases the absolute accuracy for fields with sharp jumps.

INFO

The Gauss gradient calculation shown for the example field in figure 3.3 is a textbook example of *mesh anisotropy* - the calculation strongly depends on the orientation of faces with respect to the coordinate axes.

INFO

As an exercise, try computing the gradient of steep explicit functions on different meshes and compare the error convergence for different gradient schemes.

divSchemes

The `divSchemes` are used for discretizing any convective (divergence) term in the mathematical model. Viewing the `system/fvSchemes` dicitonary of the `cavityOscillating` case shows that the spatial discretization scheme most often encountered for the divergence terms is the `Gauss` discretization scheme, which uses the Gauss-Ostrogradsky divergence theorem. The discretization scheme presents the basis of the FVM, since it

(a) `Gauss linear` gradient

(b) `pointCellsLeastSquares` gradient
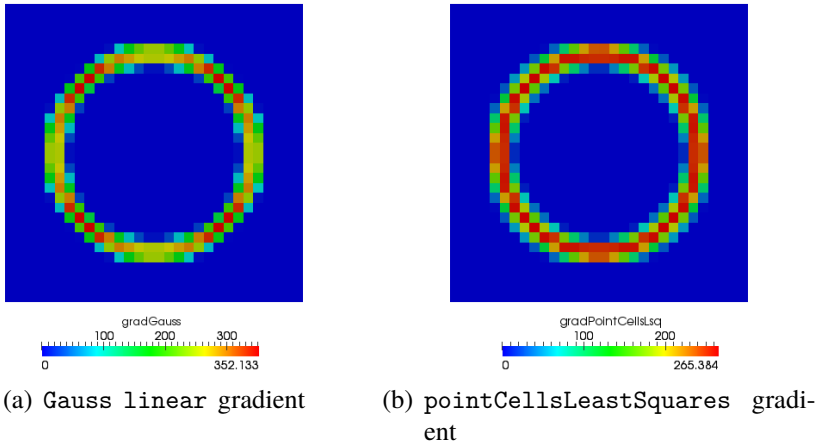
Figure 3.3: Comparing the gradient of a circular volume fraction field

produces a system of algebraic equations, which makes it unlikely to be changed in the configuration file for implicit terms. Additionally, the `bounded` option may be provided for simulations involving steady-state, or partially converged solutions where $\nabla \cdot (\mathbf{U}) = 0$ is not exactly satisfied, during the iterations of the solution algorithm. In that case, the term $\nabla \cdot (\mathbf{U})$ term is deducted from the coefficient matrix to improve the solution convergence.

laplacianSchemes

The `laplacianSchemes` subdictionary in the `system/fvSchemes` dictionary is assembled from `Gauss` discretization, and an `interpolation-Scheme`. `Gauss` discretization is the base choice for the discretization practice for diffusive (laplacian) terms, with the choice of the interpolation scheme left available to the user.

interpolationSchemes

The `interpolationSchemes` change the way the face centered values are interpolated. Different interpolation schemes may be chosen together

with the spatial discretization scheme, such as the Gauss discretization scheme. On the other hand, the values on the faces may be interpolated using a larger set of point-value-pairs. Thus increasing the accuracy of the interpolation, and the Gauss discretization scheme will then be used to discretize the divergence, laplacian, or gradient term using face interpolated values.

OpenFOAM provides a very large variety of choices when it comes to interpolation schemes: `biLinearFit`, `blended`, `clippedLinear`, `CoBlended`, `cubic`, `cubicUpwindFit`, `downwind`, etc. The majority of the available interpolation schemes are used for discretizing the convective term. Applying specific numerical schemes introduces different numerical errors in the convection, especially if the convected property has a jump in the solution domain. For example, it is commonly known the Central Differencing Scheme (CDS, `linear`) introduces numerical instabilities in the solution, and the Upwind Differencing Scheme (UDS, `upwind`) smooths the convected field artificially ([2], [7]). As a result, different numerical schemes were developed to counter the negative effects of the original schemes, either involving higher order interpolations, or computing the final interpolated value as a combination of values obtained by other interpolation schemes.

3.3.2 Solver Control (fvSolution)

The discretized formulation of the mathematical model leads to a system of algebraic equations of shape $\mathbf{Ax} = \mathbf{b}$. This system must be solved somehow, which can be done using either direct or iterative methods. Direct methods have very high computational costs, when solving large sparse matrices, as pointed out by [2]. All settings related to the solving process of such matrix equations, as well as pressure-velocity coupling are done in the `system/fvSolution` dicitonary. Depending on the solver used, the contents of that file change. For the previously used `cavity` tutorial of the `icoFoam` solver, the following definitions in the `fvSolution` dictionary are preset:

```
solvers
{
    p
    {
        solver        PCG;
        preconditioner DIC;
        tolerance     1e-06;
```

```
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}
```

This dictionary contains two main subdictionaries: `solvers` and `PISO`. The `solvers` subdictionary needs to be present for all solvers, as it contains the choice and parameters for the various *matrix* solver used. It is important to distinguish the matrix solvers from the solver applications, that implement a certain mathematical model. The second dictionary stores parameters needed by the PISO pressure-velocity coupling algorithm. There are other pressure-velocity coupling algorithms in OpenFOAM, such as SIMPLE and PIMPLE. Each of them requires a dictionary with the particular algorithm name to be present in `fvSolution`.

INFO

There are two types solvers in OpenFOAM: *solver applications* and *linear solvers*. Solver applications are programs used directly by the end-user for running simulations. Linear solvers are algorithms used to solve linear algebraic equation systems as a part of the simulation solution procedure. Usually, the solver applications are named "solvers" in short, so this term is used for solver applications from this point on.

Linear Solvers

The example shown above, uses `PCG` to solve the pressure equation and the matrix is preconditioned using `DIC`, which can be used for symmetric

matrices. For the momentum equation on the other hand, the asymmetric `PBiCG` solver is used, in conjunction with a `DILU` preconditioner.

Regardless of the matrix solver, the following parameters can be defined:

tolerance defines the exit criterion for the solver. This means that if the change from one iteration to the next is below this threshold, the solving process is assumed to be sufficiently converged and stopped. When performing e.g. steady state simulations of a steady problem, the tolerance should be quite small, in order to improve the convergence and accuracy. For transient problems on the other hand, no steady solution can be obtained and hence the tolerance must not be chosen to be very small.

relTol defines the relative tolerance as the exit criterion for the solver. If it is set to some other value than 0, it overrides the tolerance setting. Whereas `tolerance` defines the *absolute* change between two consecutive iterations, `relTol` defines the *relative* change. A value of 0.01 forces the solver to iterate until a change of 100% is reached, between two consecutive iterations. This comes in handy, when a strongly unsteady system is simulated and the tolerance setting leads to high iteration numbers.

maxIter is an optional parameter and has a default value of 1000. It defines the maximal number of iterations, until the solver is stopped anyway.

Pressure-Velocity Coupling

Depending on the solver application selected, a different pressure-velocity coupling algorithm is implemented in the respective solver. The solver application tries to read the subdictionary in `fvSolution`, depending on the pressure-velocity coupling algorithm implemented.

> **TIP**
>
> For more background information on the various algorithms, the reader is referred to [2, 7]. There is plenty of information available on the OpenFOAM wiki as well.

There are some parameters that have to be defined anyway:

nNonOrthogonalCorrectors parameter defines a number of internal loop
cycles which are used to correct for mesh non-orthogonality. This
parameter is necessary when e.g. tetrahedral meshes are used or
when local dynamic adaptive mesh refinement is applied on hex-
ahedral meshes. The effects of non-orthogonality to the equation
discretization are described in detail by [3] and [2]. The internal
loop is introduced because the correction is explicit. It needs to be
applied multiple times, as each application reduces the errors and
does not completely remove them.

pRefPoint/pRefCell Either of these options define the location where
the reference pressure is assumed to be located. `pRefPoint` takes
a `vector` in the mesh coordinate system, whereas `pRefCell` takes
just the label of the cell where the pressure should be located. For
multiphase flows, the point should always be covered by either of
the phases and not changing in between. These are only required
if the pressure is not fixed by any boundary condition.

pRefValue defines the reference pressure value and is usually zero.

Other parameters may be required by the particular pressure-velocity cou-
pling algorithm or the solver. The more complex the solver is, the more
options are usually required to be present in `fvSolution`. Tutorials for
the solvers usually provide sufficient information to get a case running.

3.4 Solver Execution and Run Control

Executing a solver is as easy as issuing any other command under Linux:
just type the command name and hit enter. As a general rule of thumb,
solvers and any other OpenFOAM utilities should be executed right in-
side the case directory. Executing a solver from some other directory,
requires to pass the `-case` parameter to the solver, followed by the path
to the case. For the `cavity` case, simulated with the `icoFoam` solver, the
command looks as such:

```
?> icoFoam
```

Depending on the mesh size, the time step and the total time that is to
be simulated this command does take significantly longer, as the usual `ls`
or `cp` command. Additionally the information printed to the terminal is

massive and gets lost as soon as the terminal is closed. Hence the syntax to execute the solver should be extended:

```
?> nohup icoFoam > log &
?> tail -f log
```

The `nohup` command instructs the shell to keep the job running, even when the shell window is closed or the user logged out. Rather than printing everything to the screen, the output is piped into a file called `log` and the job is moved into the background. As the job is running in the background and all output is forwarded to the `log` file, the `tail` command is used to basically print the end of the file to the screen. By passing `-f` as a parameter, this is updated until the command is quitted by the user. An other option to keep up to date with the running job is to use the `pyFoamPlotWatcher`, which parses any log file and generates `gnuplot` windows from that. These windows are updated automatically and include residual plots, which is handy for monitoring a simulation.

Opposed to manually starting the solver in background and redirecting the output to a log file, the `foamJob` script can be used as well:

```
?> foamJob icoFoam
```

The `foamJob` script is quite powerful as it provides an all in one solution for starting OpenFOAM jobs. Some features of `foamJob` are covered in the next section.

3.4.1 controlDict Configuration

Each case must posses a `system/controlDict` file that defines all the runtime related data. Including the time when to stop the solver, the time step width, the interval and method how time step data is written to the case directory. The content of this dictionary is re-read automatically during the run time of a solver and hence allows for changes, while the simulation is running.

The *OpenFOAM User Guide* [5] provides an overall introduction to the parameters and parameter combinations, so this information is not repeated here. Two parameters are explained here in more detail, as they are referred to in the rest of the book.

writeControl denotes when data is written to disk. The most popular choices are `timeStep`, `runTime` or `adjustableRunTime`. The actual interval is defined by `writeInterval`, which only takes scalar values. For the sake of simplicity, this intervall is named n from now on.

If `timeStep` is chosen every n-th time step is written to disk, whereas chosing `runTime` writes every n-th second to the case directory. The last commonly used option is `adjustableRunTime`, which writes every n-th second to disk but adjusts the time step so that this interval is exactly matched. Hence only nicely named time directories occur in the case folder.

Using the default settings, the amount of data that is written to disk is not limited by OpenFOAM. Especially in cases where a lot of users access the same storage unit and long run times are common, this fills up the disks in no time.

purgeWrite can be used to circumvent the above mentioned issue with using excessive storage. By default it is 0 and does not limit the amount of time instances written to disk. Changing this to 2 instructs OpenFOAM to keep only the latest 2 time instances on disk and delete the other ones, each time data is written to disk. This option cannot be used with `writeControl` set to `adjustableRun-Time`.

In addition to these standard parameters, the `controlDict` also contains custom libraries that are linked to the solvers during runtime as well as calls to function objects. Function objects are covered in chapter 12.

3.4.2 Decomposition and Parallel Execution

Up to this point, all solvers were executed on a single processor. The structure of CFD algorithms makes them viable for data parallelism: the computational domain is divided into multiple parts, and the same task is executed in parallel on each part of the computational domain. Each process communicates with its neighbors and shares relevant data. With modern multicore architectures and HPC clusters, distributing the workload over multiple computing units usually results in an execution speed up in terms of execution time. The decomposition of the computational domain must never influence the numerical properties of the method: consistency, boundedness, stability and conservation. In OpenFOAM, the

data parallelism is achieved in a very elegant way, and it is closely tied to the underlying FVM. In effect, the boundary face that was used to elaborate the equation discretization in chapter 1 might as well have been a face of the processor (process) boundary. Before the simulation can be executed in parallel, the computational domain must be divided into as many subdomains as processes are used for the simulation.

> **INFO**
>
> The execution of a solver on a single processor core is often referred to as "serial execution" - the solver was executed "in serial".

As an example of parallel execution, the `cavity` simulation case of the `icoFoam` solver is distributed over two cores of the machine the job is started on. This does imply that the machine posses at least 2 cores, otherwise the simulation will take considerably longer than a serial run. In order to make use of the two cores, the data must get distributed between them.

> **INFO**
>
> Interaction between the interprocess communication (IPC) and the FVM on unstructured meshes in OpenFOAM is handled automatically for discrete differential operators, its details are relatively complex and outside of the scope for this book.

The $FOAM_RUN folder can be used to run OpenFOAM simulations in the $HOME folder, but it needs to be created

```
?> mkdir -p $FOAM_RUN
```

To run the simulation in $FOAM_RUN, make a copy of the `cavity` tutorial for the `icoFoam` solver:

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN/
?> run
?> cd cavity
```

The `run` alias switches to $FOAM_RUN directory. In the next step, a choice is made on the way the computational domain will be decomposed. Various methods exist in OpenFOAM for decomposing the domain. The `scotch` domain decomposition is used for this tutorial. The domain decomposition configuration is stored in the dictionary file `system/decomposeParDict`. You can find available `decomposeParDict` and similar files within OpenFOAM tutorials using

```
?> find $FOAM_TUTORIALS -type f -iname decomposeParDict
```

The `cavity` tutorial case does not contain this file by default. Just any `decomposeParDict` that is located somewhere in the tutorials of Open-FOAM is fine to copy to the local `cavity` case.

```
?> cp cp $FOAM_TUTORIALS/multiphase/interFoam/\
laminar/damBreak/damBreak/system/decomposeParDict ./system
```

The most important lines of `decomposeParDict` are the following:

```
numberOfSubdomains  4;

method              simple;
```

The first line defines how many sub-domains (or MPI processes) will be used, and the second selects the method that should be used to decompose the domain. Using the `simple` method is generally a bad choice for real world applications, because it splits the domain in spatially equal parts. This happens without minimizing the size of the inter-process communication boundaries between the sub-domains. Considering a mesh that is very dense on one side and very coarse on the other, the simple method would slice it in half, producing sub-domains with very uneven distribution of finite volumes. The mesh with more finite volumes requires more computational resources, making the simulation imbalanced. Increased inter-process communication time caused by unnecessarily large inter-process communication boundaries can severely decrease the efficiency of the parallel execution.

To optimize this, various automatic decomposing methods exist, that e.g. optimize the processor boundaries in order to result with a minimal inter-process communication overhead. By changing the lines in `system/decomposeParDict` as follows, it is ensured that the domain is decomposed in two sub-domains and that the method `scotch` is employed for that:

```
numberOfSubdomains  4;
method              scotch;
```

Before decomposing the domain, the mesh must be generated. This example uses a `blockMesh` based mesh, which must hence be executed. Once the mesh is generated, the domain decompositioning using `decomposePar` must be executed:

```
?> blockMesh
?> decomposePar
```

Two new directories are generated in the case directory: `processor0` and `processor1`. Each of them contains a sub-domain, including the mesh (in the `processor*/polyMesh` directory) and the fields (in the `processor*/0` directory). The actual command to start the simulation in parallel is a little bit longer than serial command, because MPI needs to be used for that:

```
?> mpirun -np 2 icoFoam -parallel > log
```

This invokes `mpirun` with 2 subprocesses to run `icoFoam` in parallel and store the output in `log` for later assessment. The `-parallel` parameter of `mpirun` is really important because it instructs `mpirun` to run each process with its particular subdomain. If this parameter would be missing, 2 processes would get started but both employ the entire domain, which is not only redundant but also a waste of computational power.

When the simulation has finished, the subdomains can be reconstructed into one single domain. For this the `reconstructPar` tool is used that - by default - takes all time instances from each processor directory and reconstructs them. Only the last time step can be selected, when `reconstructPar` is called with the optional argument `-latestTime`. It instructs `reconstructPar` to only reconstruct the latest time. This comes in handy when the mesh is pretty large and the reconstruction takes a lot of time and disk space. The reconstruction of sub-process domains is not necessary, as ParaView can visualize decomposed OpenFOAM cases.

Though one might observe a good scaling of the execution time with the number of processors that are used, it is usually a bad idea to go below a certain total cells to subdomain ratio. The processes slow themselves down and the bottleneck then is not the available processor power but the communication between processes. Executing a simulation in parallel results with a speedup s given as

$$s = \frac{t_s}{t_p}, \tag{3.26}$$

where t_s and t_p is the serial and parallel execution time, respectively. A linear (ideal) speedup, is equal to the number of used processes N_p. Usually, because of the interprocess communication, or bottlenecks which are local to a sub-domain (aforementioned local calculations), speedup will have smaller value than the number of processes $s < N$. With simulations that involve larger meshes, the difference between N_p and the evaluated

s will be larger. It may happen, however, that *super-linear* speedup is observed, where $s > N_p$: the reason behind is usually specific to the algorithm and the architecture on which the simulation is executed.

3.5 Summary

In this chapter we took the next step of the CFD workflow into the case setup and simulation running phase. This entails setting the initial and boundary field values for the fields involved in the simulation, which can be done with various utilities. Discretization and interpolation schemes are a critical component of the finite volume method in OpenFOAM and they can be selected by the user at the start, or during the simulation. In OpenFOAM there are many interpolation and discretization schemes available, so we mentioned only a few of them. A list of all available schemes is reported by OpenFOAM if a wrong entry is provided in the configuration file. We walked though the numerical foundations for some schemes, and their clean software design allows the interested reader to learn how scheme works without a deep understanding of the C++ programming language. Finally, we showed how a user can execute a flow solver in both serial and parallel modes.

Further reading

[1]   J. U. Brackbill, D. B. Kothe, and C. Zemach. "A continuum method for modeling surface tension". In: *J. Comput. Phys.* 100.2 (June 1992), pp. 335–354. ISSN: 0021-9991.

[2]   J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer, 2002.

[3]   Jasak. "Error Analysis and Estimatino for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science, 1996.

[4]   Dimitri J. Mavriplis. *Revisiting the Least-squares Procedure for Gradient Reconstruction on Unstructured Meshes*. Tech. rep. National Institute of Aerospace, Hampton, Virginia, 2003.

[5]   *OpenFOAM User Guide*. OpenCFD limited. 2016.

[6]   Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[7]   H. K. Versteeg and W. Malalasekra. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach*. Prentice Hall, 1996.

4

Post-Processing, Visualization and Data Sampling

This chapter covers the basics of post-processing, visualizaiton and data sampling. Not only OpenFOAM tools are discussed, but also paraView is used for various tasks, as well as runtime data sampling.

4.1 Post-processing

The post-processing step of a CFD analysis is the task where useful information is extraced from the computed CFD solutions. Extracted data may take the form of images, animations, plots, statistics, and the like. Thankfully, OpenFOAM is equiped with a multitude of post-processing tools that accomodate the needs of most users. The tools come in the form of what is called function objects which can be invoked either after a simulation is complete, or while a simulation is running as a sort of auxillary calculation. For example, if a vorticity field is required but is not by default calculated by the solver, a vorticity function object can be added to the simulation such that it is calculated and written alongside the expected pressure and velocity fields. Function objects and design are reviewed in depth in chapter 12. The source code of the stock post-processing tools (a.k.a function objects) is stored in various categorized subdirectories in the `$FOAM_SRC/functionObjects` directory:

```
?> ls $FOAM_SRC/functionObjects
field  graphics      lagrangian      solvers
forces initialisation randomProcesses utilities
```

While some post-processing calculations may not be directly computable with an existing utility, a combination of calculations using stock utilities may be get the job done. Similarly to other OpenFOAM utilities, post-processing applications operate on all existing time directories in a simulation case, by default. Times and time ranges can be selected explicitly by passing the -time parameter and the last time can be selected by the -latestTime parameter. Before going into detail of example workflow scenarios, some existing post-processing tools are described in the following. While it is beyond the scope of this book to describe all of them, a few selected applications are covered that should be valuable to many users.

postProcess

The first selected tool is the generalized -postProcess solver option which can be used to perform various calculations on existing flow fields. All calculated results are stored as a new field in the respective time directory. This solver option augments the behavior of the solver such that instead of performing flow calculations, it only executes the function objects included in the command line or set in controlDict. This option is attached to a solver in order to ensure that the fields and models assoicated with a particular solver will be loaded prior to post processing. The general syntax is:

```
?> solverName -postProcess -field <field name> -func <function name>
```

where solverName is the name of the solver that was used to originally used to compute the existing flow data.

In the syntax above, <function name> defines the type of calculation to perform, <field name> denotes the field to operate on, and <arguments> supply operation specific controls. Note that not all function objects require an input field name. Additional arguments, that are common among the majority of OpenFOAM utilities, such as -latestTime and -case can also be used with this utility. All existing calculations that operate on the flow field results themselves can be found here:

```
?> $FOAM_SRC/functionObjects/field
```

Outlined below are some of the more commonly used arithmetic or field calculus operations:

components separates the components of a vector or tensor field into separate volume scalar fields. For example, the velocity field U can be separated into three volScalarFields (Ux, Uy and Uz) using the following syntax:

```
?> simpleFoam -postProcess -func 'components(U)'
```

div computes the divergence of a vector or tensor field and writes the result to a new scalar or vector field respectively. A numerical scheme for the divergence operator (for example, div(U)) must be present in the fvSchemes dictionary, in order to be executed. The divergence of the velocity field U for the latest time can be computed by

```
?> simpleFoam -postProcess -func 'div(U)'
```

And the result is stored as a new volScalarField named divU, in the particular time step directory.

mag is used to compute the magnitude of a field. For scalar values, the absolute value is computed and for vectorial values, the magnitude is computed. The result is stored in a new field that consists of the old field name with a prepended mag. For example, computing the velocity magnitude of the velocity field U can be achieved by executing the following:

```
?> simpleFoam -postProcess -func 'mag(U)'
```

magSqr compute the magnitude squared of a field respectively. Write the computed result as a new scalar field that is named according the pattern 'magSqr' followed by the original field name.

```
simpleFoam -postProcess -func 'magSqr(U)'
```

yPlus

For simulations that employ turbulence modelling, the y^+ value is an important value used to verify if the near wall flow is resolved sufficiently and that resolution is in the correct range for that turbulence model. More information on what this value means and how it is calculated can be

gathered from chapter 7. The source code for the y^+ post-processing tools is located here:

```
?> ls $FOAM_SRC/functionObjects/field/yPlus
yPlus.C      yPlus.H
```

The y^+ value is calculated after the simulation is complete, using the yPlus function object. Previous versions of the code required different utilities to be called depending on whether the simulation employed LES or RANS type models but this has recently be unified into a single process for either simulation type. The y^+ values are stored in a new volScalarField called yPlus:

```
> simpleFoam -postProcess -func yPlus
```

In the above example simpleFoam was called as the base solver. In the general case whatever flow solver was used to compute the flow field can be subsituted, for example pisoFoam for transient flow. The y^+ field is calculated only on boundary faces, which are of type wall, leaving the remaining cell centered values zero. Besides the freshly written field, both yPlus tools print some valuable information to the screen. This information contains the coefficients used for the particular turbulence model as well as the minimum, maximum and average y^+ value for all wall boundary patches.

patchAverage and patchIntegrate

These post-processing tools can be used to calculate averages and integrals over a patch, respectively. The source code for both tools is located in the patch subdirectory of the postProcessing utilities.

```
?> ls $FOAM_UTILITIES/postProcessing/patch
patchAverage      patchIntegrate
```

patchAverage calculates the arithmetic mean $\bar{\phi}_f$ of a scalar ϕ, weighted by the magnitude of the surface area normal vector $|\mathbf{S}_f|$:

$$\bar{\phi}_f = \frac{\sum_f \phi_f \, |\mathbf{S}_f|}{\sum_f |\mathbf{S}_f|} \tag{4.1}$$

patchIntegrate computes the integral value for a field on a patch. For this computation, two different approaches are employed: Using the surface area normal vector $\mathbf{S}_f$ and its magnitude $|\mathbf{S}_f|$. Both results are printed to the console, resulting in one vectorial and one scalar result, respectively.

`patchAverage` can *only* handle a `volScalarField`, whereas `patch-Integrate` will also handle a `surfaceScalarField` as input. Both commands require the same set of arguments, when called from the command-line:

```
postProcess -func 'patchAverage(<fieldName>,name=<patch>)'
postProcess -func 'patchIntegrate(<fieldName>,name=<patch>'
```

In the above code snippet, `<field>` and `<patch>` represent the field and patch to operate on, respectively. The results are not stored anywhere in the case directory, but only printed to the terminal. Though they can be stored, by piping the output of the particular utility to a log file. Shown below is an example of making these calculations on the air foil example case with existing fields and piping to a stored text file.

```
?> postProcess -func 'patchAverage(<fieldName>,name=<patch>)' > patchAveResults.txt
?> postProcess -func 'patchIntegrate(<fieldName>,name=<patch>)' > patchIntegrateResult
```

vorticity

The `vorticity` utility calculates the vorticity field ω, using the velocity field $\mathbf{U}$ and writes the result to a `volVectorField` named `vorticity`. The vorticity of a velocity field represents the local magnitude and direction of rotation in the flow and is defined in equation 4.2. Performing this operation is also known as taking the *curl* of a field. The final output of this utility is the computed vorticity field written to each time directoy.

$$\omega = \left(\frac{\partial \mathbf{U}_z}{\partial y} - \frac{\partial \mathbf{U}_y}{\partial z}, \frac{\partial \mathbf{U}_x}{\partial z} - \frac{\partial \mathbf{U}_z}{\partial x}, \frac{\partial \mathbf{U}_y}{\partial x} - \frac{\partial \mathbf{U}_x}{\partial y} \right) \tag{4.2}$$

The source code for this post-processing utility can be found in `$FOAM_UTIL-ITIES/postProcessing/velocityField/vorticity` with an execution example shown below.

```
?> postProcess -func vorticity
```

probeLocations

If field data needs to be probed at certain locations during post-processing, `probeLocations` is the tool of choice. This tool requires an input dictionary, either set in `system/probesDict` or as a function object call in `system/controlDict`. It operates around two lists, one containing the names of the fields to probe, and one for the locations in space to probe the field values from. In general if only the solution needs to be probed after calculation, using `probesDict` is appropriate. If transient prode data is required to be written during simulation runtime such as in the instance of transient simulations, add the probe procedure as a function object in the `controlDict`. To sample the pressure field p and velocity field U at two points $[0, 0, 0]$ and $[1, 1, 1]$, the `probesDict` is configured as shown below:

```
fields
(
    p
    U
);

probeLocations
(
    (0 0 0)
    (1 1 1)
);
```

If not stated otherwise, the fields are probed at all existing times with the output files located in a nested subdirectory inside the case directory. The first folder is named `probes` and the subfolder indicates the first time step the data was sampled from. For example, all pressure data gathered by the probing can be found in the file `probes/0/p`. The data is arranged in a tabulated manner with the time being stored in the first column, followed by the extracted field value. This format can then be processed using plotting utilities such as `gnuplot` or `python/matplotlib`. An example of the results of a probing of the pressure field is shown below.

```
#   x      0      1
#   y      0      1
#   z      0      1
#   Time
    0      0      0
    10     3.2323  2.2242
```
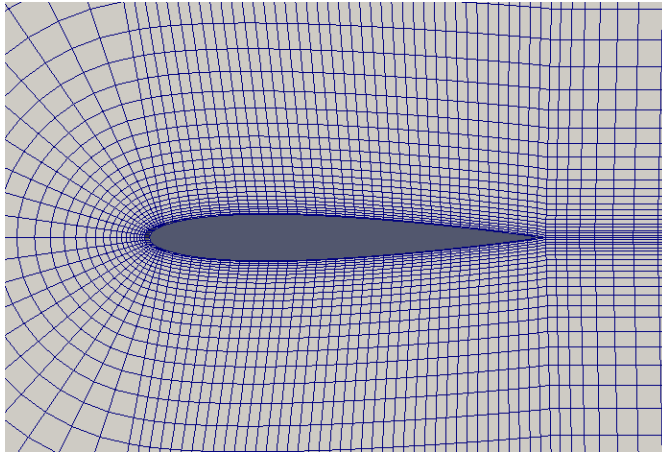
Figure 4.1: Sideview of the NACA0012 profile

Example case studies

As an example to show possible applications for the post-processing tools, a two-dimensional NACA0012 hydrofoil is selected. The hydrofoil is deeply submerged and placed in a domain with a homogeneous inflow. The resulting Reynolds-number is $Re = 10^6$. With the chord length of the NACA profile being $c = 1m$ and a kinematic viscosity of $\nu = 10^-6\frac{m^2}{s}$ this gives a freestream velocity of $v = 1m/s$.

The case can be found in the repository under `chapter4/naca` and should be copied into the user directory:

```
?> cp -r chapter4/naca $FOAM_RUN
?> cd $FOAM_RUN/naca
```

To generate the results, the `Allrun` script provided with the tutorial must be executed. After the simulation is finished, the distribution of y^+ over the surface of the NACA profile is inspected. In order to do so, the yPlus post processing function object has to be executed in the case directory and the resulting yPlus field must be visualized on the surface using paraView.

Although yPlus prints the minimum, maximum and average y^+ values to the screen, the average value can be recomputed differently, based upon the yPlus field. Using `patchAverage` does exactly this, after the initial yPlus field has been computed:

```
?> ./Allrun
?> simpleFoam -postProcess -func yPlus
?> postProcess -func 'patchAverage(yPlus,name=FOIL)'
```

The output is not immediately plot-friendly, as there is some output overhead and superfluous text elements. By using `grep` to look only for floating point numbers at the end of a line, the averages can be filtered out of the `patchAverage` function output. For later processing, they can be piped into a text file `yPlusAverage`. This can be performed by running the command shown below:

```
?> postProcess -func 'patchAverage(yPlus,name=FOIL)' | \
   grep -o -E '[0-9]*\.[0-9]+$' > yPlusAverage.dat
```

Another practical post-processing task is to calculate the integral and average wall-shear stress in the x direction. The shear stress itself can be computed by `wallShearStress`, which writes a new `volVectorField` to each time step directory. By simply chaining the respective commands, the average `wallShearStress` can be calculated, without the need to use paraView:

```
?> simpleFoam -postProcess -func wallShearStress
?> postProcess -func 'component(wallShearStress)'
?> postProcess -func 'patchAverage(wallShearStressx,name=FOIL)' | \
   grep -o -E '(-|+)?[0-9]*\.[0-9]+$' > stressXAverage.dat
```

4.2 Data sampling

The post-processing methodology `sample` provides an easy to use and powerful way to extract simulation data. While there are many visualization applications that produce attractive imagery, `sample` is better suited for the less visually appealing yet arguably more important role of quantitative analysis. For example, instead of visually estimating the boundary layer thickness from a velocity magnitude representation, it can be extracted from the raw or interpolated velocity data and into a data table.

In general, `sampling` is used to extract and produce 1D, 2D, or 3D representations of data subsets of the simulation solution such as point values, plots over lines, or iso-surfaces respectively. Different output formats are supported, as well as different geometrical sampling entities.

In previous versions of OpenFOAM, `sample` was an independent utility but has since been depreciated and is now distributed between various post-processing function objects. This more recent configuration allows all sampling operations to be performed during simulation run-time as well as after the simulation has completed. These functions are now typically defined via `controlDict` in the `functions` subdictionary by default. Alternatively the functions can be defined in a separate dictionary so long as the user points to the appropriate file when calling the post processing utility. This latter approach tends to help the case and post processing procedures stay organized and will be outlined below. Examples of various point probe, graphing, or surface extraction function configurations are outlined in example case dictionaries here:

```
?> ls $FOAM_ETC/caseDicts/postProcessing/probes
boundaryCloud      cloud.cfg      internalCloud.cfg probes.cfg
boundaryCloud.cfg internalCloud probes
?> ls $FOAM_ETC/caseDicts/postProcessing/graphs
graph.cfg sampleDict.cfg singleGraph
?> ls $FOAM_ETC/caseDicts/postProcessing/visulaization
runTimePostPro.cfg streamlines streamlines.cfg surfaces surfaces.cfg
```

Opening the examples shown above from `etc` in a text editor shows all available configuration options for the `sample` post-processing utility. There is a large number of options available, and the provided `sampleDict` is very well documented. The options are described in a clear fashion. `sample` can handle a multitude of sampling parameters: field names, output formats, mesh sets, interpolation schemes, and surfaces.

Regardless of the variety of sampling parameters, `sample` always handles the data sampling process in the same way, regardless of the user's choice of a parameter sub-set. A field to be sampled is chosen by providing the field name within the `fields` word list. A sub-set of the mesh (`sets` sub-dictionary) or a geometrical entity (`surfaces` sub-dictionary) is used to locate the data sampling points. In case when the data sampling points do not coincide with the mesh points that hold the field data (e.g. cell centers or face centers), the data is *interpolated* using different interpolation schemes (`interpolationScheme` parameter). The interpolated data is then stored in the case, in a specified output format (`setFormat` parameter).

An example of a 1D data extraction is to define a line which intersects the flow domain and sample the velocity field along this line. This can be used to sample e.g. the velocity profile, as usually done for the `cavity`

case. This extracted profile can then be compared to other datasets, using any preferred plotting utility.

Another example application of `sample` is to extract boundary field values on large simulation cases. Instead of trying to open up the entire simulation case in paraView, `sample` can be used to extract only the values on the boundary patch in question. This localized approach to post-processing using `sample` can drastically reduce the required computational resources, depending on the size of the datasets.

Any simulation case can be used to show how to sample the simulation data using the `sample` utility. For that purpose, the two-dimensional rising bubble test case is selected, available in the `chapter4/risingBubble2D` sub-directory of the example case repository. In order to use `sample` sucessfully on that case, the simulation has to be executed:

```
?> blockMesh
?> setFields
?> interFoam
```

Sections that follow cover examples of using `sample` and they all involve manipulating a `functions` subdictionary. While this subdictionary can be defined in the `controlDict` it can also be defined in a separate, custom dictionary file which is the approach we will use in this example.

WARNING

Before proceeding with the examples, open the `sampleDict` provided with the `sample` source code in a text editor of your choice, in order to see all the possible sampling configuration options.

4.2.1 Sampling along a line

In this examplei (rising bubble) the width of the 2D bubble is examined after it has settled on the top wall of the simulation domain. The sampling line will sample the `alpha.liquid` field at time $t = 7.0s$ along a line which crosses the bubble. A `functions` subdictionary configured to sample along a line is shown below:

```
setFormat raw;
interpolationScheme cellPoint;
fields
(
    alpha.liquid
```

```
    p_rgh
    U
);
sets
(
    alphaWaterLine
    {
        type       uniform;
        axis       distance;
        start      (-0.001 1.88 0.005);
        end        (2.001 1.88 0.005);
        nPoints    250;
    }
);
```

The `setFormat` option changes the format of the data written to file and the `interpolationScheme` option dictates what type (if any) of value interpolation occurs, before data is mapped to the sample line. All fields to be sampled need to be listed in the `fields` list - `alpha.water` field in this case. The `sets` subdictionary contains a listing of all of the sample lines that are extracted.

The `type` entry of the `alphaWaterLine` sub-dictionary defines how the sampled data is distributed along the line - in this case, `uniform` point distribution is used. The `axis` parameter determines how to write the point coordinates - `distance` results in a parametric output, as the coordinate is a distance along the line, starting at the first line point. There are other options available for the `axis` parameter, such as `xyz` where the absolute position vector of the sample point will be written as the first column of the data file. Next, for a line sample, three-dimensional `start` and `end` points need to be provided. In this setting, `nPoints` determines the number of sample points.

> WARNING
>
> Note that the sample line position is slightly adjusted to be away from the boundary. In general, the sample line is not to be co-planar with mesh faces, as it prevents `sample` from determining which cells the line is intersecting.

Once the `sampleDict.set` has been configured, `postProcess` can be executed for $t = 7$ s:

```
?> interFoam -postProcess -dict ./system/sampleDict.set -time 0.7
```

A new directory is created, holding the following files:
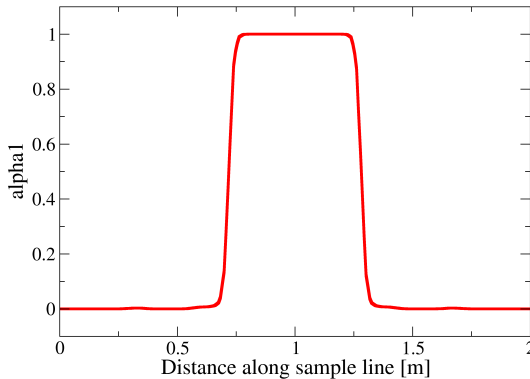
Figure 4.2: A sample of the `alpha.water` field along our defined line

```
?> ls postProcessing/sets/7
alphaWaterLine_alphaWater_p.xy  alphaWaterLine_U.xy
```

The scalar fields `alpha.water` and `p` are stored in the same file, while the sampled vector velocity field is stored in a separate file. `alpha.water` field values can be visualized from the stored data, using a plotting tool and should lead to a figure similar to figure 4.2.

> **TIP**
>
> If the `-time 7` option is omitted when executing `sample`, every timestep is sampled resulting in the correspoding XY data being interpolated and saved.

4.2.2 Sampling on a plane

Sampling along a plane is especially useful for large 3D cases which are big enough to require a long time to transfer the simulation data across a network connection. The process for setting up the `sampleDict` for a plane is very similar to setting up line sampling. The `surfaceFormat` entry needs to be configured and a list of sample planes needs to be provided inside the `surfaces` sub-dictionary. Similar to line sampling, having the sample plane co-planar with mesh faces should be avoided.

The `interpolationScheme` set previously is again used to interpolate the cell centered flow data onto the plane. An example `sampleDict` for this kind of setup looks like this:

```
functions
{
    // Surfaces sample function object
    surfaces
    {
        type surfaces;
        surfaceFormat vtk;
        writeControl writeTime;
        interpolationScheme cellPoint;
        fields (U p_rgh alpha.water);

        surfaces
        (
            constantPlane
            (
                type cuttingPlane;
                planeType pointAndNormal;

                pointAndNormalDict
                {
                    point      (1.0 1.0 0.005);
                    normal     (0.0 0.0 1.0);
                }
            )
        );
    };
}
```

The VTK output format is chosen for the surface, and the surface `type` is set to `plane` in the `constantPlane` sub-directory. The plane is defined using a point position vector (`point`) and a plane normal vector (`normal`).

In order to generate the Visualization Toolkit (VTK) planar surface and extract the `alpha.water` data, `-postProcess` is executed on a specific explicit time directory of the `chapter4/risingBubble2D` case:

```
?> interFoam -postProcess -dict ./system/sample.surface -time 7
```

A `surfaces/7` sub-directory is created in the `postProcessing` folder and it contains the sampled data:

```
?> ls postProcessing/surfaces/7
  constantPlane.vtp
```

Obviously, as many VTK surfaces are stored as fields are sampled. A surface stored in a VTK format can be visualized by opening it directly in paraView, through the `Open` dialogue. In figure 4.3, the sampled plane

surface is shown with a sampled `alpha.water` field. Since the `rising-Bubble2D` case is two-dimensional, the result of this sampling example is the same as when the `cut` filter is used in paraView.
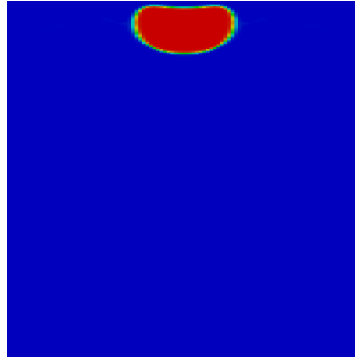
Figure 4.3: A sample of the `alpha.water` field on a plane

4.2.3 Iso-surface generation and interpolation

In addition to extracting data, the `-postProcess` utility can produce iso-surfaces from existing flow data. In this example it is used to generate an iso-surface representing the gas-liquid interface and interpolate the pressure field onto it. The `isoSurface` type is configured in a similar `surfaces` sub-dictionary of `sampleDict`:

```
functions
{
    // Surfaces sample function object
    surfaces
    {
        type surfaces;
        surfaceFormat vtk;
        writeControl writeTime;
        interpolationScheme cellPoint;
        fields (U p_rgh alpha.water);

        surfaces
        (
            fluidInterface
            (
                type        isoSurface;
                isoField    alpha.water;
                isoValue    0.5;
                interpolate true;

                pointAndNormalDict
```
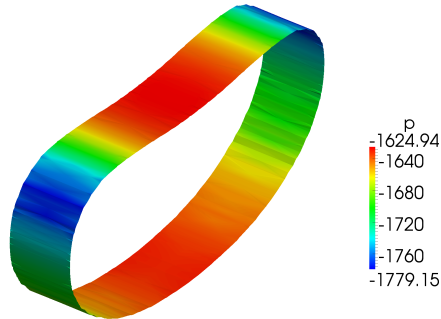
Figure 4.4: An illustration of an iso-surface of `alpha.water` $= 0.5$ colored by pressure p

```
{
    point    (1.0 1.0 0.005);
    normal   (0.0 0.0 1.0);
}
    )
);
};
}
```

In order to generate the iso-surface, `-postProcess` is to be executed on the $t = 7\,\mathrm{s}$ solution output:

```
?> interFoam -postProcess -dict ./system/sample.isoSurface -time 7
```

In figure 4.4, the `alpha.water` $= 0.5$ iso-surface is shown with the interpoalated pressure acting on the bubble.

4.2.4 Boundary patch sampling

The capabilities of `-postProcess` are not limited to creating sample planes and sample lines. An entire patch can be extracted, with any desired boundary flow field values mapped to it. Even though a rather small two-dimensional simulation case is used to illustrate this process, patch sampling is better suited for large cases, where loading the entire flow domain is not efficient or perhaps too large to open in a post processor due to RAM limitations. In this example, the top wall patch `top` of the `risingBubble2D` example case is extracted and the pressure field acting upon it is visualized. Using the appropriately configured `sampleDict`:

```
functions
{
    // Surfaces sample function object
    surfaces
    {
        type surfaces;
        surfaceFormat vtk;
        writeControl writeTime;
        interpolationScheme cellPoint;
        fields (U p_rgh alpha.water);

        surfaces
        (
            walls_constant
            (
                type      patch;
                patches   ( top );
            )
        );
    };
}
```

the sampling is performed on the final time step of the `risingBubble2D`
example simulation case:

```
?> interFoam -postProcess -dict ./system/sample.patch -time 7
```

The `sample` utility generates the `postProcessing/surfaces/7` sub-
directory in the simulation case directory, with the following contents:

```
?> ls postProcessing/surfaces/7/
walls_constant.vtp
```

One patch VTK file is created with the set of three fields written at the
boundary elements.

4.2.5 Sampling multiple sets and surfaces

The `sample` utility allows sampling of different sets and surfaces, thus
`sampleDict` stores lists of such elements. A working `sampleDict` con-
figuration file is prepared in the `risingBubble2D/system` simulation
case directory. It stores the configuration for all the examples described
in this section. When additional functionalites are required from `sample`,
the `sampleDict` dictionary provided with the application source code
should be the first place to look.

4.3 Visualization

To visualize the results of the simulation, the visualization application paraView will be used. paraView is an advanced open-source application used for visualizing field data, and there is a lot of information available on its use at `paraview.org/Wiki/ParaView`. In order to avoid repetition, only the minimum amount of information required to visualize the results of the OpenFOAM simulations in paraView is provided in this section.

paraView allows the user to execute a series of filters on the data and visualize the respective output. The filters support different operations, such as: calculation of streamlines, visualizing vector fields as glyphs, calculating iso-contour surfaces, just to name a few. Some of the presented OpenFOAM post-processing tools can be replicated within paraView as well, such as sampling data from points and along lines.

paraView has a native reader for OpenFOAM data and can then be used directly to display OpenFOAM case data. There is no need to compile `ParaFoam` and using paraView also provides the opportunity to utilize the latest verion of the upstream software. The detour using `ParaFoam` is still working, but not required anymore. In order to make use of the native paraView reader, a file that ends with `.foam` needs to be present in the case directory. Usually, this file is named in the same way as the simulation case directory, but the choice is of no importance.

In the following, the `risingBubble2D` example case for `interFoam` is selected, to introduce some of the most basic working principles of paraView. This case can be found in the example repository under `chapter4/risingBubble2D`. First, the simulation must be executed in order to generate the data which can be visualized:

```
?> blockMesh
?> setFields
?> interFoam # wait a little bit
?> touch risingBubble2D.foam
```

The last line is important in the context of this section. It generates an empty file with the "Foam" suffix, required by paraView to open the OpenFOAM case data. Finally, paraView is used to open the case files and it is started as a background process, leaving the current terminal accessible for further commands.
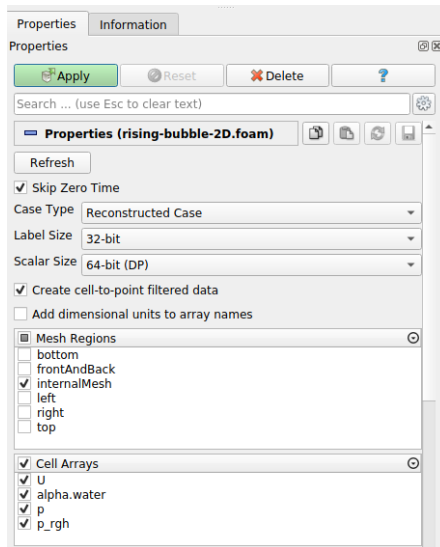
Figure 4.5: paraView `Properties` window

```
?> paraview risingBubble2D.foam &
```

Note that if the paraview binary is not on your computer's `PATH` then you may have to call it with the full directory path. An example is shown below but will need to be adjusted to represent your paraview installation location.

```
?> /path/to/install/ParaView-5.7.0/bin/paraview risingBubble2D.foam
```

Once paraView is started, the `risingBubble2D.foam` file should appear, as shown in figure 4.5. Otherwise the OpenFOAM case can be opened by pointing the paraView file browser to the `risingBubble2D.foam` file, via the *Open* dialogue. On the left side of the paraView Graphical User Interface (GUI), options for choosing parts of the mesh and different fields to be read are displayed to the user. This window is named `Properties`, and if it is closed, it can be reactivated using the `View` drop-down menu. The `Properties` view is shown in figure 4.5.

By default, the internal mesh and all the cell centered fields of an Open-FOAM case are chosen for visualization. The fields and the mesh are read when the selection is accepted by clicking on the `Apply` button in the `Properties` window. The choice of the visualized field can be
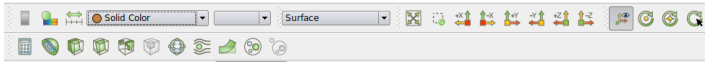
Figure 4.6: paraView field and display form selection
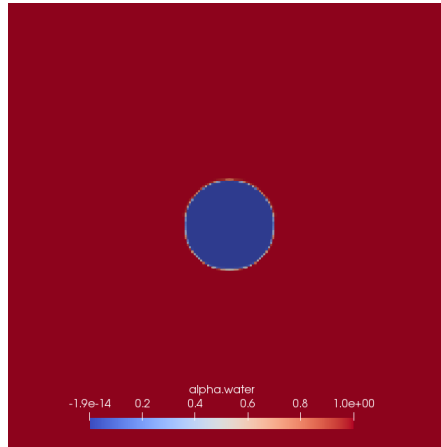
Figure 4.7: Visualized `alpha.water` field

made using the panel above the `Properties` window and the `Pipeline browser`, shown in figure 4.6.

Initially, the mesh is colored with a solid color. Clicking on the `Solid color` tab in the panel shown in figure 4.6, produces a drop-down menu with all available fields. The `alpha.water` field in the initial time step is shown in figure 4.7.

Manipulation of fields using filters is a very straightforward process in paraView. Most often used filters are shown as buttons at the bottom part of the main panel shown in figure 4.6. Other filters are available in the `Filters` roll-down menu on the main panel. Documentation covering the use of different filters is extensive, including the paraView community wiki page (`paraview.org/Wiki/ParaView`), and the official documentation.

One often used Filter is the `Contour` filter for visualizing the interface between two fluids. The `Contour` filter computes an iso-surface of a prescribed value, based on a scalar field. In this case the scalar field is `alpha.water`. This operation is in fact fairly similar to the `isoSurface` sampling utility, described in section 4.2. The filter can be selected from
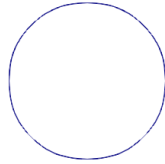
Figure 4.8: Visualized 0.5 iso-contour of the `alpha.water` field
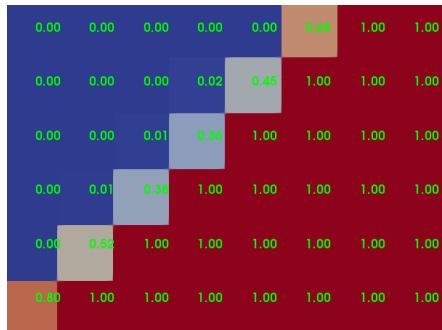
Figure 4.9: Visualized values of the `alpha.water` field
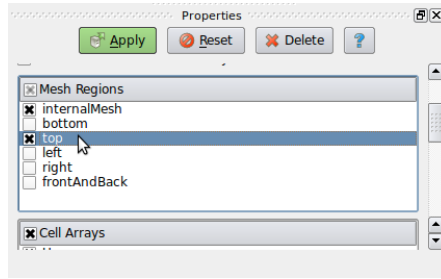
Figure 4.10: Reading the `top` boundary mesh patch.

the main menu `Filters|Common|Contour`. The scalar field `alpha.wa`‐`ter` must be selected for the `Contour by` parameter and 0.5 for the contour value. When all settings are defined as desired, a click on `Apply` shows the result in the right 3D view. Finally the interface line in the XY plane should be visible, similar to figure 4.8.

An interesting aspect of the visualization in paraView is the ability to enu‐merate cell and/or point centered values with field data using the `Selec`‐`tion Inspector`. To start, make sure that the "rising-bubble-2D.foam" is selected in the `Pipeline Browser`. Click on `View->Selection In`‐`spector` and within the `Selection Inspector` window, and check the `Invert Selection` checkbox. You can decrease the opacity of the selec‐tion to 0 in the `Display Style` sub-window, to prevent it from obscuring the value numbers shown in the cells. To display the values of the `al`‐`pha.water` field in each cell, select the `Cell Label` tab of the `Display` `Style` sub-window and choose `alpha.water` from the `Label Mode` drop‐down menu. In the `Cell Label` the format of the shown values can be specified, e.g. "%.1f" would be a floating point format with a single decimal digit. Figure 4.9 shows a zoomed-in detail of the `alpha.water` field with the displayed cell-centred values.

Extracting values from the `top` patch can be done using paraView as opposed to the `sample` utility which was discussed in Section 4.2. In order to sample the patch, the boundary mesh and not just the internal mesh needs to be read in paraView. To do so, select the `top` patch in the `Mesh Regions` part of the `Properties` window as shown in Figure 4.10. The `top` patch is then isolated in the pipeline from the rest of the case mesh by selecting it within the `Patches` branch using the `Filter->Alphabetical->Extract Block` filter. To plot the data of
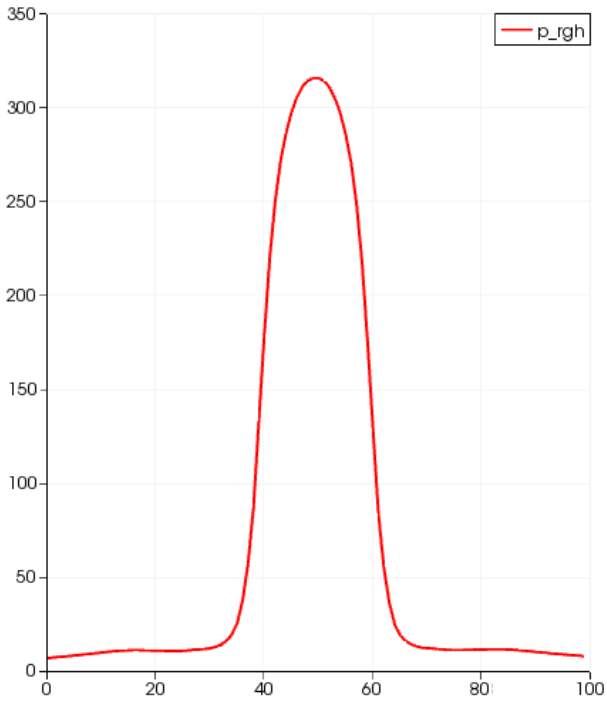
Figure 4.11: Dynamic pressure at the `top` boundary mesh patch at 7s simulation time.

this patch directly without interpolation, select the extracted block in the pipeline browser and choose `Filters->Data Analysis->Plot Data`. In the display window (`View->Display`), select the dynamic pressure p_{rgh} and skip to the last time step of the simulation. The resulting diagram is shown in Figure 4.11.

In this section, the absolute minimal aspects of the workflow with the paraView visualization application are provided. For further details, please consult the material available on the Internet, as well as the official documentation.

Part II

Programming with OpenFOAM

5

OpenFOAM design overview

The FOAM part of the OpenFOAM name is given as an acronym for Field Operation and Manipulation. As noted in the previous chapters, FOAM can be quite complex, because they represent and manipulate physical properties as tensor fields in geometrically complex solution domains using unstructured finite volume discretization . Explicit and implicit operations on discrete tensor fields can be performed. Field changes are governed by a conservation law, modeled as a Partial Differential Equation (PDE), and the numerical solution of the PDE usually involves assembling and solving large sparse linear equation systems. Sometimes the PDE are strongly coupled and the coupling needs to be taken into account by a numerical algorithm, for example the coupled solution of the momentum and pressure equations. Solving the linear algebraic equation systems for new field values is performed using iterative solution algorithms because of usually large matrix sizes[1, 3].

Translating all the aforementioned aspects of the FVM into a software framework is only possible if the chosen programming language supports abstraction of complex concepts and computationally efficient and portable implementation. In the context of OpenFOAM these concepts include: fields, meshes, discretization schemes, interpolation schemes, equations of state, spray particle systems, matrices, matrix storage formats, configuration files, parallel programming aspects, etc. Even this short list shows how elements occupy different *levels of abstraction*. For example,

the field and the mesh concepts are on a higher level of abstraction, than a dictionary data structure. Abstraction allows the programmer to effectively build software elements which model the behavior of complex concepts in the area of interest by allowing him/her to concentrate not exclusively, but more strongly, on the currently implemented concept. This way, the programmer does not have to hold an image of an entire software system in his mind, but switch concentration from one relatively isolated concept to another. Well-modeled concepts are those that are strongly cohesive and loosely coupled with one another. This enables simplifies extensions and makes the software more modular. For example, an important concept in CFD is a finite volume mesh that is used to discretize the flow domain.

The mesh will hold various geometrical and topological data, as already described in sections 1.3 and 2.1. Additionally, different (often complex) functions are needed to operate on that data. As an example of abstraction in OpenFOAM and the C++ programming language, both the data and the related functions of the finite volume mesh are *encapsulated* into a *class* `fvMesh`. This allows the programmer to *think in terms of a mesh*, and not bother with all the details involving the data structures and functions that build it. Such high-level of abstraction in thinking allows the programmer to write algorithms that use the entire mesh as one of their arguments, which makes the algorithm interface much easier to understand. Otherwise, algorithms working with specific sub-elements of the mesh would have dozens of arguments, as is often the case in procedural programming. Without abstraction, a very large number of global variables would also be present and operated on by various algorithms (routines) which makes it very difficult to determine the program flow.

INFO

The first sections of this chapter cover an overview of the software design of OpenFOAM. However, terms from software development are not covered and should be learned independently. OpenFOAM is a large software and learning software development is a requirement for learning how to develop new methods in OpenFOAM.

Programming in a higher level of abstraction is supported by the C++ programming language. It is a multi-paradigm language that supports *procedural*, *object oriented*, *generic* and the *functional* programming paradigm. Implementing higher abstraction layers makes the implementation more

readable, yet the C++ language still maintains very high computational efficiency and is cross-platform. These aspects make the C++ language a common choice for scientific and computational software development.

The best way to understand how different parts of OpenFOAM are designed and interact with each other is through browsing the source code. In order make this easier for the user, both the official and extend OpenFOAM release provide support for generating HyperText Markup Language (HTML) documentation. Generating HTML documentation is performed by via the `Doxygen` documentation system. The Doxygen documentation generated for the main development version is also available online as the Extended Code Guide.

5.1 Generating doxygen documentation

The generated Doxygen documentation allows the reader to quickly find information on the class in question. The Doxygen documentation system generates HTML documentation which can be browsed using a web browser, with the advantage of viewing class and collaboration diagrams with linked elements. This makes it especially helpful when interactions between classes and algorithms are investigated. Following links in a HTML browser to a base class may be more effective for most readers compared to browsing through source code using a text editor. Alternatively, an Integrated Development Environment (IDE) might be used when working with OpenFOAM with support for browsing through the source code. To start learning OpenFOAM, we recommend a text editor and the Doxygen generated documentation. Setting up an IDE to work with OpenFOAM might be a complex task in itself for a novice OpenFOAM user.

In order to generate the documentation, `Allwmake` needs to be executed in the

```
$WM_PROJECT_DIR/doc
```

folder. Once Doxygen has finished generating the documentation, the

```
$WM_PROJECT_DIR/doc/Doxygen/html/index.html
```

file can be viewed with a web browser. This page is the starting page of the local generated HTML help of the OpenFOAM installation. Alternatively, a LaTeX based documentation, is available. This documentation is available in PDF format but must be converted manually.

EXERCISE

Using the Doxygen source code documentation find what is a requirement for a `Type` managed by the `tmp<Type>` smart pointer.

5.2 Parts of OpenFOAM encountered in simulations

This section will address some parts of OpenFOAM that are encountered by the user who is running simulations.

Setting the initial and boundary conditions in chapter 3 has already shown that the user has considerable flexibility at his or her disposal. Without compiling any additional code existing boundary conditions, interpolation and discretization schemes, viscosity models, equations of state, and similar parameters can be selected in input (field or configuration) files.

In terms of the underlying implementation, choosing a boundary condition based on input data from files is relatively complex: an object of a specific class is chosen and instantiated at *runtime* based on a read parameter (boundary condition type) defined by a user. Running simulations puts the user directly or indirectly into contact with the following parts of OpenFOAM: the executable applications (solvers, pre-processing utilities, post-processing utilities), the configuration system (dictionary files), the boundary conditions and the numerical operations (choosing discretization schemes).

5.2.1 Applications

The executable applications are programs that are run by the user in the command line or via a GUI. They belong to what is usually called *client code*, that uses (is a client) of various OpenFOAM libraries. Executable

```
applications

        ├───────────→ solvers
        ├───────────→ test
        └───────────→ utilities
```
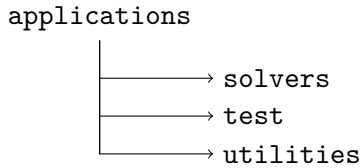
Figure 5.1: Application categories in OpenFOAM.

applications (short: applications) are organized in a directory structure as shown in figure 5.1. The applications folder can be easily accessed by executing the alias app in the command line, or by switching to the applications directory:

```
?> cd $FOAM_APP
```

The modular design and high level of abstraction in OpenFOAM allows the user to easily build mathematical models. Different solution algorithms for systems of coupled partial differential equations can also be implemented in a straightforward way using OpenFOAM's high level DSL. As a result, a very wide choice of solver applications has become available over time. The solver applications are categorized in groups, as shown in figure 5.2.

Different testing applications can be found in the test sub-directory. For example, the dictionary test application tests the main features of the dictionary class, while the parallel and parallel-nonBlocking application implement the code used for abstracting parallel communication in OpenFOAM. Viewing test application source code can be very beneficial in cases when examining the library source code is not enough to understand how the class/algorithm works.

5.2.2 Configuration system

The configuration system is composed of configuration (dictionary) files: text files stored in a format similar to JSON. The dictionary file consists of categorized input data which is used for constructing an associative data structure called *dictionary*. A dictionary is in essence a hash-table. This data structure is used often in OpenFOAM to relate (map) *keys* to *values*.

```
applications
            ├──────────→ solvers
            │           ├──────────→ basic
            │           ├──────────→ combustion
            │           ├──────────→ compressible
            │           ├──────────→ dicreteMethods
            │           ├──────────→ DNS
            │           ├──────────→ electromagnetics
            │           ├──────────→ financial
            │           ├──────────→ heatTransfer
            │           ├──────────→ incompressible
            │           ├──────────→ lagrangian
            │           ├──────────→ multiphase
            │           └──────────→ stressAnalysis
            ├──────────→ test
            │           ├──────────→ ...
            │           ├──────────→ dictionary
            │           ├──────────→ ...
            │           ├──────────→ parallel
            │           ├──────────→ parallel-nonBlocking
            │           └──────────→ ...
            ├──────────→ utilities
            │           ├──────────→ finiteArea
            │           ├──────────→ mesh
            │           ├──────────→ miscellaneous
            │           ├──────────→ parallelProcessing
            │           ├──────────→ postProcessing
            │           ├──────────→ preProcessing
            │           ├──────────→ surface
            │           └──────────→ thermoPhysical
```
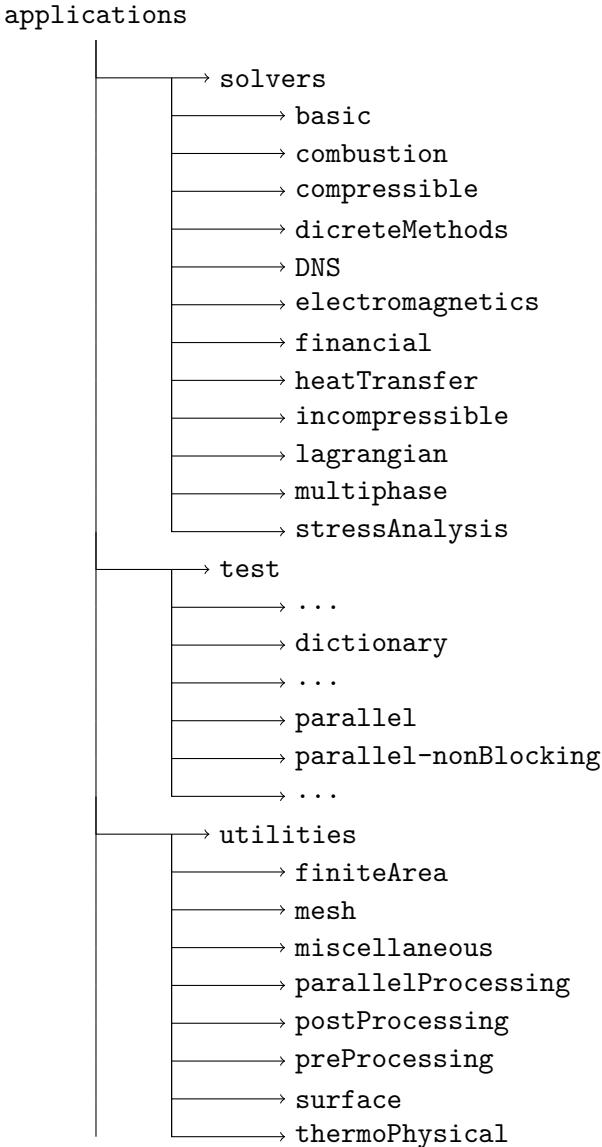
Figure 5.2: Application sub-directories.

INFO

The test applications are a very useful resource of information regarding specific classes and algorithms. They show how the classes and algorithms are meant to be used.

The various dictionary files found in the simulation case directory contain different configuration parameters: boundary conditions, interpolation schemes, gradient schemes, numerical solvers, etc. Choosing different elements as well as initializing them properly is performed at runtime after starting an executable application. The process of choosing types at runtime is called Runtime Selection (RTS) and it employs quite a few software design patterns and C++ language idioms. The RTS process is quite complex, and its description is outside the scope of this chapter. At this point it is only important to remember that it makes OpenFOAM very flexible easy to use.

INFO

The configuration system itself does not exist in a form of an abstraction, implemented as a single class. The system is supported by the functionallity of the `dictionary` and `IOdictionary` classes, Input/Output (IO) file streams, and the RTS mechanism used in the runtime selectable classes.

For example, any user defined class in OpenFOAM can be made *runtime selectable*. Additionally, the class attributes can be modified during the simulation - as soon as the dictionary file is modified, the file can be re-read and the class attributes are set to the new values.

INFO

The parameters defined in dictionaries are not changed in the executable application immediately when they are modified. The change is applied

5.2.3 Boundary conditions

Boundary conditions are applied on discrete fields, as described in section 1.3, and are implemented as a separate class hierarchy. Because they are implemented as classes, the RTS mechanism allows the user to configure the boundary condition types and parameters as dictionary entries at runtime. Boundary conditions are covered in the chapter 10 so they are not fully addressed in this chapter.

5.2.4 Numerical operations

The numerical operations are used by the solver and are responsible for equation discretization component of the simulation. The user will be in contact with them when he/she modifies the `system/fvSchemes` dictionary file in the simulation case directory in order to choose a different kind of discretization practice, interpolation scheme, or similar parameter. Different numerical operations have different properties and choosing them requires experience in CFD and also, ideally, in numerical mathematics.

The part of OpenFOAM responsible for numerical operations of the FVM can be found executing the alias `foamfv` in the console, or switching to the appropriate directory:

```
?> cd $FOAM_SRC/finiteVolume
```

Numerical operations are comprised of *interpolation schemes* which are then used by the *discrete differential operators* to discretize the model equations.

The discrete differential operators used often in CFD are: divergence ($\nabla\cdot$), gradient(∇), laplacian ($\nabla\cdot\nabla$) and curl ($\nabla\times$). They are either explicitly or implicitly evaluated. Explicit operators result in new fields as results. Implicit operators are used to assemble *coefficient matrices* - they discretize the equation terms of the mathematical model. The discrete differential operators are, on the other hand, implemented as function templates, parametrized by the geometric tensor field parameters. This generic implementation of the differential operators makes it straightforward to write mathematical models under the same function names which are used to differentiate tensor fields of various ranks. Since the operators are implemented as functions, assembling a different mathematical

model is trivial: a different sequence of function calls results in a different model. OpenFOAM has both implicit and explicit discrete operators that are named the same: e.g. the explicit divergence and the implicit divergence. To avoid ambiguity in the name lookup process, the operators are categorized under two *c++ namespaces* and accessed using fully qualified names (`fvc::` as explicit, `fvm::` as implicit).

The discrete operators are generic algorithms templated for a tensor parameter, using a class trait system to determine the resulting tensor rank. Standard choice of discretization in OpenFOAM is based on the divergence theorem, and the operator calculation is delegated to the discretization scheme. Those interested in developing their own discretization schemes may want to examine the files `fvmDiv.C` and `convectionScheme.C`. Both files contain implementations that show how the divergence term approaches the equation discretization. The source code listing below holds the implementation of the code that delegates the divergence calculation from the operator to the convection scheme.

```
return fv::convectionScheme<Type>::New
(
    vf.mesh(),
    flux,
    vf.mesh().divScheme(name)
)().fvmDiv(flux, vf);
```

The schemes can be selected via RTS, but they must comply to the interface prescribed by `convectionScheme.H`. Such flexible design can be summarized with the following way:

- the operators delegate the discretization to the schemes,
- schemes compose a runtime selectable hierarchy,
- to write a new convection scheme one must only inherit from `convectionScheme` and add RTS,
- once the convection scheme is implemented *none of the solver application code needs to be modified*.

Implementing the operators as function templates and binding them to schemes has many advantages and it prepares the framework for easy extensions without introducing modifications to the existing code. Describing in full detail how the discretization mechanism is implemented lies outside of scope for this book.

The source code which implements other discrete operators can be found in the directory `$FOAM_SRC/finiteVolume/finiteVolume` in the sub-directories `fvc` and `fvm`. The directory `fvc` stores implementations of *explicit discrete operators*, which *result in computed fields*. The directory `fvm` stores the implementation of *implicit discrete operators*, which result in *coefficient matrix assembly* of the algebraic system of equations.

> **INFO**
>
> Whenever there is an `fvc::` (finite volume calculus) operator in the code, the result will be a field. When the `fvm::` operator is encountered, the result will be a coefficient matrix.

The actions are performed by the solver application, the interaction with the solver code is done when the user modifies the solver in order to make it function in a different way. OpenFOAM supports runtime solver modifications via fvOptions and function objects (see chapter 12), without requiring the user to modify the solver code. More information on solver applications and how to program new solvers is covered in chapter 9. Generally, when standard solvers are used to obtain simulation results, their code will typically not be modified by the user.

A high level of abstraction in OpenFOAM allows the user to write new solvers and solution algorithms very quickly. The high abstraction level of OpenFOAM can be nearly be used as a CFD programming language (DSL) - also reffered to as *equation mimicking* ([2]). Take, for example, a mathematical model for a scalar transport of a scalar property T:

$$\frac{\partial T}{\partial t} + \nabla \cdot (T\mathbf{U}) + \nabla \cdot (k\nabla T) = S \tag{5.1}$$

Equation (5.1) describes the transport of a field T composed of a passive advection with the velocity $\mathbf{U}$, diffusion with the diffusion coefficient field k together with a source term S. Discretization operators, being function templates, can operate on fields of different tensors, and thus on a scalar field they produce the following model equation in OpenFOAM:

```
ddt(phi) + fvm::div(phi, T) + fvm::laplacian(k, T) = fvc::Sp(T)
```

The code describing the model is consisted of discrete operators `ddt`, `div`, `laplacian` and `Sp`.

Interpolation schemes also belong to the numerical operations in Open-FOAM. The most often used interpolation schemes are built upon owner-
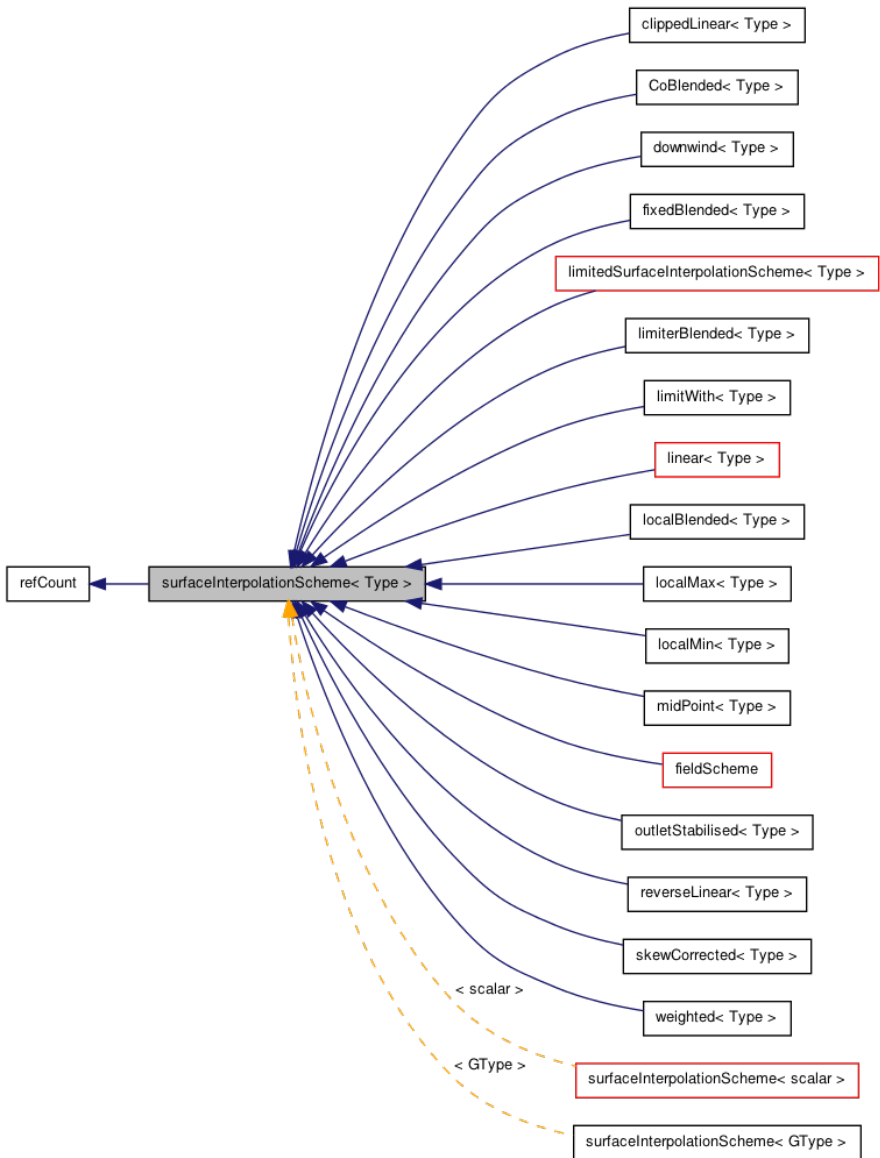
Figure 5.3: Hierarchy of the interpolation schemes based on owner-neighbour addressing.

neighbour addressing of the unstructured mesh with the interpolation resulting in face-centered values. From this, a tree like class hierarchy for the interpolation schemes has been built, as shown in Figure 5.3 with the `surfaceInterpolationScheme` at the root.

From the software design perspective, the interpolation schemes are encapsulated into classes that form a class hierarchy. Some schemes share attributes and functionalities, so it makes sense to organize them as a class hierarchy. As a consequence of this kind of organization, the RTS mechanism allows the user to select different discretization/interpolation schemes at the start (or even during) a simulation. No re-compilation of the program code is required.

5.2.5 Post-processing

Post-processing can be performed after or during the simulation. When post-processing is done after the simulation has finished, the `postProcess` application is used. OpenFOAM provides another distinct method to post-process the data during the simulation by invoking *function objects*.

Post-processing applications are distributed together with OpenFOAM or are written by the users themselves.

WARNING

Before programming a post-processing application, it is advisable to check if one with the desired functionality already exists. OpenFOAM provides a large number of utility applications to choose from.

The `$FOAM_APP/utilities/postProcessing` directory holds all the post-processing applications which are distributed together with OpenFOAM, categorized into different groups.

The `$FOAM_APP/utilities/postProcessing` directory and its subdirectories are the first places to look for an existing post-processing application as shown in figure 5.4. If it is necessary to write a new application, it is likely that some parts of an existing application can used as starting points for the development. The post-processing applications are typically used to compute some integral quantity based on the fields stored during the simulation or to sample field values in specific

```
┌──────────────→ utilities
│  ┌───────────────→ postProcessing
│  │        ┌──────────────→ dataConversion
│  │        ├──────────────→ lagrangian
│  │        ├──────────────→ lumped
│  │        ├──────────────→ miscellaneous
│  │        ├──────────────→ noise
│  │        ├──────────────→ optimisation
│  │        └──────────────→ postProcess
```
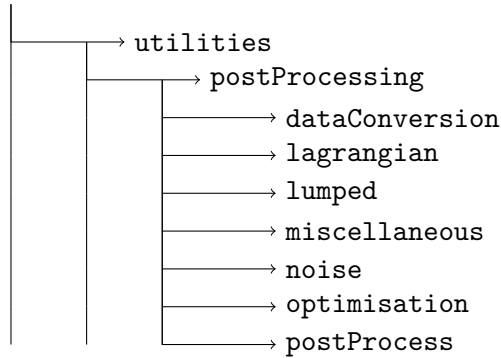
Figure 5.4: Post-processing application categories in OpenFOAM.

parts of the flow domain. This must not be limited to this, though. You can basically compute anything during the post-processing step, using a post-processing application.

Function objects, unlike the post-processing applications, are called during the simulation run. The term *function object* comes from the C++ language terminology, where it denotes a class which is *callable*, since it implements a call operator: `operator()()`. The function object is basically a function encapsulated into a class, which is advantageous e.g. when the function needs to store information about its state after execution. For example, a function object which computes an average maximum pressure in a simulation can stop a simulation if the pressure value exceeds a prescribed value. Such a function needs access to the maximal pressure value and needs to store data required to calculate a running average. Those two attributes are therefore encapsulated into a function object class. More information on function objects in OpenFOAM can be found in chapter 12.

5.3 Often encountered classes

In the previous section, as parts of OpenFOAM are discussed that the user comes into contact with when running simulations, some classes are only briefly mentioned. The following sections provide deeper insights as well as best practices of some selected and frequently used classes in OpenFOAM.

5.3.1 Dictionary

The `dictionary` class is one of the first classes an OpenFOAM user interacts with. While the dictionary class interface is not very complex, some aspects of it may not be apparent to a novice user.

Reading data from a dictionary

Reading dictionary entry values is the most basic form of working with the dictionary class. The `dictionary` class interface provides multiple methods that can be used to read data. The `getOrDefault` method is the commonly used example of this as it not only provides read access to the specified dictionary entry, it also defines a default value if no entry is found. This eliminates runtime errors caused by missing non-critical input values.

```
const auto& solution(mesh.time().solutionDict());
const auto name(solution.getOrDefault<word>("parameter1"));
const auto vector1(solution.getOrDefault<vector>("vector1");
```

As shown in the above code, `getOrDefault` requires more than just the name of the parameter to read. It also requires a template argument with the name of the data type that is to be looked up in the dictionary.

Accessing the table of contents

The table of contents contains the names of sub-dictionaries of a dictionary. In the following example, the table of contents of the `fvSolution` dictionary file read by the `Foam::Time` class is accessed:

```
const dictionary& fvSolutionDict(mesh.time().solutionDict());
Info<< fvSolutionDict.toc() << endl;
```

Accessing sub-dictionaries

Accessing sub-dictionaries is frequently encountered while developing OpenFOAM, because the custom classes or solution algorithms developed in OpenFOAM will be configured by dictionary files. Assuming the dictionary `A` located in the `constant` directory contains the following data:

```
axis    (1 0 0);
origin  (5 10 15);
type    "modelA";
modelA
{
    name    "uniform";
}
```

In order to access the `modelA` sub-dictionary in the simplest form, the following code suffices:

```
const dictionary dict(fileName("constant/A"));
const auto& subDict(dict.subDict("modelA"));
```

5.3.2 Dimensioned types

Dimensioned types attach units of measurement to scalars, vectors and tensors. They extend the tensor arithmetic operations to include units of measurement: *dimensions* in OpenFOAM terminology. For example, both the velocity **U** and momentum ρ**U** are vectors, however they cannot be added, since they differ in their dimensions. Dimension checking in OpenFOAM is implemented by a class template `dimensioned<Type>`.

The `dimensioned<Type>` is a wrapper (adapter) class that delegates the computation of the tensor arithmetic to the wrapped tensor `Type` and the dimension checking to the wrapped `dimensionSet` object. Investigating the arithmetic operators of `dimensioned<Type>` class template leads to the implementation of the `+=` operator shown below:

```
template<class Type>
void Foam::dimensioned<Type>::operator+=
(
    const dimensioned<Type>& dt
)
{
    dimensions_ += dt.dimensions_;
    value_ += dt.value_;
}
```

It can be seen that there are two arithmetic operations being performed: one for the dimension (units) and the other for the numerical value of the tensor. The arithmetic operators of the `dimensionSet` class are responsible for the dimension checking process. The source code of the `+=` arithmetic operator of the `dimensionSet` class looks like the following:

```
bool Foam::dimensionSet::operator+=(const dimensionSet& ds) const
{
    if (dimensionSet::debug && *this != ds)
    {
        FatalErrorIn("dimensionSet::operator+=(
            const dimensionSet&) const")
            << "Different dimensions for +=" << endl
            << "    dimensions : " << *this << " = "
            << ds << endl << abort(FatalError);
    }

    return true;
}
```

This provides enough information to conclude exactly how the dimension
check is performed: a fatal error which aborts the program execution
is generated when the addition operation is performed for differently
(!=) dimensioned sets, provided dimension checking is turned on. The
`dimensionSet` class implements dimensions as a set of integer exponents
of the physical measures as shown below:

```
//- Define an enumeration for the names of the dimension exponents
enum dimensionType
{
    MASS,                 // kilogram  kg
    LENGTH,               // metre     m
    TIME,                 // second    s
    TEMPERATURE,          // Kelvin    K
    MOLES,                // mole      mol
    CURRENT,              // Ampere    A
    LUMINOUS_INTENSITY  // Candela    Cd
};
```

As shown in the code below, the `!=` dimension check operator is im-
plemented in terms of the equality operator `==`. The checking procedure
iterates over the dimensions of the operating dimensionSets. Through the
loop it tests if the magnitude of the difference between dimension expo-
nents is large enough to consider the sets unequal (`> smallExponent`).

```
bool Foam::dimensionSet::operator==(const dimensionSet& ds) const
{
    for (int Dimension=0; Dimension < nDimensions; ++Dimension)
    {
        if
        (
            mag(exponents_[Dimension] - ds.exponents_[Dimension])
          > smallExponent
        )
        {
            return false;
        }
```

```
    }

    return true;
}
```

The value of `smallExponent` is a class-static variable:

```
atic const scalar smallExponent;
```

initialized to `SMALL`, which is equal to 10^{-15} for double precision scalars.

Turning on dimension checking is done by setting the debug flag to *on* for the `dimensionedSet` class in the `$WM_PROJECT_DIR/etc/control-Dict`:

```
DebugSwitches
{
    Analytical       0;
    APIdiffCoefFunc    0;

    ...

    dictionary       0;
    dimensionSet     1;
    mappedBase    0;
    ...
```

The conclusions presented for the `+=` operator are exactly the same for other dimensioned tensor arithmetic operations. Dimension checking is activated by default and should generally not be deactivated in Open-FOAM. Even if a custom application implements equations in dimensionless form, those equations will be scaled with what should be dimensionless numbers. The dimension checking system can check if for example the Reynolds number

```
auto Re = Foam::mag(U) * L / mu;
Info << Re.dimensions() << endl;
```

is really dimensionless, in which case the output will be a `dimensionSet` that only contains zeros. However, if a mistake was done in calculating, say L, dimension checking that relies on the SI measurement unit system will pick this up. So, dimension checking should be used even when equations are written in dimensionless form, to check for scaling errors.

The most popular name synonyms for `dimensioned<Type>` are `dimensionedScalar` and `dimensionedVector` and are used in the following examples. Also, note that the dimensioned types are constructed in a slightly more complex manner than what might be expected from what

EXERCISE

Why doesn't the arithmetic operator `+=` of the `dimensionSet` class perform an actual arithmetic operation when the `if` block is evaluated as `false`?

INFO

For many class templates in OpenFOAM with a complex name, there is a `typedef` provided. In C++ the `typedef` keyword allows the programmer to define shorter and more concise type names. Even though this `typedef` might not be much shorter of the original, it does at least save the typing of the `<Type>` part.

In the case of `dimensioned<Type>` the emphasis is not on name length, but on code style - `dimensionedVector` is a name in *camel case* which is used for types in OpenFOAM application level code.

is described so far. Rather than requiring the tensor value and the dimensioned set, the dimensioned types require an additional parameter: a name. For example, if two `dimensionedVector` objects are constructed in the following way:

```
dimensionedVector velocity
(
    "velocity",
    dimLength / dimTime,
    vector(1,0,0)
);

dimensionedVector momentum
(
    "velocity",
    dimMass * (dimLength / dimTime),
    vector(1,0,0)
);
```

Please note that there are some pre-defined `dimensionedSet` objects used to initialize the `velocity` and `momentum` `dimensionedVector` objects: `dimLength`, `dimTime` and `dimMass`. These are constant and global objects, and they can be found in the source file `dimensionedSets.C`:

```
const dimensionSet dimless(0, 0, 0, 0, 0, 0, 0);
const dimensionSet dimMass(1, 0, 0, 0, 0, 0, 0);
const dimensionSet dimLength(0, 1, 0, 0, 0, 0, 0);
const dimensionSet dimTime(0, 0, 1, 0, 0, 0, 0);
const dimensionSet dimTemperature(0, 0, 0, 1, 0, 0, 0);
const dimensionSet dimMoles(0, 0, 0, 0, 1, 0, 0);
```

```
const dimensionSet dimCurrent(0, 0, 0, 0, 0, 1, 0);
const dimensionSet dimLuminousIntensity(0, 0, 0, 0, 0, 0, 1);
```

INFO

Because the fundamental physical dimension units are used to construct the complex ones (e.g. $N = kgm/s^2$), use the global predefined `dimensionSet` objects when defining your own dimension sets to improve code readability.

Moving on to arithmetics of dimensioned types and adding `velocity` to `momentum` like this:

```
momentum += velocity;
```

will result in a following error at the program execution:

```
--> FOAM FATAL ERROR:
Different dimensions for +=
dimensions : [1 1 -1 0 0 0 0] = [0 1 -1 0 0 0 0]


From function dimensionSet::operator+=(const dimensionSet&) const
in file dimensionSet/dimensionSet.C at line 179.

FOAM aborting
```

The dimension checking process in OpenFOAM is performed at runtime. As a result, code which contains errors in dimension operations will compile, but will not run. Dimension checking errors in OpenFOAM can be easily debugged with a debugger (for example `gdb`), provided OpenFOAM and the custom code are compiled in the `Debug` mode.

EXERCISE

Turn off dimension checking using the aforementioned debug flag and run the arithmetic addition for momentum and velocity.

5.3.3 Smart pointers

Pointers are a special kind of variable which stores the memory address of an object, that can be used to refer to that object. In C++ it is possible to pass objects either by *value* or by *reference*, with the latter being significantly faster for larger objects. This is not only faster but also

more conservative in terms of memory usage as no data is temporarily copied. Returning larger objects by value and passing them as function arguments can be much more computationally expensive, compared to working with only their pointers.

A good example for the application of pointers in OpenFOAM are the functions which implement interpolation algorithms. Interpolations operate on fields, which in CFD often have hundreds of thousands of components per CPU core. If the interpolation algorithm is implemented as a function, the function can provide the necessary field result in two ways, by returning the result as an object

```
result = function(input)
```

or by modifying a result argument that is passed as a non-constant reference to the function

```
function(result, input)
```

The first option is preferred for discretization algorithms (operators) because they are often composed of arithmetic expressions to build mathematical models. For example, consider the momentum conservation equation code taken from the `interFoam` solver:

```
fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(rhoPhi, U)
  + turbulence->divDevRhoReff(rho, U)
);
```

The result of the sum of operators acting upon fields `rho`, `U` and `rhoPhi` will be a coefficient matrix (`fvVectorMatrix`). As a consequence, the following points must be fulfilled from the above code:

- `fvVectorMatrix` needs to be copy-constructable,
- all operators must *return* a `fvVectorMatrix`,
- the `fvVectorMatrix` addition operator must return a `fvVector-Matrix`.

If the operators `ddt` and `div` had been implemented as functions that take modifiable parameters, writing mathematical models easily (usually referred to as *equation mimicking*) would not be possible. The matrices that are returned by the functions are quite large, so returning them

by value introduces the penalty of creating temporary objects. Open-FOAM avoids creating the unnecessary copy operations in an explicit way not relying on compiler optimizations, with the help of natively implemented OpenFOAM *smart pointers*. A smart pointer is initialized within the interpolation function, and is returned by value. The same goes for other operations involving equation discretization, such as convection schemes. For example, the `fvmDiv` divergence operator of the Gauss convection scheme initializes such a smart pointer (`tmp<fvMatrix<Type>>`) is shown below

```
tmp<fvMatrix<Type> > tfvm
(
    new fvMatrix<Type>
    (
        vf,
        faceFlux.dimensions()*vf.dimensions()
    )
);
```

Afterwards it performs the calculations that are responsible for defining the elements in the coefficients matrix shown:

```
fvm.lower() = -weights.internalField()*faceFlux.internalField();
fvm.upper() = fvm.lower() + faceFlux.internalField();
fvm.negSumDiag();

forAll(vf.boundaryField(), patchI)
{
    const fvPatchField<Type>& psf =
        vf.boundaryField()[patchI];
    const fvsPatchScalarField& patchFlux =
        faceFlux.boundaryField()[patchI];
    const fvsPatchScalarField& pw =
        weights.boundaryField()[patchI];

    fvm.internalCoeffs()[patchI] =
        patchFlux*psf.valueInternalCoeffs(pw);
    fvm.boundaryCoeffs()[patchI] =
        -patchFlux*psf.valueBoundaryCoeffs(pw);
}

if (tinterpScheme_().corrected())
{
    fvm += fvc::surfaceIntegrate
        (
            faceFlux*tinterpScheme_().correction(vf)
        );
}
```

The *smart pointer* initialized at the beginning of the function is then returned by value:

```
    return tfvm;
```

Returning a smart pointer to the finite volume matrix (`tfvm`) by value results in a copy operation *of the smart pointer*. However, there is a significant difference between a copy of the smart pointer and a copy of an entire matrix object: the pointer's value is only the address of the matrix, and not the entire matrix itself. This approach substantially increases the efficiency of the implementation.

> INFO
>
> Avoiding unnecessary copy operations has a commonly used shorter name: *copy elision*. Copy elision can be enforced in various ways in the C++ programming language: compiler optimizations (Return Value Optimization, Named RVO), Expression Templates (ET), or by using rvalue references and move semantics provided by the C++11 language standard.

In general, when pointers are used, they point to objects which are created on the heap using the operator `new`:

```
    someType* ptr = new someType(arguments...);
```

Since the C++ programming language does not, on purpose, support automatic garbage collection[1], the programmer is left responsible for releasing the resources.

Thus, each invocation of the `new` operator needs to be followed by a corresponding call to an appropriate `delete` operator. Of course, the call to the `delete` operator must be placed in an appropriate location, such as a class destructor. This opens the possibility that the programmer simply forgets to delete pointers, which results in memory leaks. As an alternative source of errors, accessing a part of the memory which is referred to by an already deleted pointer will lead to undefined behavior. Both issues will occur at runtime and are usually sources of errors which are notoriously difficult to find and debug. To circumvent both problems, direct handling of raw pointers is to be avoided. In C++, handling of raw pointers has been replaced by an idiom called Resource Acquisition Is Initialization (RAII).

The RAII idiom states that raw pointers need to be encapsulated into a class, whose destructor takes care of deleting the pointer and releasing

---

[1]Automatic garbage collection referrs to the deletion of heap-allocated objects.

the resource at the appropriate place in the code. Adapting raw pointers in such classes, and providing different functionalities to the adapted raw pointers has lead to the development of so-called *smart pointers*. Different smart pointers exist and provide different functionalities. OpenFOAM implements two of such smart pointers: `autoPtr` and `tmp`.

Provided that the environmental variables are set by sourcing the `etc/bashrc` configuration script of the example code repository, the example application code is made available in the folder `$PRIMER_EXAMPLES_SRC/applications/test/`. Those readers that are interested in following the tutorials presented in the next sections step-by-step, need to create a new executable application - `testSmartPointers`. Creating new applications in OpenFOAM is simplified, since scripts are available that generate *skeleton* directories for applications. To create a new application, a directory is chosen where the application code will be placed, and the following commands are executed:

```
mkdir testSmartPointers
cd testSmartPointers
foamNew source App testSmartPointers
sed -i 's/FOAM_APPBIN/FOAM_USER_APPBIN/g' Make/options
```

The final line replaces the placement directory for the application binary file from the platform directory, to the user application binaries directory.

> **INFO**
>
> It is a good practice to build your own applications into the `$FOAM_USER_APPBIN`, which needs to be specified in the `Make/options` build configuration file.

> **INFO**
>
> In this section, the `gcc` compiler is used. Be aware of this when reading about specific compiler flags in the text.

Using the `autoPtr` smart pointer

In order to illustrate how the `autoPtr` is used, a new class needs to be defined for the used examples. This class should notify the user each time its constructors and destructor are called. For sake of this example,

it inherits from the basic field class template `Field<Type>`, and is named `infoField`. Any other class could be used, since the optimizations performed by the compiler for copy operations do not depend on the size of the objects.

To start, the `infoField` class template can be defined as shown below:

```
template<typename Type>
class infoField
:
    public Field<Type>
{
    public:

        infoField()
            :
                Field<Type>()
        {
            Info << "empty constructor" << endl;
        }

        infoField(const infoField& other)
            :
                Field<Type>(other)
        {
            Info << "copy constructor" << endl;
        }

        infoField (int size, Type value)
            :
                Field<Type>(size, value)
        {
            Info << "size, value constructor" << endl;
        }

        ~infoField()
        {
            Info << "destructor" << endl;
        }

        void operator=(const infoField& other)
        {
            if (this != &other)
            {
                Field<Type>::operator=(other);
                Info << "assignment operator" << endl;
            }
        }
};
```

The class template inherits from `Field<Type>` and uses the following:

- empty constructor
- copy constructor

- destructor
- assignment operator

Each time those functions are used, an `Info` statement signals the particular call to the standard output stream. This is solely done for the purpose of obtaining information on which function has been executed.

To continue with the example, a function template needs to be defined which returns an object by value:

```
template<typename Type>
Type valueReturn(Type const & t)
{
    // One copy construction for the temporary.
    Type temp = t;

    // ... operations (e.g. interpolation) on the temporary variable.

    return temp;
}
```

The name of the type used in the example is shortened in order to reduce the amount of unnecessary typing:

```
// Shorten the type name.
typedef infoField<scalar> infoScalarField;
```

In the main function the following lines are implemented:

```
Info << "Value construction : ";
infoScalarField valueConstructed(1e07, 5);

Info << "Empty construction : ";
infoScalarField assignedTo;

Info << "Function call" << endl;
assignedTo = valueReturn(valueConstructed);
Info << "Function exit" << endl;
```

Compiling and executing the application with either `Debug` or `Opt` options will produce exactly the same results, even though the `Debug` option turns of compiler optimizations. As mentioned at the section beginning: the compiler is very clever at recognizing the fact that a temporary object is returned only to be discarded away after assignment. Another advantage of not using the geometrical fields is that this small example application does not need to be executed within an OpenFOAM simulation case directory. It can be called directly from the directory where the code is stored. Executing the application results in the following output:

```
?> testSmartPointers

Value construction : size, value constructor
Empty construction : empty constructor
Function call
copy constructor
assignment operator
destructor
Function exit
destructor
destructor
```

Considering each line of the output individually, while comparing it to the `valueReturn` function code, interestingly reveals that *the temporary return variable was never constructed*. A copy construction is missing for the `return` statement of the function, since the function *returns by value*, as well as a corresponding destructor call. The reason for this kind of behavior is the *copy elision* optimization performed automatically by the compiler, that is present even in the `Debug` mode. In order to remove this optimization, an additional compiler flag can be added to the `Make/options`:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -fno-elide-constructors

EXE_LIBS = \
    -lfiniteVolume
```

The `gcc` compiler flag `-fno-elide-constructors` will prevent the compiler from performing optimizations, important to enable tracking of what happens in `valueReturn`, that would otherwise be optimized-away by the compiler.

> **INFO**
>
> Remember to call `wclean` before executing `wmake`. Modifications to the `Make/options` file are not recognized by the `wmake` build system as a source code modification that requires re-compilation.

Compiling and running the application with the `options` file as defined above results in the following output[2]:

```
?> testSmartPointers
  Value construction : size, value constructor
  Empty construction : empty constructor
  Function call
```

---

[2]Please note that the comments, starting with a **#** are added by the authors

```
copy constructor # Copy construct tmp
copy constructor # Copy construct the temporary object
destructor # Destruct the tmp - exiting function scope
assignment operator # Assign the temporary object to assignedTo
destructor # Destruct the temporary object
Function exit
destructor
destructor
```

The output shows the unnecessary creation and deletion of the temporary object. Eliding copies of temporary objects by performing an in-place construction of an object at the point where the function returns has become a standard option for compilers. It happens regularly that this feature cannot be disabled even when surpressing optimizations in Debug mode with the compiler flags:

```
-O0 -DFULLDEBUG
```

To disable the constructor copy eliding optimization the compiler flag `-fno-elide-constructors` had to be passed explicitly in `Make/op-tions`.

TIP

The above example serves one major purpose: To show that it is not necessary to use the `autoPtr` smart pointer in expressions where a named temporary object is returned anyway. On modern compilers, even when compiling in the debug mode (for OpenFOAM this means setting `$WM_COMPILE_OPTION` to Debug), this optimization is turned on by default.

Some of the most prominent examples for the usage of the `autoPtr` are within models employing RTS, such as `turbulenceModel` and `fvOptions`. The particular base-classes are instantiated based on user specified keywords in a dictionary. Using the `turbulenceModel`-class as an example, the base class is used as template argument for `autoPtr` in e.g. the solver application and the RTS can then instantiate the specific turbulence model into the `autoPtr`. RTS allows class users to instantiate objects of a specific class in a class hierarchy at runtime. Instantiating objects in that way enables the ability of C++ to access the derived class object via a base class pointer or reference. This is usually referred to as *dynamic polymorphism*.

The turbulence models are typically instantiated in the particular solver's `createFields.H`, which is included before the beginning of the time

INFO

It is impossible to cover all details of the C++ language standard used by OpenFOAM in this book. Whenever a C++ construct is encountered that does not sound familiar, it should be looked up elsewhere.

loop. The relevant lines are the following:

```
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

Obviously, an `autoPtr` is used to store the turbulence model. *If raw* pointers would have been used instead to access the `RASModel` object, another line in the solver code would be required, which deletes the raw pointer, in order to *deallocate* the memory at the end of the solver. With `autoPtr` being a smart pointer, this deletion is performed automatically within the `autoPtr` destructor. The code for the raw pointer method (which luckily is not used anywhere in OpenFOAM) would look something like this:

```
incompressible::RASModel* turbulence =
    incompressible::RASModel::New(U, phi, laminarTransport)
```

At the end of the solver code, the following lines would be required:

```
delete turbulence;
turbulence = NULL;
```

Of course, for a single pointer, adding the lines to release resources might not be a problem. However, the RTS is used for: transport models, boundary conditions, fvOptions, discretization schemes, interpolation schemes, gradient schemes and so forth. All of those objects are small compared to things like fields and the mesh, so could they be returned by value? From the standpoint of efficiency - yes, especially considering the Return Value Optimization (RVO) is implemented by all modern compilers. From the standpoint of flexibility - there is no chance of doing that, since the dynamic polymorphism relies on access via pointers or references. Keeping the runtime flexibility high, means relying on pointers or references.

TIP

Using `autoPtr` or `tmp` is necessary when RTS is used to select objects at runtime.

The following example examines the `autoPtr` interface and its ownership-based copy semantics. The `autoPtr` owns the object it points to. This is expected, as RAII requires the smart pointer to handle the resource release so the programmer doesn't have to. As a consequence, creating copies of `autoPtr` is complicated - copying an `autoPtr` invalidates the original `autoPtr` and transfers the object ownership to the copy. To see how this works, consider the `main` function of the following code snippet:

```cpp
int main()
{
    // Construct the infoField pointer
    autoPtr<infoScalarField> ifPtr (new infoScalarField(1e06, 0));

    // Output the pointer data by accessing the reference -
    // using the operator T const & autoPtr<T>::operator()
    Info << ifPtr() << endl;

    // Create a copy of the ifPtr and transfer the object ownership
    // to ifPtrCopy.
    autoPtr<infoScalarField> ifPtrCopy (ifPtr);

    Info << ifPtrCopy() << endl;

    // Segmentation fault - accessing a deleted pointer.
    Info << ifPtr() << endl;

    return 0;
}
```

It is important to notice that once a copy of the `autoPtr` object is performed, the ownership of the object pointed to is transfered. Commenting out the line that causes a segmentation fault, the resulting application output looks like this:

```
?> testSmartPointers
  size, value constructor
  1000000{0}
  1000000{0}
  destructor
```

Obviously only a single constructor and destructor of the `infoScalarField` class are envoked, even though the `autoPtr` objects are passed by value. Hence `autoPtr` can be used to save unnecessary copy operations.

Using the `tmp` smart pointer

The `tmp` smart pointer prevents unnecessary copies of objects by performing reference counting. Reference counting is a process where the same object is being passed around. In this context, the reference counting is performed using the `refCount` class. This data structure is an essential base class for any class that is supposed to be stored in a `tmp` smart pointer. It is wrapped by the smart pointer, with the number of references to this object being increased each time a copy or assignment of the smart pointer is performed.

Complying with RAII, the destructor of the `tmp` pointer is responsible for destroying the wrapped object. The destructor checks the number of `refCount` the object has in the current scope. If this number is greater than zero, the destructor simply reduces `refCount` by one and allows the object to live on. As soon as the destructor is called in a situation where the `refCount` to the wrapped object has reached zero, the destructor deletes the object.

> TIP
>
> The definition of the `tmp` smart pointer can be found in `$FOAM_SR-C/OpenFOAM/memory/tmp`.

There is a catch when using the `tmp` smart pointer: the pointer class template is not made responsible for counting the references. Reference counting is expected from the wrapped type. This can be easily checked when examining the class destructor:

```cpp
template<class T>
inline Foam::tmp<T>::~tmp()
{
   if (isTmp_ && ptr_)
   {
      if (ptr_->okToDelete())
      {
         delete ptr_;
         ptr_ = 0;
      }
      else
      {
         ptr_->operator--();
      }
   }
}
```

The `ptr_` attribute is the wrapped raw pointer to an object of type `T` and the destructor attempts to access two member functions:

- `T::okToDelete()`
- `T::operator--()`

As a result, the wrapped object must adhere to a specific class interface. Another way to test this catch is to try and use the `tmp` with a trivial class that can be define as follows:

```
class testClass {};
```

Then trying to wrap this class into a `tmp` smart pointer:

```
tmp<testClass> t1(new testClass());
```

The above code results in the following errors:

```
tmpI.H:108:9: error: 'class testClass'
has no member named 'okToDelete' if (ptr_->okToDelete())

tmpI.H:115:13: error: 'class testClass'
has no member named 'operator--' ptr_->operator--();
```

By means of this error, the compiler complains about the fact that the aforementioned member functions are not implemented by `testClass`.

Since OpenFOAM makes the objects perform the reference counting, it has been encapsulated into a class that such objects inherit from, named `refCount`. The `refCount` class implements the reference counter and the related member functions.

> **TIP**
>
> In order to use `tmp<class T>` in OpenFOAM, the type `T` of the wrapped object should inherit from `refCount`.

If the `testClass` is modified to inherit from `refCount`:

```
class testClass : public refCount {};
```

It can then be wrapped with `tmp`. To see how reference counting works and how unnecessary construction of objects is avoided, the `tmp` can be used with the class `infoScalarField`, that was used in the examples describing `autoPtr`. The `refCount` class allows the user to get information on the current reference count with the member function `refCount::count()`, which is used in the following example. Artificial scopes are used to decrease the life-span of `tmp` objects so that their

destructors are called. This would occur in a regular program code when nested function calls or loops are present. Here is the example code:

```
tmp<infoScalarField> t1(new infoScalarField(1e06, 0));
Info << "reference count = " << t1->count() << endl;
{
    tmp<infoScalarField> t2 (t1);
    Info << "reference count = " << t1->count() << endl;
    {
        tmp<infoScalarField> t3(t2);
        Info << "reference count = " << t1->count() << endl;
    } // t3 destructor called

    Info << "reference count = " << t1->count() << endl;
} // t2 destructor called
Info << "reference count = " << t1->count() << endl;
```

This results in the following command line output:

```
>? testSmartPointers
 size, value constructor
 reference count = 0
 reference count = 1
 reference count = 2
 reference count = 1
 reference count = 0
 destructor
```

A single constructor and the corresponding destructor output coming from the `infoField` class shows that only a single construction and destruction was performed, even though the `tmp` objects were passed by value as function arguments.

5.3.4 Volume fields

An indepth discussion of boundary fields and boundary conditions, including the theory behind them, is covered by chapter 10.

Volume fields are those fields typically used to store cell centered field values. Depending on the tensor type stored by the field, either a `vol-ScalarField`, `volVectorField` or a `volTensorField` can be used. There are surface- and point- fields as well, which store field values at the face centers and cell corner-points, respectively. Note that all mentioned fields are constructed similarly, independent of their type.

The fields that map values to the elements of the unstructured mesh in OpenFOAM are implemented by the general `GeometricField` class template. The specific fields, such as volume fields, surface fields and

similar other fields are then generated in the form of concrete classes by instantiating the `GeometricField` class template with specific template arguments. As noted in the section on the dimensioned types, the type names used for fields in OpenFOAM are shortened for convenience using the `typedef` keyword.

INFO

Compilation errors that involve fields such as `volScalarField` cause C++ template errors, which are quite long and not intuitive to read. Because the fields are instantiations of the `GeometricField` template, the `GeometricField` class template source code can investigated to get more information about the error.

Before being able to work with an object of a field, it must be constucted, just like with any other object. The class interface has several overloaded constructors which may come in handy, depending on the particular situation. The simple constructor is the *copy constructor*:

```cpp
// Assuming that the volScalarField p exists
const volScalarField pOld(p);
```

As the name suggests, it constructs a copy of the original, which is type-identical. However, in order to use this constructor, a `volScalarField` must be present in the first place. Two approaches can be used for this: either read field data from a file or generate it from from scratch. The initialization of a field based on an input file is fairly straightforward and an example for it can be found in the `createFields.H` file of every solver or processing application in OpenFOAM. For that purpose, the `IOobject` is leveraged for performing the file system level input output for accessing that file and acts as a wrapper class around this. It is required by the field, in order to be constructed from a file:

```cpp
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

The above code initializes the `volScalarField` based on the contents of the T file, which must be present in a time directory (or at least the 0-directory). Of course, the T file has to be of proper formatting, in order to be used in the constructor for the `IOobject`, which then gets used to construct the `volScalarField`. If the T file does not exist, the execution of the code will stop due to the `IOobject::MUST_READ` instruction. Other instructions can be selected as well, depending on particular needs. The remaining two options are `READ_IF_PRESENT` and `NO_READ`. The first one only reads the file if it's present in any of the time directories, otherwise it's constructed via provided default values. `NO_READ` never reads anything from a file, just constructs the field.

In some circumstances, fields need to be constructed without reading data from a file. The flux field is a good example for that:

```
surfaceScalarField phi
(
    IOobject
    (
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(rho*U) & mesh.Sf()
);
```

Here the `phi` field is constructed from the field data itself if the file is present. Otherwise, the flux is calculated from the velocity field directly.

This shows the difference between the construction of the T field and the `phi` field; both of which take an `IOobject` as a first argument but the construction of the T field takes a `polyMesh` as a second argument, whereas phi takes a `surfaceScalarField`. If you have a closer look at $FOAM_SRC/OpenFOAM/fields/GeometricFields/Geo-metricField/GeometricField.H, you will discover that there is a huge variety of overloaded constructors for the `GeometricField` class, which is the *base-class*, of all fields in OpenFOAM. You can basically use all of them, depending on the particular needs.

Accessing cell data

Specific cells can be addressed by invoking the access operator `Type&`
`operator[](const label cellI)` of the particular field and passing the
desired cell label as an argument. When performing arithmetic operations
with any field, it is not advisable to do this on a cell-to-cell basis. Instead,
the field arithmetic operators should be used.

> **INFO**
>
> Using loops for fields in the application-level program code reduces
> code readability and may cause a significant drop in computational
> efficiency.

In conclusion, this selection of specific cells should only be used if a
subset of cells needs to be used for the computation. The following code
shows an example of how cells of the pressure and velocity fields can be
selected:

```
labelList cellIDs(3);
cellIDs[0] = 1;
cellIDs[1] = 42;
cellIDs[2] = 39220;

forAll(cellIDs, cI)
{
    Info<< U[cellIDs[cI]] << tab << p[cellIDs[cI]] << endl;
}
```

Accessing boundary fields

As covered in chapter 1, the internal field values are separated from the
boundary field values. Such logical separation of cell centered and bound-
ary (face-centered) values is defined by the principles of the numerical
interpolations that support the FVM.

Separating boundary field values from internal field values has an impor-
tant impact onto the way the algorithms are parallelized in OpenFOAM.
Numerical operations are paralellized in OpenFOAM using data paral-
lelism where the domain is decomposed into sub-domains and the nu-
merical operations are executed on each separate sub-domain. As a result,
a number of parallel processes are executed which require communication

with each other across the decomposition (processor) boundaries. Modeling the process boundary as a boundary condition results in automatic parallelization of all the numeric operations based on owner-neighbor addressing in OpenFOAM. Automatic parallelization of the unstructured FVM discretization is a noteworthy feature of OpenFOAM.

The following example shows how to access the boundary field values of a volumetric scalar field `pressure`, for the boundary patch `outlet`. For more information on boundary fields and their differentiation towards the internal field, please referr to chapter 10. The boundary field on the outlet can be found based on the patch ID mapped to the name of the boundary mesh patch (a sub-set of the boundary mesh):

```
const label outletID(mesh.boundaryMesh().findPatchID("outlet"));
```

and the boundary field is then accessed using the `GeometricField::boundaryField` member function:

```
const scalarField& outletPressure =
    pressure.boundaryField()[outletID];
```

The volumetric field `pressure` has a member function `boundaryField` which returns a list of pointers to the boundary fields. The position of the `outlet` boundary field in that pointer list is defined by the `outletID` label (index). The above code snippet sets the boundary field as a constant reference `outletPressure`, however non-constant access to the boundary field is provided by the member function. This can be confirmed by observing the declaration of the `GeometricField` class template:

```
//- Return reference to GeometricBoundaryField
GeometricBoundaryField& boundaryFieldRef();
```

The `GeometricBoundaryField` is a class template which is parameterized with the same parameters as the `GeometricField` and the definition of this class template is placed in the public part of the `GeometricField` class interface.

Further reading

[1] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer, 2002.

[2]   Hrvoje Jasak, Aleksandar Jemcov, and Željko Tuković. "Open-FOAM: A C++ Library for Complex Physics Simulations". In: *Proceedings of the International Workshop on Coupled Problems in Numerical Dynamics (CMND 2007)* (2007).

[3]   H. K. Versteeg and W. Malalasekra. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach*. Prentice Hall, 1996.

6

Productive programming with OpenFOAM

Any software project of a significant size demands some level of organization. From the directory organization to the Version Control System (VCS), there are many aspects of code development that call for standard practices.

Development of CFD applications puts a strong emphasis on computational efficiency because of the large datasets as well as the large number of computational operations involved in a numerical simulation. Without careful consideration of even simple algorithms, crippling bottlenecks can be created. Resolving problems in the code such as execution errors (bugs) or computational bottlenecks can typically be completed faster and easier with the appropriate tools.

Running simulations in parallel mode on an High Performance Computing (HPC) cluster requires the user to first install OpenFOAM on a cluster. Even in the case when a complete OpenFOAM installation is available on the cluster, having background knowledge on that topic makes it easier for the user to more accurately assess possible installation problems and report them to the cluster administrator.

In this chapter, best practices are covered that increase productivity when programming OpenFOAM and using OpenFOAM on an HPC cluster.

6.1 Code organization

This section describes the organization of the OpenFOAM source code and the workflow associated with developing OpenFOAM. When performing code development, the code should be organized into two layers: library code and application code. The library code may contain single or multiple *libraries* that implement various re-usable parts or components. This entirely depends on the designer of the library and how he or she structured the library. Sometimes the library level code is referred to as application logic. The library code is, by design, not specific to one single application: it is re-used by many executable applications. A library will contain declarations of functions or classes, as well as their implementation. The library is usually compiled into what is called 'object code', to save compilation time. The library object code is then linked to the application when it is executed by a linker program. Note that while libraries contain compiled code, they cannot be executed in the command line like executable applications.

The application code on the other hand is using the library code to assemble a higher level functionality. A flow solver of OpenFOAM is a good example of this, as it combines separate libraries to handle conceptually distinct tasks such as disk I/O, mesh handling, discretisation, etc. The author of the solver does not have to take care of the logic behind the respective libraries but can concentrate on developing the solver.

When source code is organized into application and library layers, it is typically more easily extensible and can be shared more readily with others. Because OpenFOAM follows this approach, the top-level directory `$WM_PROJECT_DIR` holds two sub-directories: `applications` and `src`. The `src` folder stores the various libraries while the `applications` folder stores the executable applications that make use of those libraries.

During development, the programmer has two options for code organization: programming within the OpenFOAM directory structure, or programming within a separate directory structure. Programming within the OpenFOAM structure makes sense at first glance as it is logical to keep similar code in close proximity within the directory trees. This 'main structure' approach to development can, unfortunately, quickly cause problems when collaborating with others using a Version Control

System (VCS). Version control systems are absolutely essential for collaborative programming. Even if if a developer is working alone, VCS significantly speeds up development and safety of the development process, because it enables straightforward investigation of alternative ideas. Even if a version control system is used properly, sharing custom code located within the main OpenFOAM repository with others might be problematic for the following reasons:

- there is not a clear overview of the files that belong to the project,
- tutorials and test cases are placed in a separate directory structure,
- handling additional dependencies that are not distributed alongside OpenFOAM might cause problems with change integration,
- collaborating with others requires having access to a full-fledged OpenFOAM release.

The last point makes collaborative work difficult, as creating a clone of the entire release repository makes it necessary for anyone to clone the entire OpenFOAM platform in order to collaborate on a usually significantly smaller new project. There are additionally situations when the projects are not shared with the general public: they are developed by research departments in companies, or the functionality is not yet mature enough for a release. In such cases when the project is not to be integrated with an OpenFOAM release from the start, bundling a library and application code in a single separate repository, that can be compiled directly, makes individual and collaborative development much easier. At a latter point, when the project proved to be of higher quality, integration into one of the OpenFOAM release projects can be performed. Nowadays there are multiple VCS hosting services available. These services provide advanced web interfaces allowing the user to use bug tracking, hosting a wiki for each project, and other tools that make working on such a project much easier. Two of the most popular ones are `gitlab`, `github` and `bitbucket`.

6.1.1 Directory structure of a new OpenFOAM project

Applying the same organization structure of OpenFOAM to a custom OpenFOAM project simplifies collaborative work and future integration into OpenFOAM. If the code is organized this way, the structure of the directories will be more self-explanatory to users familiar with the directory structure of OpenFOAM. Maintaining a uniform directory structure is

```
primer-examples
                ┌────────→ applications
                ├───────→ solvers
                ├──────→ test
                ├─────→ utilities
                │           ┌────────→ postProcessing
                │           └───────→ preProcessing
        ├───────→ doxyfile
        ├──────→ etc
        ├──────→ doc
        ├──────→ src
        ├──────→ README
        ├──────→ Allwmake
        └──────→ Allwclean
```
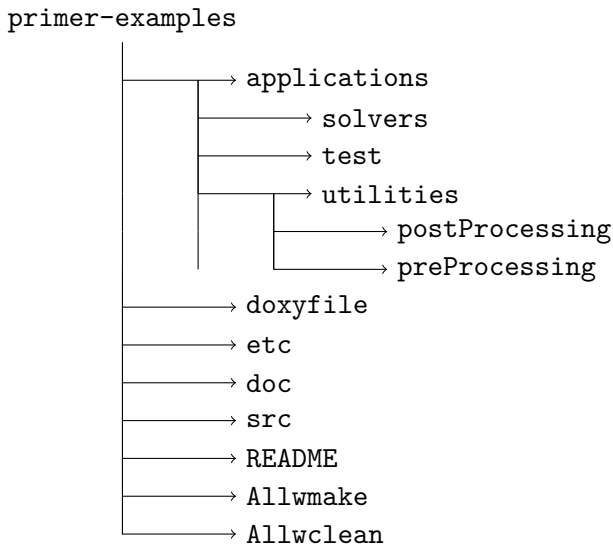
Figure 6.1: Directory structure of the example code repository.

also a fundamental way of indirectly documenting the code. To examine an example directory organization, consider the example code repository for the book and how it is organized.

As shown in figure 6.1, the `applications` directory is where the applications are stored. Inside the `applications` directory, following the OpenFOAM structure, the following subdirectories are present: `solvers`, `test` and `utilities`. The `etc` directory is used to configure the compilation of the code. The organization of the `src` folder is different for each library. As the classes extract and encapsulate common behavior between different abstractions, so does layered code organization separate different collections of class implementations into separated linkable libraries. Organizing and separating library categories reduces the size of the compiled application code when changes are introduced, and therefore speeds up the compilation.

The `README` file usually found in the top directory of the code repository is useful because it is the first file that is read when a user starts working with new code. A general description of the background of the project and its most important applications, as well as the up-to-date links to external documentation sources and forums, can usually be found there. Local documentation of the code can be generated using the `Doxygen`

documentation system, which uses the `doxyfile` to specify the files as well as the details involving the look and feel of the generated HTML documentation.

6.1.2 Automating installation

In addition to the directory organization from figure 6.1, an simplified, automated build process should be enabled. OpenFOAM uses its own build system called `wmake`, which makes use of various environmental variables to automatize the compilation and linking of the library and application code. The same approach can be applied to a custom code repository and is implemented for the provided example code repository. The `bashrc` configuration script for the example code repository looks like

```sh
#!/bin/sh

DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
export PRIMER_EXAMPLES=${DIR%%/etc*}
export PRIMER_EXAMPLES_SRC=$PRIMER_EXAMPLES/src
export PATH=$PRIMER_EXAMPLES_SRC/scripts:$PATH
```

The `bash` configuration script sets the path variable named $PRIMER_EX-AMPLES, which is the path variable to the main folder of the code repository. The PATH variable needs to be extended, since the example code repository contains scripts in OpenFOAM, that are located in `src/scripts`. Otherwise those scripts cannot be invoked from anywhere in the filesystem. The structure and configuration outlined here can be reused in another repository by using different variable for the project's path. The applications and libraries of the code repository rely on the variable $PRIMER_EXAMPLES to find the directories that hold the header files that are to be included before compilation.

The compilation scripts, `Allwmake` and `Allwclean` in the root directory are used to respectively compile and clean the project binaries. Additionally, similar scripts are placed in the `src` and `applications` sub-directories, making it possible to compile of only libraries or only applications. An example content of the `Allwmake` script for building libraries within the `src` directory is shown in in the script below:

```sh
#!/bin/sh
cd ${0%/*} || exit 1    # run from this directory
```

```
applicationName
         |
         |─────────────→ applicationName.C
         |──────────→ Make
                       |
                       |───────────→ files
                       └──────────→ options
```
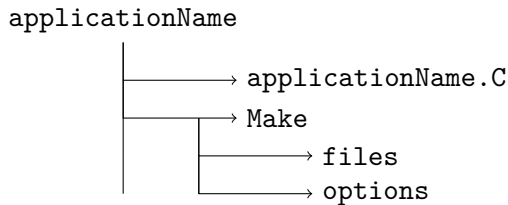
Figure 6.2: Application directory structure.

```
wmakeLnInclude .
wmake exampleLibrary
```

In the wmakeLnInclude script searches recursively through the current directory, finds all OpenFOAM source files, and creates symbolic links to these files in the src/lnInclude directory. This greatly simplifies the configuration of the building process, since all header files of a library are not scattered over various sub-directories; all header files required at compile time are linked into one location, lnInclude. As soon as the absolute path of the repository folder is defined ($PRIMER_EXAMPLES variable defined by etc/bashrc script), the inclusion of header files holding the class declarations is relying on the symbolic links of all source files stored in the src/lnInclude directory. The name of library to be compiled (in this case: exampleLibrary) is passed as an argument to the wmake script which ensures the building process will result in a dynamically linkable library.

Usually application code of OpenFOAM is stored in a directory named after the application. The contents of an example application directory are shown in figure 6.2. The applicationName.C is the source file of the application. Both, files and options files are used by the wmake build system to compile the application code. The Make/files contents are easily understood:

```
applicationName.C

EXE = $(FOAM_USER_APPBIN)/applicationName
```

The Make/files lists the *.C files that are to be compiled and the name and location of the binary file that will contain the compiled code. The application installation target directory is set to $FOAM_USER_APPBIN, in order not to pollute the primary OpenFOAM system application directory $FOAM_APPBIN with custom applications. Using $FOAM_USER_APP-

BIN stores the binaries in a folder that is a sub-folder of $HOME: this avoids requiring root rights to build custom projects on computers where a single OpenFOAM installation may be shard among all the users with non-root privileges, e.g., clusters. The Make/options contain all directories that contain declaration (*.H) files - so-called *include directories -* as well as directories (-L) that contain libraries (-l) our new application or or library links with:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(PRIMER_EXAMPLES_SRC)/lnInclude

EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lfiniteVolume \
    -lmeshTools \
    -lexampleLibrary
```

The options file shows that the custom application applicationName relies on the repository variable $PRIMER_EXAMPLES_SRC and the Open-FOAM generated lnInclude directory, to locate the required header files. Additionally, the application will link to the library exampleLibrary and use the needed functionality contained within it.

> INFO
>
> The key step in having a custom project directory structure is preparing the bashrc configuration script. Relying on the variables set by that script to locate header files and libraries of the custom project separates the custom project from the OpenFOAM platform.

This kind of configuration is a simple way of working and programming with custom code. The following list is a summary of steps in the workflow described above.

- source ./etc/bashrc to set the $PRIMER_EXAMPLES variable for the shell terminal,
- add source /path/to/code/directory/etc/bashrc to the startup script of the shell *if you want to permanently set the environment variable,*
- execute ./Allwmake in the top code directory to compile all libraries and applications,

- execute ./Allwclean in the top code directory for clean up of binaries,
- when you add a library e.g. libraryName to the repository, edit src/Allwmake and add wmake libraryName for compilation, as well as wclean libraryName in src/Allwclean for cleanup of binaries for the new library code,
- if you use standard directories in the example code repository for your applications, they will be compiled and cleaned without editing the compilation or cleanup scripts.

6.2 Debugging and profiling

The first and easiest method of debugging is adding Info statements to narrow down broken parts of the code. However, this approach in end effect takes much more time than learning how to use a debugger, such as gdb. The process of code profiling, however, is typically only applied to functioning programs with the goal of increasing computational efficiency.

> INFO
>
> Learn how to use a debugger: if OpenFOAM is running in parallel, there is no prescribed order of execution of the code in different MPI processes, which makes Info statements and their parallel counterparts Pout or Perr useless for debugging.

6.2.1 Debugging with GNU debugger (gdb)

Debugging code using GNU Debugger (gdb) is more beneficial when OpenFOAM has been compiled without any optimizations. Optimizations are performed by the compiler and generally do not interfere with the debug information generated for the compiled code. However, profiling and examining code that is not optimized by the compiler may provide more insight into hidden computational bottlenecks.

Compiler flags used for OpenFOAM are bundled up into groups of options which can be interchanged depending on the target configuration.

The compiler options are found in the `$WM_PROJECT_DIR/etc/bashrc` global configuration script and are listed below:

```
#- Optimised, debug, profiling:
#   WM_COMPILE_OPTION = Opt | Debug | Prof
export WM_COMPILE_OPTION=Opt
```

To enable working with OpenFOAM in debugging mode using gdb, the `$WM_COMPILE_OPTION` variable must be set in the following way:

```
export WM_COMPILE_OPTION=Debug
```

In order for this change to take effect, OpenFOAM must be re-compiled. Re-compilation is also necessary for all custom libraries and applications which require debugging. Please bare in mind that the execution time of code compiled with the debug option active is significantly longer.

> INFO
>
> Debugging with gdb is quite straightforward and there is plenty of information on working with gdb available on the official website `www.gnu.org/software/gdb/`.

Debugging the code can sometimes be simple for bugs such as a segmentation fault (accessing the wrong memory address) or a floating point exception (dividing by zero). Those errors in the execution of the code trigger system signals, such as `SIGFPE` (floating point exception signal) and `SIGSEV` (segmentation violation signal) and are trapped by the debugger. The debugger then allows the user to browse through the code in order to find the error, set break points, examine variable values and much more.

In order to show gdb in action, an example for debugging with gdb is covered in this section. This tutorial is provided in the sample code repository in the `ofprimer/applications/test/testDebugging` directory. The tutorial consists of a testing application that contains a function template used for computing a harmonic mean of a given field, over all time steps of the simulation. Because the harmonic mean involves computing $\frac{1}{x}$, where x is the field value, a floating point exception (`SIGFPE`) appears if the code is executed on a field which contains zero values. The function template is defined in the `fvc::` namespace and it behaves similar to the rest of the OpenFOAM operations. It does, however, have a slightly reduced functionality. For this reason, the function template

definition and declaration are packed together with a testing application which is not (and should *never* be) a standard practice.

The tutorial simulation case that can be used with this example is the case of book-cases/chapter4/rising-bubble-2D. Calling the test-Debugging application with the -field alpha.water argument within the risingBubble-2D case directory results with the following error:

```
?> testDebugging -field alpha1
(snipped header output)
Time = 0
#0  Foam::error::printStack(Foam::Ostream&) at ??:?
#1  Foam::sigFpe::sigHandler(int) at ??:?
#2  ? in /usr/lib/libpthread.so.0
#3  ? in $FOAM_USER_APPBIN/testDebugging
#4  ? in $FOAM_USER_APPBIN/testDebugging
#5  __libc_start_main in /usr/lib/libc.so.6
#6  ? in $FOAM_USER_APPBIN/testDebugging
Floating point exception (core dumped)
```

The $FOAM_USER_APPBIN variable is of course expanded to the respective path on your machine. The first step in debugging this problem is to start gdb, combined with testDebugging. All components of testDebugging must be compiled in debug mode, this also includes all libraries of interest as well as the OpenFOAM platform.

```
?> gdb testDebugging
```

This leads to the console version of gdb, which which is used to execute any programs which require debugging:

```
(gdb)
```

Within this debugging console of gdb any command can be executed for debugging purposes when prepended by the run command. For the testDebugging application, this means that the alpha1 field must also be passed as an additional argument:

```
(gdb) run -field alpha1
```

After the execution of the above command, the SIGFPE error appears again, but with much more details:

```
Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79                   resultField[I] = (2. / resultField[I]);
(gdb)
```

Due to the compilation of OpenFOAM and the application in `Debug` mode, `gdb` points directly to the line in the source code which is responsible for the `SIGFPE`. For the very basic example of the `testDebugging` application, analyzing the source code manually may lead to the same insight. With increasing algorithm complexity, it become less likely that a manual search for a bug will be successful. While inserting `Info` statements is the first method beginners use for debugging, in the end it consumes much more time, than it takes to learn a few basic `gdb` commands. The debugger, on the other hand, can: use information stored on the memory stack to step into functions, execute loops one iteration after another, change variable values, post break points in execution, condition the break points based on variable values, and many other advanced options you can find in the debugger documentation.

To review a basic workflow with `gdb`, the aforementioned error is assumed to occur inside a convoluted code-base. This code is located in a library with non-cohesive classes which are strongly coupled with fat interfaces and member functions that span hundreds of lines. In this situation, the programmer needs to examine the code above and below the signal line to get a hold on the context of execution. The debugger can display the path that the signal has taken: from the top-level call in the testing application, all the way down to the low level container that raises this error. This information can be obtained within `gdb`, by executing `frame` in the `gdb` console:

```
Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79                    resultField[I] = (1. / resultField[I]);
(gdb) frame
#0  0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79                    resultField[I] = (1. / resultField[I]);
```

Since the `testDebugging` example solely consists of the `main` function, a single stack frame is available, where the information regarding the function object code is stored. When multiple function calls are made, there are multiple frames available. The lowest frame of the `SIGSEV` signal usually leads somewhere to the basic OpenFOAM container `UList`. It is very unlikely, however, that the error is caused by `UList`, and much more likely that it is caused by the custom code. In this case, an addressing error in a container (which inherits from `UList`) is the reason for the error.

Choosing frame 0 and listing the source code with the `list` command, narrows down the location of the error:

```
(gdb) frame 0
(gdb) list
75 volumeField& resultField = resultTmp();
76
77 forAll (resultField, I)
78 {
79     // SIGFPE.
80     resultField[I] = (1. / resultField[I]);
```

Hence the culprit for the SIGFPE signal is line 80. To put a break point at line 80, the `break` command must be executed and `testDebugging` must be re-run:

```
(gdb) break 80
Breakpoint 1 at 0x4146cd: file testDebugging.C, line 80.
(gdb) run
The program being debugged has been started already.
Breakpoint 1, Foam::fvc::harmonicMean<double> (inputField=...) at
testDebugging.C:80
80                    resultField[I] = (1. / resultField[I]);
```

After having placed the break point in line 80, the variable `I` can be evaluated at this position:

```
(gdb) print I
$1 = 0
```

Printing `resultField[I]` shows that it's value is 0. In order to check the influence of different values of `resultField[I]`, they can be set manually using gdb:

```
(gdb) set resultField[I]=1
(gdb) c
Continuing.

Breakpoint 1,
Foam::fvc::harmonicMean<double> (inputField=...)
at testDebugging.C:80
80resultField[I] = (1. / resultField[I]);
(gdb) c
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in
Foam::fvc::harmonicMean<double> (inputField=...)
at testDebugging.C:80
80 resultField[I] = (1. / resultField[I]);
(gdb) print I
```

```
$2 = 1
(gdb)
```

For this simple example, it is obvious that at least two values of I lead to 0 for `resultField`. As, by definition, the field `resultField` must never be 0, the next step is to investigate if the field has been preprocessed properly.

INFO

The solution to problem is already included in the sources for this example - uncommenting the marked lines allow the application to run correctly.

6.2.2 Profiling

Code profiling using performance measurements is crucial for CFD software because CFD software must not only be accurate, robust, but also fast on a single CPU core and, for large problems, fast when it runs on many CPU cores.

Profiling the code with the `valgrind` profiling application is relatively straightforward. `valgrind` allows a developer to find computational bottlenecks in code as well as displays them graphically using call-graphs and similar diagrams. This information can then be used to concentrate efforts more effeciently when optimizing code for efficiency. Typically, performance is distributed using an estimated 90-10 or 80-20 rule, which means that 90 (80) % of the computation is usually performed by 10 (20) % of the code.

Similar to the previous section on debugging with gdb, the code needs to be compiled with the `Debug` compilation option. The high-level form of optimization is directly related to software design: choosing algorithms and data structures appropriately. Once the algorithms and data structures are chosen efficiently, there is a possibility of performing what is called low-level optimizations (e.g. loop unrolling).

Complexity is a parameter which is used to compute the efficiency of algorithms and data structures. It is denoted by the capital "O": e.g. $O(n \log n)$, where n stands for the number of elements operated on. It generally is not advisable to delve into low-level optimization when there

is an alternative algorithm available with a lower complexity. The same applies to data structure selection: choosing data structures carefully is an absolute must. More information of the container structures can be obtained from documentation on C++ data structures present in the Standard Template Library (STL) (see Josuttis [1] for details). OpenFOAM containers are not STL-based (although some of them provide STL compilant iterator interfaces), but are very much alike in how they function, for example:

`DynamicList` is similar to `std::vector`. Both have direct access of elements with $O(1)$ complexity and $O(1)$ insertion complexity at the end of the container, if the container size is smaller than its capacity. If the resulting size from an insertion operation is greater than the current capacity, the insertion complexity will be proportional to $O(n)$ (see [2]).

`DLList` is similar to `std::list`. The insertion complexity (anywhere in the container) is $O(1)$ and the access complexity is $O(n)$.

A lot of the algorithmic work present in OpenFOAM is related to the specific owner-neighbour addressing of the unstructured finite volume mesh, with the advantage of automatic parallelization of the algorithm code. For standard OpenFOAM library code, the choice of data structures falls between four main container families:

1. `List`
2. `DynamicList`
3. A linked list `*LList`
4. An associative map implemented as a `HashTable`

Before selecting a container for a specific algorithm, it is important to examine the container interface beforehand. Otherwise the selected container may not suit the needs of the algorithm properly, leading to longer execution times.

The example application `testProfiling` can be used to make this point for the `DynamicField` container using a very simple example. Try executing the profiling test application anywhere on your system (it doesn't require an execution within an OpenFOAM simulation case directory):

```
?> testProfiling -containerSize 1000000
```

The `-containerSize` option passed to the `testProfiling` testing application sets the number of elements that are appended to the two different

`DynamicField` objects: the first object is null-initialized, and the second one is initialized with the size that corresponds to the integer value passed to the application. Timing code has been added to the application for the part of the code that does the appending of elements at the end of the `DynamicField`. You can execute the application providing different values for the size of the container and see how the times differ between the initialized container and the container with an initialized size. The `DynamicField` has, as the `DynamicList` container, constant complexity in inserting elements at the end of the container *if the size of the container is smaller after insertion than the initial container capacity*, otherwise the complexity of an insertion at the end (appending) operation is linear with respect to the container size. This means it will have a larger impact on the efficiency of your code if the container sizes are large and even more if the container stores complex objects which then add additional burden of unnecessary $n - 1$ constructions for every element that is appended at the end of the container.

Once you have built the example source code repository in *Debug mode*, you can profile the `testProfiling` application using `valgrind`:

```
?> valgrind --tool=callgrind testProfiling -containerSize 1000000
```

Valgrind utilizes various tools such as catching cache misses, checking the programs for leaked memory, etc, but this is out of the scope of this book. More information on those topics can be found in the official valgrind documentation.

Executing the above command produces an output file called `callgrind.out.ID`, where ID is the process identification number. The output file may be opened with `kcachegrind`, which is an open source application used to visualize the output of valgrind. As you might have noticed yourself in the timing output produced by the `testProfiling` application, `valigrind` shows that around 62 % of the total execution time is spent for appending the elements to the un-initialized `DynamicField` object, and around 14% is spent on appending elements to pre-initialized `DynamicField` object.

This brings us to conclusion that although `DynamicList` and `DynamicField` are dynamic, there is a cost to pay for appending elements at the end of such a container once its capacity is exceeded. If our algorihm doesn't require direct access to elements, if it loops over the elements one after another, then using a heap-based linked-list might prove to be

a better choice. Allocating non-sequential memory blocks and pointer in-direction also cost CPU cycles, however, and as such it becomes difficult to know the proper choice of a container in advance. The answer to this problem comes in the form of generic programming, separating algo-rithms from containers in a careful and thoughtful way (it is not always possible), and profiling the code.

6.3 Using git to track an OpenFOAM project

Git is a distributed VCS which is very popular in the open source commu-nity. There are many benefits of using `git`: creating new versions is very straightforward which simplifies trying out new ideas, handling conflicts in files edited by multiple contributors is simplified by `git` web services (GitLab, GitHub, Bitbucket), each contributor holds a complete copy of the project and contributors can work without access to the internet be-cause an open connection to a central repository is not required while working, etc. There is ample of information on `git` available online[1]. A reasonable understanding of git is assumed for the remaining chapter. For working with OpenFOAM, it is sufficient to learn how to: clone a remote repository, create delete and merge versions (branches) locally and on the remote repository, push and pull from a remote repository, and create git tags (snapshots) of your project. A great branching model can be found on `envie.com`[2]. In the following we outline some use cases involving git and OpenFOAM.

> ⎛ INFO ⎞
> Although not covered here, learning the basic usage of the `git` version control system is necessary for OpenFOAM development.

Obtaining OpenFOAM using git

The OpenFOAM git repository can be found at `https://develop.openfoam.com/Development/openfoam/` and it can be cloned using

```
?> git clone https://develop.openfoam.com/Development/openfoam.git
```

---

[1] `http://git-scm.com/book`
[2] `http://nvie.com/posts/a-successful-git-branching-model`

After the command is finished, the entire history is available on your local machine and the history of the release can be investigated. The repository is cloned in the *main* (master) branch. OpenFOAM snapshots (releases) are periodically created as git tags

```
?> openfoam
?> git tag
OpenFOAM-v1601
OpenFOAM-v1606
OpenFOAM-v1612
OpenFOAM-v1706
OpenFOAM-v1712
OpenFOAM-v1806
OpenFOAM-v1812
OpenFOAM-v1812.200312
OpenFOAM-v1812.200417
OpenFOAM-v1906
OpenFOAM-v1906.191111
OpenFOAM-v1906.200312
OpenFOAM-v1906.200417
OpenFOAM-v1912
OpenFOAM-v1912.200129
OpenFOAM-v1912.200312
OpenFOAM-v1912.200403
OpenFOAM-v1912.200417
OpenFOAM-v1912.200506
OpenFOAM-v1912.200626
OpenFOAM-v2006
```

A release tag can be checked out with

```
?> git checkout OpenFOAM-v2006
```

and compiled. The history, or log, provides information about changes: the author, date and the message describing a change (a git commit).

```
?> git log
```

Actively developed features are available as feature branches

```
?> git branch -a
remotes/origin/cloud-function-objects-extension
remotes/origin/code-review.mol
remotes/origin/code-review.saf
remotes/origin/develop
remotes/origin/doc-utilities.kbc
remotes/origin/feature-GIS-tools
remotes/origin/feature-MPPIC-dynamicMesh
remotes/origin/feature-PatchFunction1-ACMI
remotes/origin/feature-dlLibrary-unloader
remotes/origin/feature-film-flux-function-object
remotes/origin/feature-generalizedNewtonian
remotes/origin/feature-generalizedNewtonian.orig
remotes/origin/feature-generalizedNewtonian.up1
```

```
remotes/origin/feature-liquidFilm
...
```

Should you decide to work directly within the main OpenFOAM structure, create an account on `develop.openfoam.com`, request membership to the project from one of the project maintainers and fork OpenFOAM. Once your feature is thoroughly tested, you can submit a merge request, so that your code is integrated into OpenFOAM.

Placing a standalone OpenFOAM project under version control

The most common way of using git for custom developments is to track each of your projects in a *seperate* repository that is unrelated to the main release. This simplifies the sharing of your code as it can be shared without the full OpenFOAM release. Assuming three projects should be put under version control: `projectA`, `projectB` and `projectC`. Each of these projects' directories has to be entered individually and a git repository has to be initialized in each of them:

```
?> cd projectA
?> git init
```

Each of those repositories is local and empty. Files have to be added to the repository manually. To prevent the `.deb` files and `lnInclude` directories, that are generated at compile time, from polluting the repository, a `.gitignore` file should be added to prevent them from appearing in the repository. This method enables you to share and port your code between different OpenFOAM versions more easily.

Developing in the main repository

Though developing directly in the OpenFOAM repository might look a little bit odd at first, it has a couple of advantages over putting the developments into separate repositories. If administrating a HPC cluster and installing a global OpenFOAM version for all users that should all use some custom developments, this way of developing might come in handy. Merging off the master branch and applying changes *only* to this branch is a fairly safe way to integrate custom developments into the OpenFOAM repository. This branch is then deployed to the HPC cluster. The `master` branch should be similar to `master` of the main repository

and any upstream changes to `master` should get merged into the local development branch, in order to stay up to date. The deployment itself can get automated further using *git hooks*. Migrating developments from one major release to the next may be more difficult in this configuration than with seperate and independent repositories, however.

Version control for simulation cases

As git can track any text files, it is not only limited to source code but can also deal with OpenFOAM cases. Some files and directories, such as the time step and processor directories, should not be tracked by git and must hence be excluded by an appropriate `.gitignore` file. This selection is already done by the `.gitignore` in the `ofbook-cases` project, which ignores the mesh and a lot of other files and directories that are not directly related to the case setup. A good application for tracking a case is when the influence of various parameters on the simulation is to be investigated. This renders copying the same case many times obsolete. Snapshots of versions of simulation cases used in a publication or a technical report can be created using git tags.

6.4 Installing OpenFOAM on an HPC cluster

INFO

Contact the system administrator of the HPC cluster to make sure you build OpenFOAM properly.

In this section some topics are considered when attempting to install OpenFOAM on a HPC cluster. When working on a remote cluster, a whole new system comprised of different hardware, software libraries, and compiler versions has to be expected. Due to these differences, every system will be different and may hold special challenges for installation. With that said, this section will try to outline some of the more generic issues that you may encounter on any particular system instead of covering system particulars. Moving forward it is assumed that the reader has some experience with executing a compiler and setting its configuration as well as using Linux system environment variables. This includes linking include folders and library binaries to a compilation.

Listing 2 Changes to `$WM_PROJECT_DIR/etc/bashrc`, that alter the compiler

```
# [WM_COMPILER_TYPE] - Compiler location:
# = system | ThirdParty
export WM_COMPILER_TYPE=system

# [WM_COMPILER] - Compiler:
# = Gcc | Gcc4[8-9] | Gcc5[1-5] | Gcc6[1-5] | Gcc7[1-4] | Gcc8[12] |
#   Clang | Clang3[7-9] | Clang[4-6]0 | Icc | Cray | Arm | Pgi
export WM_COMPILER=Gcc
```

It is advisable to use the MPI libraries that are available on an HPC cluster, because they are carefully built by the cluster administrators to maximise performance on that specific machine. If compiling against the OpenMPI library included in the stock `ThirdParty` folder that is packaged with OpenFOAM, the solver may still function but will be without considerable parallel optimizations and interconnect support. This could severely hinder parallel speed and scaling and is not recommended.

Without a fully function MPI compilation, the OpenFOAM `Pstream` library will not compile correctly. The `Pstream` library acts as an interface between the CFD library and the raw MPI functions, which is necessary for any parallel computations.

Each cluster will usually have an officially supported compiler for use on the system. This could be an open source compiler such as Gcc, but it may also be a pre-configured commercial C++ compiler such as Icc, offered by Intel®. To select the compiler for OpenFOAM, the main configuration file `$WM_PROJECT_DIR/etc/bashrc` must be adapted. The section of the file that concerns compiler selection is shown in listing 2. The first option, `Compiler location`, defines whether the system compiler is used or the compiler that came packaged with OpenFOAM in the `ThirdParty` directory is used by wmake in any compilation activities. In the case of an HPC installation, the `system` compiler should always be used. The other option will set which compiler to use. There are a lot of options available, but they all may not be installed on the system in question. For example, many Linux based compute clusters will have Gcc installed somewhere on the system, but the reccomended compiler may be a highly optimized and tuned version of Icc. In this case, the compiler should be defined as:

```
# [WM_COMPILER] - Compiler:
```

Listing 3 Changes to `$WM_PROJECT_DIR/etc/bashrc`, that alter the MPI configuration

```
# [WM_MPLIB] - MPI implementation:
# = SYSTEMOPENMPI | OPENMPI | SYSTEMMPI | MPI | MPICH | MPICH-GM |
#   HPMPI | CRAY-MPICH | FJMPI | QSMPI | SGIMPI | INTELMPI | USERMPI
# Also possible to use INTELMPI-xyz etc and define your own wmake rule
export WM_MPLIB=SYSTEMOPENMPI
```

```
# = Gcc | Gcc4[8-9] | Gcc5[1-5] | Gcc6[1-5] | Gcc7[1-4] | Gcc8[12] |
#   Clang | Clang3[7-9] | Clang[4-6]0 | Icc | Cray | Arm | Pgi
export WM_COMPILER=Icc
```

As with the compiler, `wmake` needs to be configured to build against the supported MPI implementation specific to the HPC system. This is done in the same `bashrc` file as compiler selection (see listing 3). Depending on which implementation is selected, OpenFOAM sets a different target location for `MPI_ARCH_PATH` and `MPI_HOME`, which is intended to be the location on the cluster file system where these mpi executibles, libraries, and header files exist. One of the most common issues with HPC installations is that this directory is not correctly set and the `Pstream` library doesn not compile correctly due to missing MPI libraires and headers.

In this case, the `WM_MPILIB=OPENMPI` setting will instruct `wmake` to use the OpenMPI version that was packaged with the `ThirdParty` directory. If `SYSTEMOPENMPI` is set instead, the script will look to the system to load the OpenMPI library. The script which defiens these environment variables depending on the selection, is located in `$WM_PROJECT_DIR/etc/-config/settings.sh`, under the `Communications library` subsection.

There the `MPI_ARCH_PATH` and `MPI_HOME` environment variables are set. Unfortunately, the way the variables are set may not be suitable for the specific configuration of the cluster. For example, choosing `HPMPI` as the MPI library, the script will attempt to load a specific directory (around line 463):

```
HPMPI)
    export FOAM_MPI=hpmpi
    export MPI_HOME=/opt/hpmpi
    export MPI_ARCH_PATH=$MPI_HOME

    _foamAddPath $MPI_ARCH_PATH/bin
```

Listing 4 Definition of a custom MPI version in `$HOME/.bashrc`

```
source ~/OpenFOAM/OpenFOAM-2.2.0/etc/bashrc
export MPI_ARCH_PATH=/opt/apps/ssmpi/1.3
export MPI_HOME=/opt/apps/ssmpi/1.3
```

Listing 5 Rules for `wmake`

```
?>  cat $WM_PROJECT_DIR/wmake/General/mplibOPENMPI

PFLAGS    = -DOMPI_SKIP_MPICXX
PINC      = -I$(MPI_ARCH_PATH)/include
PLIBS     = -L$(MPI_ARCH_PATH)/lib$(WM_COMPILER_LIB_ARCH)
            -L$(MPI_ARCH_PATH)/lib -lmpi
```

If the location of the `hpmpi` folder is not exactly `/opt/hpmpi`, `MPI_HOME` will not be set correctly and the error will cascade throughout the script and eventually disrupt the compilation. Thankfully, setting these variables manually is quite easy, assuming the location of the target MPI implementation in the system directory tree is known.

Lets say this cluster supports a specific, *fictional*, MPI implementation called Super-Scaling MPI, `ssmpi` for short. This fictional library, version 1.3, can be found within the directory tree `/opt/apps/ssmpi/1.3`. Because the script will not know how to configure `wmake` to link to this, the mpi environment variables have to be defined manually in `$HOME/.bashrc` (see listing 4). The variables are set accordingly *after* the OpenFOAM `bashrc` is sourced to ensure any incorrectly loaded values are overwritten. The importance of having this environment variable set is evident when looking at the `wmake` rules for setting MPI compiler flags (see listing 5). In the configuration file for compiling with OpenMPI, these variables are being used to set the paths to header and binary locations. If an installation is set up with the fictional SSMPI code, a new configuration file would be created, following this naming convention and populated with the appropriate file paths. In the end, all of this setup is to ensure that the `Pstream` library correctly compiles and links to the local MPI implementation. The `Pstream` compilation options, stored in `$WM_PROJECT_DIR/src/Pstream/mpi/Make/options`, contain `PFLAGS`, `PINC`, and `PLIBS` being fed directly to the compiler flags via `-I` and `-L`. This will be used at compilation time for header and library linking. Beyond interfacing with an MPI library, OpenFOAM is mostly self contained and has only a few other external library depen-

dencies. If significant trouble occurs during compilation, there is a high probability it occurred during configuration of these MPI libraries and compiler settings.

Unfortunately, any number of issues and errors could arise during compilation on any given system. Often times reaching a successful compile will require a user to draw upon personal experience gained from using compilers as well as working within different operating system environments. Luckily, with the growing usage base of OpenFOAM, many new systems are coming online with a local version pre-installed and configured. If a user has relatively little experience using compilers and requires an HPC cluster to be used with OpenFOAM, it may be a good idea to specifically look to systems with the configured package.

Further reading

[1]   Nicolai M. Josuttis. *The C++ standard library: a tutorial and reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2]   *Why is std::vector::insert complexity linear (instead of being constant)?* URL: http://stackoverflow.com/questions/25218880/ why-is-stdvectorinsert-complexity-linear-instead-of-being-constant (visited on 03/2016).

7

Turbulence modeling

This chapter covers the modeling of turbulence in OpenFOAM. The details on the physical and mathematical modeling are kept to a minimum. The reason for this decision lies in the fact that tubulence modeling is a large topic in itself, one beyond the scope of this book. For more in-depth discussions of turbulence modelling, the reader is referred to respective literature, such as Pope [11], Pozrikidis [12], Wilcox [16], and Lesieur [8].

7.1 Introduction

In general, there are four different approaches to modeling turbulence and it's effects. The main purpose of the turbulence models is to determine the Reynolds stresses, as they are unknown in the momentum equation (see chapter 1).

Depending on the model type, the method of calculating the Reynolds stresses can vary from relatively simple to very complex, which in turn leads to different requirements to the required computational effort and the computational grid. Figure 7.1 provides a compact overview over these models.

The most basic category of turbulence models are the Reynolds averaged Navier-Stokes (Reynolds Averaged Navier Stokes (RANS)) models, as all of models of this group work on the *temporal* fluctuation of the mean velocity, the *Reynolds averaging* (see [13, 8]). Models of this type can

> TIP
>
> The Reynolds stresses can be regarded as *"averaged momentum flow per unit area, and so comparable to a shear stress"* (see [1]) and they introduce additional unknowns to the flow equations. This leads to a *closure* problem, due to the nonlinearity of the Navier-Stokes equations, when they are averaged, which leads to the *Reynolds* equations (more details in [8]). To be able to solve this problem, more information must be provided. In this specific case, the Reynolds stresses must be modeled somehow, which is the point where the turbulence models come into play (see Pope [11]).
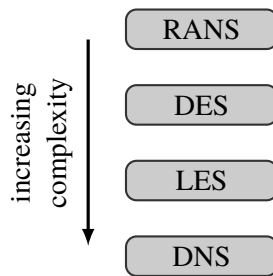
Figure 7.1: Relative computational complexity of various turbulence models and direct numerical simulations.

work on a comparatively coarse grid, as the turbulent fluctuations are not resolved geometrically, but modelled. If taken literally, these models are only suitable for steady state simulations, as no real turbulent fluctuations can be modeled. Prominent examples for such RANS models are the k-ϵ model (see [5, 6]), the k-ω model (see [15]) and k-ω-SST model (see [9, 10]). Implementations of these models in OpenFOAM are located in the Reynolds Averaged Simulation (RAS) model library.

The next group of models is called large eddy simulation (Large Eddy Simulation (LES)) and differ from RANS models therein that only small scale eddies are modeled. Large scale eddies are resolved spatially by the computational grid, which needs to be significantly finer to function correctly compared to RANS models. In terms of computational costs and efficiency, LES lies between RANS and Direct Numerical Simulation (DNS), as shown in figure 7.1. Pope [11] states that *"LES can be expected to be more accurate and reliable than Reynolds-stress models, for flows*

in which large-scale unsteadiness is significant".

DNS is based on solving the Navier-Stokes equations for all flow scales and does not deploy any turbulence modeling whatsoever. Implementation-wise this approach is the simplest one, but the required computational efforts are extremely high as all spatial and temporal time scales are fully resolved.

Turbulence modeling in OpenFOAM is generic, hence any model can be selected within each solver. This of course assumes that the solver supports turbulence modeling, which is the case for a majority of solvers. The generic implementation has the major benefit of being able to use RTS in conjunction with the turbulence models and not having to recompile the solver, when a different turbulence model should be used. All turbulence models can be found in `$FOAM_SRC/TurbulenceModels` and their respective implementation varies between compressible, incompressible and LES type models. In the following, only a small subset of the RANS models is covered, whereas LES, Detached Eddy Simulation (DES) and DNS models are neglected.

7.1.1 Wall functions

"At high Reynolds number, the viscous sublayer of a boundary layer is so thin that it is difficult to use enough grid points to resolve it" (see [3]). The wall functions rely on the *universal law of the wall*, which states that the velocity distribution very near to a wall is similar for almost all turbulent flows.

One of the most prominent parameterS when judging the applicability of wall functions is the dimensionless wall distance y^+, defined by Schlichting and Gersten [14] as:

$$y^+ = \frac{y u_\tau}{\nu} \tag{7.1}$$

With y representing the absolute distance from the wall and u_τ and ν denote the friction velocity and kinematic viscosity, respectively.

Pope [11] provides great insight into wall functions and why they are so important. The turbulence model need to account for the steep velocity profile at the wall and in relatively close proximity to the wall. This

is where wall functions come into play, which were first suggested by Launder and Spalding [7]. The idea is that additional boundary conditions are applied at some distance to the wall, to fulfill the log-law. Hence the additional equations introduced by the turbulence model are not solved close to the wall. Depending on the particular turbulence model used, different wall functions must be applied to the respective fields of the turbulence model. Meaning that a $k - \epsilon$ model requires different wall functions than a $k - \omega$ model.

WARNING

Especially when the flow suffers severe flow separation, RANS models are often unable to capture this separation properly. Hence such flow problems must be handled with care, if RANS models are used (see Pope [11], Wilcox [15], and Ferziger and Perić [3]).

Different turbulence models and their associated wall functions require different values of y^+ and hence different spatial resolution of the near wall area. The reader is referred to the particular literature for required y^+ values that the computational grid near the wall must achieve. Note that if the log-law region is resolved geometrically by the mesh, no wall functions need be applied. Depending on the type of simlation, such low y^+ values are either extremely hard to achive during the meshing process or even undesireable, as this significantly decreases the time step.

In OpenFOAM, wall functions are nothing else than ordinary boundary conditions that are applied to boundary patches of type `wall`, rather than the usual `patch`. If a wall function boundary condition is applied to a `patch` boundary type the the solver will complain with an error message on execution. As wall functions are quite similar to boundary conditions in terms of implementation, their design is not discussed explicitly in this chapter. Boundary conditions are discussed at great length in chapter 10.

7.2 Pre- and post-processing and boundary conditions

Depending on the choice of turbulence model, new fields are introduced to the simulation, which need to be solved as well. For the sake of

simplicity, the $k-\omega$ turbulence model is selected as an example. There are basically two types of boundary conditions that can be used to model the parameters of the turbulence model at the boundaries: standard conditions and turbulence specific conditions.

Standard boundary conditions such as `fixedValue` or `inletOutlet` boundary conditions can be used if the particular inflow values are known to the user. These values, namely turbulent kinetic energy k and the specific dissipation rate ω can be calculated manually from either the turbulence intensity I or the mixing length L (see [4]):

$$k = \frac{3}{2}\left(|\mathbf{U}_{\text{in}}|\,I\right)^2 \tag{7.2}$$

$$\omega = \rho\frac{k}{\mu}\left(\frac{\mu_t}{\mu}\right)^{-1} \tag{7.3}$$

$$\omega = \frac{\sqrt{k}}{C_\mu^{\frac{1}{4}}L} \tag{7.4}$$

Here the turbulence intensity should be selected as $I \in [0.01; 0.1]$ and for a freestream flow, $I = 0.05$ is a common choice. The turbulent viscosity ratio $\frac{\mu_t}{\mu}$ at inflow boundaries is assumed to be quite low and usually selected as $1 \leq \frac{\mu_t}{\mu} \leq 10$, as discussed by Fluent [4]. Equation (7.4) is commonly used, when the mixing length is known, whereas equation (7.3) can be used in the other cases, in a straight forward fashion. When these values are determined they can simply be assigned to the respective boundaries.

Turbulence specific boundary conditions can be used for some cases, which in turn implement some of the above equations, which can then be used for the initialisation of the inflow. The first of these special inflow boundary conditions is `turbulentIntensi-tyKineticEnergyInlet`, which initialises k according to equation (7.2). Applying this boundary condition to a boundary `INLET`, leads to the code listed in listing 6.

Additionally, the inlet for ω based on the turbulent mixing length initialization (equation (7.4)), can be directly defined using `turbu-lentMixingLengthFrequencyInletFvPatchScalarField`. Similar to `turbulentIntensityKineticEnergyInlet`, this boundary condition just requires some additional parameters and then computes ω based on equation (7.4). The required dictionary for the

---

Listing 6 Example of `turbulentIntensityKineticEnergyInlet`

```
INLET
{
    type    turbulentIntensityKineticEnergyInlet;
    I   0.05;
}
```

Listing 7 Example of `turbulentMixingLengthFrequencyInlet`

```
INLET
{
    type        turbulentMixingLengthFrequencyInlet;
    L   0.005;
}
```

---

boundary named INLET is shown in listing Here, C_μ is read directly from the selected turbulence model and need not be specified again.

For other fields introduced by other turbulence models, there are equivalent boundary conditions contained in the OpenFOAM framework.

7.2.1 Pre-processing

Some other usefull preprocessing applications include `boxTurb` and `applyBoundaryLayer`. `boxTurb` can be used to initialise a non-uniform and turbulent-appearing velocity field, which is convenient for cases where turbulent effects play a major role and must be present from the start of the simulation. The resulting velocity field still suffices the continuity equation and is hence divergence free.

The development of a near-wall flow can be simplified and its convergence improved by using `applyBoundaryLayer`. It calculates the boundary layer based on the $\frac{1}{7}$-th power-law (see [4, 2]) and the velocity field is adjusted accordingly.

This tool provides two custom commandline parameters, `Cbl`. `Cbl` calculates the boundary layer thickness as the product of its argument and the mean distance to the wall. Optionally the turbulent viscosity field ν_t can be stored to disk, which is not required to be present by the turbulence models.

7.2.2 Post-processing

In conjunction to the previously described preprocessing tools, there are various postprocessing applications in OpenFOAM. After each simulation that employs turbulence modelling, the y^+ values need to be checked. For this purpose, there are yPlusRAS and yPlusLES. Each of which can be used for RANS or LES simulations, respectively. Their working principle is quite similar and is not to be discussed. To calculate the particular y^+ values, the respective command must be invoked in the simulation case of question. By default, y^+ is calculated for each available time directory of the case. This can be limited to specific ones by passing the -times parameter with appropriate arguments or simply the -latestTime option. The (shortened) output looks like the following:

```
?> simpleFoam -postProcess -func yPlus -latestTime
Reading field p

Reading field U

Reading/calculating face flux field phi

Selecting incompressible transport model Newtonian
Selecting turbulence model type RAS
Selecting RAS turbulence model kOmegaSST
Selecting patchDistMethod meshWave
[...]
No finite volume options present
yPlus yPlus write:
    writing field yPlus
    patch FOIL y+ : min = 62.8783, max = 151.871, average = 122.018
```

As can be seen from the above listing, not only the minimum, maximum and average y^+ values for the patches of type wall are printed to the screen.

Sometimes the Reynolds stresses R must be calculated and written to a field, for postprocessing purposes. The command line tool R is tailored to do exactly this, with no additional arguments required and executed in a similar manner as the yPlus post processing function as shown.

```
?> simpleFoam -postProcess -func R -latestTime
```

7.3 Class design

The design of turbulence models is quite similar to that of the transport model classes and is hence not repeated in this section. Transport models are covered in chapter 11. Rather than being able only to access an individual turbulence model directly, the models are nested into sub-categories. As a result, both the sub-category of turbulence models as well as each individual turbulence model can be selected using Runtime Selection (RTS). This leads to the `RASModel` and `LESModel` type of sub-categories, each of them containing separate model implementations.

Further reading

[1] H.D. Baehr and K. Stephan. *Heat and Mass Transfer*. Springer, 2011.

[2] Lawrence J. De Chant. "The venerable 1/7th power law turbulent velocity profile: a classical nonlinear boundary value problem solution and its relationship to stochastic processes". In: *Applied Mathematics and Computation* 161.2 (2005), pp. 463–474.

[3] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer, 2002.

[4] Fluent. *Fluent 6.2 User Guide*. Fluent Inc. Centerra Resource Park, 10 Cavendish Court, Lebanon, NH 03766, USA, 2005.

[5] W.P Jones and B.E Launder. "The prediction of laminarization with a two-equation model of turbulence". In: *International Journal of Heat and Mass Transfer* 15.2 (1972), pp. 301–314.

[6] B.E. Launder and B.I. Sharma. "Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc". In: *Letters in Heat and Mass Transfer* 1.2 (1974), pp. 131–137.

[7] B.E. Launder and D.B. Spalding. *Mathematical Models of Turbulence*. Academic Press, 1972.

[8] M. Lesieur. *Turbulence in Fluids*. Fluid Mechanics and Its Applications. Springer, 2008.

[9] F. R. Menter. "Zonal two-equation $k - \omega$ turbulence models for aerodynamic flows". In: *AIAA Journal* (1993), p. 2906.

[10] F. R. Menter. "Two-equation eddy-viscosity turbulence models for engineering applications". In: *AIAA Journal* 32.8 (1994), pp. 1598–1605.

[11] S. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[12] C. Pozrikidis. *Introduction to Theoretical and Computational Fluid Dynamics*. 2nd ed. Oxford University Press, 2011.

[13] O. Reynolds. "On the Dynamical Theory of Incompressible Viscous Fluids and the Determination of the Criterion". In: *Philosophical Transactions of the Royal Society of London. A* 186 (1895), pp. 123–164.

[14] Hermann Schlichting and Klaus Gersten. *Boundary-Layer Theory*. 8rd rev. ed. Berlin: Springer, 2001.

[15] D. C. Wilcox. "Re-assessment of the scale-determining equation for advanced turbulence models". In: *American Institute of Aeronautics and Astronautics Journal* 26 (1988).

[16] D. C. Wilcox. *Turbulence Modeling for CFD*. D C W Industries, 1998.

8

Pre- and post-processing applications

There are many ways to pre- and post-processing simulations OpenFOAM, programming new solver applications or numerical algorithms may require the user to develop new pre- and post-processing applications

Before considering the development of a new pre- or post-processing application, one should make sure the required algorithm is not available in OpenFOAM.

For example, a user-friendly pre-processing application `funkySetFields` was developed by Bernhard Gschaider as a part of the `swak4foam`project. This application can be used to initialize OpenFOAM fields using algebraic expressions. The expression parser in `funkySetFields` extracts arithmetic and differential operations from user-defined expressions and evaluates them using OpenFOAM numerical libraries. If a required functionality is not available in OpenFOAM, any of its sub-modules, or related projects, it is likely simpler to extend an existing application or library with new features than to program and test the new application from scratch.

In this case, a good practice is to find an existing application with similar functionality and modify it to fit the required task. As covered in chapter 6, the `Doxygen` generated HTML documentation can help identify and locate those parts of the OpenFOAM framework that can be used to build the new application.

Listing 8 Creating a new OpenFOAM application using the `foamNewApp` script.

```
?> foamNewApp myApp
Creating application code directory myApp
Creating Make subdirectory
```

8.1 Code Generation Scripts

A set of Linux shell scripts are available that help with creating "skeleton" source code files: application, class, class templates, and build files (used by the `wmake` build system). When creating a simple pre- or post-processing application, only a single source code file is required (e.g. `myApp.C`). Following the OpenFOAM naming convention, it is stored in the directory named in the same way as the application.

To begin programming a new application from scratch the `foamNewApp` script can be used as shown in listing 8. The `foamNewApp` utility will set the target location for storage of compiled binaries to `$FOAM_USER_APPBIN` in the file `Make/files`. This setting will ensure the installation process will copy the compiled binary files into a separate user specific folder as opposed the directories usually reserved for binary files of the applications distributed with OpenFOAM. Populating the OpenFOAM binary folders is a bad practice because it pollutes the directories reserved for the applications distributed with OpenFOAM. Sometimes such compilation is necessary. For example, if a system is used that has OpenFOAM installed in a directory that requires root user access rights, and applications are to be made available to all system users. To start working on the new application, the file `myApp.C` is to be edited and afterwards compiled with the OpenFOAM build script `wmake` called within the directory `myApp`:

```
?> wmake
```

The `Make/options` file holds the list of directories where the declarations are stored, and possibly definitions in cases when class/function templates are used. The directories that are to be searched by the `wmake` build system are set with the `-I` option:

```
-I$(LIB_SRC)/finiteVolume/lnInclude
```

In cases when the `-I` option does not use a relative path to a folder, environmental variables are used, such as `$(LIB_SRC)` in the example above. Additionally, a list of directories that hold precompiled dynamic libraries that are linked with the application is defined. The directories that contain the libraries are appended to the build options with the option `-L`:

```
-L$(LIBRARY_VARIABLE)/lib
```

In this example, `LIBRARY_VARIABLE` is an environmental variable which stores the path to the package `lib` folder. This is the location where loadable library binary files are stored. When an application is composed from additional libraries, the `Make/options` file needs to list the libraries to be linked against:

```
-lusedLibrary
```

Here the `usedLibrary` library will be linked against previously compiled application at runtime.

The steps described above are application specific so they will be described in more detail in the following sections. More information about libraries, linking, and the build process can be found in any book about programming in the Linux environment and on the Internet.

8.2 Custom pre-processing applications

Custom pre-processing applications may involve preparing initial conditions in ways that are not already available in OpenFOAM, preparing parallel simulation execution on a HPC cluster, or setting up parameter variations. In this section, we focus on parallel execution and parameter variations.

A surprising number of tasks can often be executed and automated using various applications chained together via shell or Python-based scripts. Good practice in OpenFOAM is to prepare a so-called `Allrun` script in the simulation case folder that executed pre-processing scripts necessary to prepare the simulation.

This section covers two examples with different complexity. In the first example, a simple shell script calls OpenFOAM executables using existing pre-processing applications. The second example covers using the

PyFoam library to program pre-processing applications for generating parameter variations. Bernhard Gschaider primarily develops the PyFoam project: a set of libraries and executable programs written in the `Python` programming language for straightforward parametrization and analysis of OpenFOAM simulations.

8.2.1 Parallel application execution

Consider a situation where a large number of simulations require the use of a high-performance cluster. For this example, all simulation cases will be decomposed into different sub-domains and stored in a directory named `simulations`. The `simulations` directory is to be placed as a sub-directory of the `$FOAM_RUN` directory. Instead of manually starting each simulation, it is preferable to automate this process as much as possible. For this purpose, a shell script can be programmed using the following steps:

1. Retrieve the solver name from `controlDict`.
2. Retrieve the number of sub-domains from `decomposeParDict`.
3. Execute the `mpirun` command with the correct number of processors.

In this example, the script is named `Allparrun` and is stored in the `simulations` directory. As a first step, a few lines should be added to the beginning of the script. Similar code is found in most of the tutorials that are distributed along with the release of OpenFOAM:

```
#!/bin/bash
cd ${0%/*} || exit 1

source $WM_PROJECT_DIR/bin/tools/RunFunctions
application=`getApplication`
```

The `getApplication` function is an existing bash function in Open-FOAM and fits in nicely for this example as the name of the application will be the name of the solver in this case.

Next, all case sub-directories in the `simulations` directory must be located. It is assumed that those directories are all proper OpenFOAM simulation cases. As a consequence, an OpenFOAM simulation will be started in each sub-directory of the `simulations` directory. The `find` command is used to look specifically for directories that can be found

---

Listing 9 Looping over OpenFOAM cases, using bash

```bash
for d in `find $FOAM_RUN/simulations -type d -maxdepth 1`
do
    # This part will be programmed at a later point in this section.
done
```

---

in $FOAM_RUN/simulations (see listing 9). The above lines look for any directory that is located directly (not recursively) in the $FOAM_RUN directory so that the same actions can be performed on each of these cases later on. The find command can be easily enhanced by e.g. filtering for certain directory names using wildcards, but for the sake of simplicity it is assumed in this example that the simulations directory solely contains OpenFOAM cases.

The next step is to retrieve the number of sub-domains from system/-controlDict and store this in a variable for later use. For this purpose a new function is introduced into the Allparrun script called nProcs which returns the number of sub-domains:

```bash
function nProcs
{
    n=`grep "numberOfSubdomains" system/decomposeParDict \
        | sed "s/numberOfSubdomains\s//g" \
        | sed "s/;//g"`
    echo $n
}
```

The function calls the grep program to filter for the line in system/decomposeParDict that contains numberOfSubdomains. In this line the 'numberOfSubdomains' string is deleted, any whitespace between this keyword and the actual numerical value is removed, and the tailing semicolon is deleted. The result of this chained command is stored in the bash variable n. To return it, echo is used.

Running any OpenFOAM solver or utility in parallel is done using the mpirun command. For this example, this means that two variables need to be passed over to mpirun as option arguments: the solver name, and the number of processors. The completed Allparrun script is shown in the listing 10.

For the most simple case, where a job queuing and submission system is not available, you'll only have to provide a machine file to mpirun. The machine file contains either the IP addresses or the host names of the computing nodes on the HPC cluster network. Refer to the respective

Listing 10 Final `Alparrun` script used for simple automated execution of a number of simulations.

```bash
#!/bin/bash
cd ${0%/*} || exit 1

source $WM_PROJECT_DIR/bin/tools/RunFunctions

function nProcs
{
    n=`grep "numberOfSubdomains" system/decomposeParDict \
        | sed "s/numberOfSubdomains\s//g" \
        | sed "s/;//g"`
    echo $n
}

PWD=`pwd`

for d in `find $FOAM_RUN/simulations -type d -maxdepth 1`
do
    # Jump into the current case directory
    cd $d
    n=`nProcs`
    application=`getApplication`

    runApplication decomposePar

    # Depending on the environment of your HPC, this must be
    # adjusted accordingly. Remember to provide a proper
    # machine file, otherwise all jobs will be started on the
    # master node, which is undesirable in any way.
    mpirun -np $n $application -parallel > log &

    # Jump back
    cd $PWD
done
```

> **INFO**
>
> Keep in mind that in this case the call to `mpirun` is starting all jobs as *local* jobs on the machine. In order to initiate it on an HPC cluster and use its compute nodes, other steps are necessary that depend on the specific cluster configuration and are therefore omitted here.

users manuals provided by your HPC cluster administration team for further information on this topic.

8.2.2 Parameter variation

Parameter variation consists of pre-processing of many simulations with varying physical and discretization parameters, which requires automation. To simulate parameter studies in OpenFOAM, shell scripts, Python scripts, or available OpenFOAM utility applications can be used to clone simulations and adjust their parameters in the text-based input (configuration) files. Since OpenFOAM is built for the command-line use, OpenFOAM is better suited for automation than other GUI-based CFD platforms.

This example covers the parameterization of a two-dimensional NACA0012 airfoil. The mesh generation is done with the `blockMesh` application, and the mesh is mirrored using the `mirrorMesh` application. The parametric study investigates the influence of the angle of attack α on the lift and the drag force exerted by the flowing air on the airfoil. The parameter study consists of 15 simulations with a varying angle of attack from $\alpha = 0°$ to $\alpha = 15°$, resulting in a rough distribution of the lift force w.r.t. the angle of attack. The initial lift force distribution is then refined at the peak in the graph by automatically creating new simulations after the first 15 converge. Without automation, extensive parameter variation in CFD projects can quickly become a laborious and error-prone activity.

Using a shell script for automation is possible but cumbersome because of the amount of source code required to manipulate OpenFOAM configuration files effectively and how shell scripts handle alphanumeric calculation.

The Python scripting language is straightforward to use alternative to shell scripts, which further simplifies the processing and visualization of simulation results. Additionally, many Python-based libraries already exist with functionalities designed to support the process of numerical simulations, such as interpolation methods, root-finding methods, data processing, and analysis libraries, machine learning, and similar. A Python project exists that already implements many functionalities required for automating parameter variation of OpenFOAM simulations: PyFoam.

The PyFoam project consists of different Python modules that can manipulate OpenFOAM data. In addition to this library part, PyFoam already offers many different ready-to-use applications that simplify different tasks.

Tab-completion in the command line shows the list of many available PyFoam applications; here, we comment only on a few of them: `py-FoamClearCase.py` clears the OpenFOAM simulation case directory of previously computed results, `pyFoamPlotWatcher.py` renders in real-time diagrams of residuals from the log file of an OpenFOAM solver, `pyFoamRunParameterVariation.py` creates a parameter study from a Cartesian product of parameter vector arrays and a template case.

The PyFoam executables build upon the PyFoam library, in the same way as the following example that covers the programming of a PyFoam pre-processing application. Using PyFoam libraries in Python scripts enables customized manipulation of OpenFOAM cases. Additionally, it is possible to execute OpenFOAM executables from within Python in a straightforward manner using the `subprocess.call` Python submodule. For this example, we assume some basic background knowledge of Python and that PyFoam is available on the machine.

INFO

PyFoam can be installed using the `pip` package installer for Python with `sudo pip install PyFoam` , or without root privileges, with `pip install --user PyFoam`.

Although it is possible to manipulate CFD results from OpenFOAM using PyFoam, it is essential to understand that manipulating large data arrays (field values) will run somewhat slower than in a compiled language such as C++. This example focuses on the preparation of the parameter variation study and does not manipulate fields resulting from the simulations with OpenFOAM.

Parameter studies in OpenFOAM consist of many simulation folders that have different configuration files. The simulation folders are all generated from the same input (template) simulation folder. In this example, we will re-use the simulation from chapter 4: `ofprimer/cases/chapter04/naca` as the template case for the parameter variation.

WARNING

It is essential that the template simulation case is set up properly, as all parametrized cases are created from it.

Before programming a pre-processing application for a parameter study

that is more complex, it makes sense to learn what can already be achieved using pyFoamRunParameterVariation.py. The PyFoam applications have extensive documentation that can be displayed using the --help option in the command line. Executing pyFoamRunParameter-Variation.py --help shows an extensive list of options that are not all covered here.

It is good practice to separate the generation of the parameter study directory structure from meshing and simulation execution in real-world scenarios. This way, mesh generation and simulation run in parallel across the sub-domains of decomposed problems and the study's parameters. In other words, we first generate the simulation folder structure in serial, as this only involves the manipulation of text files. Meshing then runs in parallel for every parameter vector, followed by their respective simulations, using workload managers on a HPC resource. To achieve this kind of setup, pyFoamRunParameterVariation.py requires a specific set of options wrapped in a Python script called create_naca_study.py to facilitate re-use in an effective workflow.

> INFO
>
> The script create_naca_study.py can be found in the ofprimer/ cases/chapter04 folder alongside the naca template simulation case.

The options passed to pyFoamRunParameterVariation.py in create_ naca_study.py are shown in listing 11:

--every-variant-one-case-execution ensures a OpenFOAM simulation case is generated for each parameter vector in the variation,
--create-database stores the relation between the simulation ID and the paramater vector (crucial for reproducing results),
--no-mesh-create prevents mesh generation,
--no-execute-solver prevents solver (simulation) execution,
--cloned-case-prefix is used to differentiate this parameter study from other studies.

It is useful to store "configurations" of such calls to pyFoamRunParame-terVariation.py in scripts like create_naca_study.py: this simplifies the simulation workflow and ensures easy reproduction of results.

The pyFoamRunParameterVariation.py application requires as input a template simulation case and a file containing varied parameters. The

---

Listing 11 The `create_naca_study.py` script.

```python
#!/usr/bin/env python

import argparse
import sys
from subprocess import call

parser = argparse.ArgumentParser(description='Runs \
        pyFoamRunParameterVariation.py to create simulation folders \
        without generating the mesh.')

parser.add_argument('--study_name', dest="study_name", \
                type=str, required=True, \
                help='Name of the parameter study.')

parser.add_argument('--parameter_file', dest="parameter_file", \
                type=str, required=True,
                help='PyFoam .parameter file')

parser.add_argument('--template_case', dest="template_case", \
                type=str, required=True,
                help='Parameter study template case.')

args = parser.parse_args()

if __name__ == '__main__':

    args = parser.parse_args(sys.argv[1:])

    prefix = args.study_name + "_" + args.parameter_file

    call_args = ["pyFoamRunParameterVariation.py",
                "--every-variant-one-case-execution",
                "--create-database",
                "--no-mesh-create",
                "--no-server-process",
                "--no-execute-solver",
                "--no-case-setup",
                "--cloned-case-prefix=%s" % prefix,
                args.template_case,
                args.parameter_file]

    call(call_args)
```

---

listing 12 shows the parameters for the `naca` case, where only the kinematic viscosity is varied. The values listed in listing 12 and stored in the `naca.parameter` file should be replaced somehow in the `constant/transportProperties` file of the `naca` case. To ensure PyFoam replaces the values of ν in `constant/transportProperties`, the file is copied into `constant/transportProperties.parameter`, and nu is

---

Listing 12 The `naca.parameter` file.

```
values
{
   solver ( simpleFoam );

   NU
   (
      1e-06 2e-06 3e-06 4e-06
   );
}
```

---

modified as shown in listing 13.

Listing 13 Parameterization of dictionary files using `PyFoam`.

```
...

transportModel  Newtonian;

rho            rho [ 1 -3 0 0 0 0 0 ] 1000;

nu             nu [0 2 -1 0 0 0 0] @!NU!@;

...
```

Running

```
?> ./create_naca_study.py --study_name my_study \
   --template_case naca  --parameter_file naca.parameter
```

generates from the `naca` simulation case template four OpenFOAM new simulation cases with four viscosity values listed in listing 13,

```
?> ls -d my_study*
my_study_naca.parameter_00000_naca
my_study_naca.parameter_00001_naca
my_study_naca.parameter_00002_naca
my_study_naca.parameter_00003_naca
```

If multiple parameter vectors are defined in listing 13, `PyFoam` constructs a Cartesian product of those vectors and their respective simulation folders. Large parameter variations require a relation between the unique simulation IDs (00000, 00001, . . .) and parameter vectors. This is easily obtained with

```
?> pyFoamRunParameterVariation.py --list-variations \
naca naca.parameter

===============================
4 variations with 1 parameters
===============================


==================
Listing variations
==================


Variation 0 : {'NU': 1e-06}
Variation 1 : {'NU': 2e-06}
Variation 2 : {'NU': 3e-06}
Variation 3 : {'NU': 4e-06}
```

Having covered the variation of existing OpenFOAM simulation parameters with PyFoam's `pyFoamRunParameterVariation.py`, the next steps involve the mesh generation and simulation.

The next example covers programming of parameter variation scripts using sub-modules from the `PyFoam` library. This task becomes relevant when the generation of parameter variations goes beyond the Cartesian product of parameter vectors, or more complex calculations determine the parameter values.

Before going into details about writing a custom `PyFoam` script, the necessary background information is provided about the simulation. A symmetric mesh is generated using `blockMesh`, that consist of three blocks as outlined in figure 8.1. The grading of the blocks is adjusted so that there is no noticeable change in cell sizes from block 1 to 2 and from 2 to 3. To resolve the viscous boundary layer along the airfoil fairly small cells are generated near the NACA profile.

All patches are named according to labeling shown in figure 8.1. The vertex numbers are indicated by the two numbers separated by a dash. All edges on the `FOIL` patch that are not oriented in the normal direction of the figure are shaped using the `polyLine` of `blockMesh` with the station being calculated according to the function that defines any two digit NACA profile. The same goes for the vertices of the blue `INLET` patch - but rather than using the two digit NACA polymonial, a quarter circle is used. This makes the mesh look more visually appealing and removes any boundary condition issues between the `INLET` patch and the `SIDE` patch.
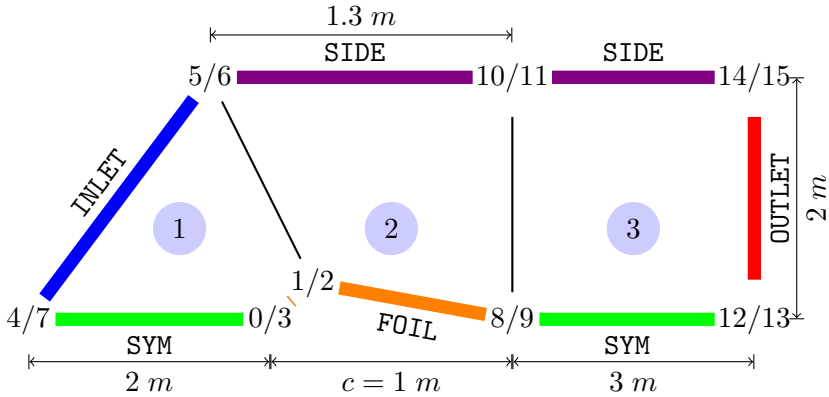
Figure 8.1: Overview of the NACA mesh

Listing 14 Imported PyFoam packages

```
from PyFoam.RunDictionary.SolutionDirectory import SolutionDirectory
from PyFoam.RunDictionary.ParsedParameterFile import ParsedParameterFile
from PyFoam.Basics.DataStructures import Vector
from numpy import linspace,sin,cos,array
from os import path, getcwd
```

All dimensions of the computational domain as well as the NACA profile itself can be obtained from figure 8.1. The employed solver is `simple-Foam` and will be executed in serial mode.

> **INFO**
>
> The `cases/chapter04/naca` case is set up only for mesh generation and parameter variation, not for obtaining physical results.

Programming the Python script begins with importing necessary modules, shown in listing 14. Three `PyFoam` submodules are important for the example covered in this section:

SolutionDirectory : handles an entire case and provides easy access to its data as well as methods to clone the case to a different directory,

ParsedParameterFile : reads any OpenFOAM dictionary and provides access to it as if it is a Python dictionary,

Vector : handles manipulation of vectors.

Listing 15 Imported Python packages

```python
# Commandline arguments and paths
import os
import sys
import argparse

# Parameter variation
import numpy as np
from numpy import linspace,sin,cos,array,pi
import yaml
```

Listing 16 Velocity vector rotation

```python
def rot_matrix(angle_deg):
    angle_rad = angle_deg * pi / 180.
    return np.array([[cos(angle_rad),0,-sin(angle_rad)],
                     [0,1,0],
                     [sin(angle_rad), 0, cos(angle_rad)]])

def rot_vector(vec, angle_deg):
    return np.dot(rot_matrix(angle_deg), vec)
```

The modules for parsing command-line arguments and file paths and performing computations in the custom parameter variation (cf. listing 15) directly follow the `PyFoam` modules from listing 14. The custom parameter study will vary the angle of attack of the NACA0012 airfoil, which requires a rotation of a vector around the $-y$ coordinate axis in this example case, implemented by functions from listing 16. The parameter variation script controls the study via command-line options. Alternatively, one could control the parameter variation similarly as `pyFoamRunParameterVariation.py`: using a configuration file, in a standardized format such as CSV, JSON, or YAML. The command-line arguments are parsed by code shown in listing 17. The main function of the parameter variation script is shown in listing 18 and it is outlined in detail with comments. The core idea is to read a template simulation case, then clone the case into multiple simulation cases, identified by unique identifiers (IDs) that correspond to a parameter variation. Each parameter variation in this example case consists of a single scalar value: the angle of attack α. Generally, a parameter variation is identified by a n-dimensional parameter vector in a parameter space, where n corresponds to the number of parameters varied in a simulation. In this case, varying the angle of attack requires the rotation of the velocity field around the $-y$ coordinate axis direction. Velocity file is read using `PyFoam`, and from it access is gained

Listing 17 Parsing command line arguments

```
rsing command line options
parser = argparse.ArgumentParser(description='Generates simulation cases \
                         for parameter study using PyFoam.')

parser.add_argument('--template_case',
                dest="template_case", type=str,
                help='OpenFOAM template case.', required=True)

parser.add_argument('--study_name', dest="study_name", type=str,
                help='Name of the parameter study.', required=True)

parser.add_argument('--alpha_min', dest="alpha_min", type=float,
                help='Minimal angle of attack in degrees.',
                required=True)

parser.add_argument('--alpha_max', dest="alpha_max", type=float,
                help='Maximal angle of attack in degrees.',
                required=True)

parser.add_argument('--n_alphas', dest="n_alphas", type=int,
                help='Number of angles between alpha_min and alpha_max.',
                required=True)

args = parser.parse_args()
```

to the inlet boundary condition and the internal velocity field. Both the inlet boundary condition and the initial internal values are then set such that the specific angle of attack is achieved. Once the velocity field is initialized like this, its file is written. The dictionary that relates the ID of the unique parameter variation with the parameter vector is then saved as a YAML file. This ID - parameter vector mapping is also saved by `pyFoamRunParameterVariation.py` in a `PyFoam`-specific format. Here, YAML is used as a standard format that is easily manipulated in Python. Although a single parameter is varied in this small example script, the example shows how the main sub-modules of `PyFoam` can be used together with standard Python modules to extend parameter variation in OpenFOAM to having full control over the variation. A next step would be, for example, to reduce the full Cartesian product of parameter vectors by applying conditions that disable parameter vectors that do not make sense.

The script is available in the code repository, as `cases/chapter04/parametricStudy.py`. Calling it to vary the angle of attack betweek $0°$ and $10°$ with 10 steps, i.e.

Listing 18 Main function for parameter variation.

```python
if __name__ == '__main__':

    args = parser.parse_args(sys.argv[1:])

    # Distribute angles of attack
    angles = np.linspace(args.alpha_min, args.alpha_max, args.n_alphas)

    # Initialize the template case
    template_case = SolutionDirectory(args.template_case)
    # Initialize the parameter dictionary
    param_dict = {}
    # For every variation
    for variation_id,alpha in enumerate(angles):
        # Add variation-d : parameter vector to parameter dictionary.
        param_dict[variation_id] = {"ALPHA" : float(alpha)}
        # Clone the template case into a variation ID case.
        name = "%s-%08d" % (args.study_name, variation_id)
        case_name = os.path.join(os.path.curdir, name)
        cloned_case = template_case.cloneCase(case_name)
        # Read the velocity field file.
        u_file_path = os.path.join(cloned_case.name,"0", "U")
        u_file = ParsedParameterFile(u_file_path)
        # Read the velocity inlet boundary condition and internal field.
        u_inlet = u_file["boundaryField"]["INLET"]["value"]
        u_internal = u_file["internalField"]
        # Rotate the velocity clockwise by alpha around (-y).
        u_numpy = np.array(u_inlet[1])
        u_numpy = rot_vector(u_numpy, alpha)
        # Set inlet and internal velocity.
        u_inlet.setUniform(Vector(u_numpy[0],u_numpy[1],u_numpy[2]))
        u_internal.setUniform(Vector(u_numpy[0],u_numpy[1],u_numpy[2]))
        u_file.writeFile()

    # Store variation ID : study parameters into a YAML file.
    with open(args.study_name + '.yml', 'w') as param_file:
        yaml.dump(param_dict, param_file, default_flow_style=False)
```

```
?> ./parametricStudy.py --study_name angle-of-attack \
--template_case naca --alpha_min 0 \
--alpha_max 10 --n_alpha 10
```

results in

```
?> ls
   angle-of-attack-00000001
   angle-of-attack-00000002
   angle-of-attack-00000003
   ...
   angle-of-attack.yml
```

The `angle-of-attack.yml` contains the ID-parameter vector mapping

```
0:
   ALPHA: 0.0
1:
   ALPHA: 1.1111111111111112
2:
   ALPHA: 2.2222222222222223
...
```

Although the simulation parameters of the `naca` case do not ensure correctness of physical results in this example, the streamlines of two extreme variants of the parameter study are shown in figure 8.2.
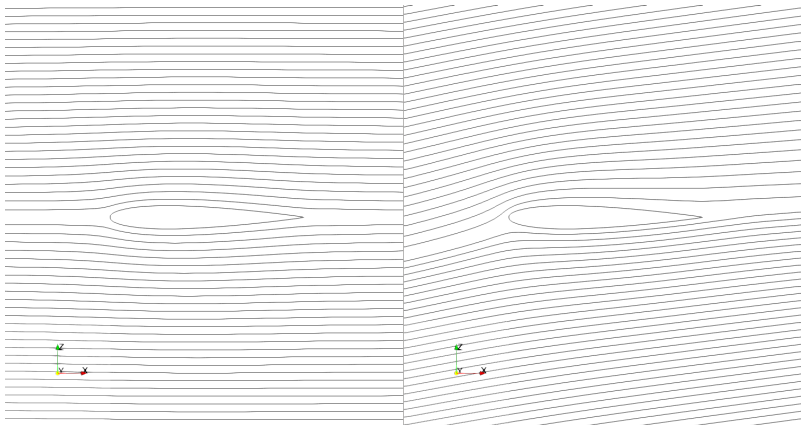

Figure 8.2: Streamlines for $\alpha = 0$ (`angle-of-attack-00000000`), and $\alpha = 10$ (`angle-of-attack-00000009`).

8.3 Available post-processing applications

Post-processing of simulation results in OpenFOAM is based on *function objects*. Chapter 12 covers the usage, design and implementation of function objects in OpenFOAM, this section covers only their application to post-processing. As the name suggests, post-processing is generally performed on simulation data after the simulation has finished. Function objects replace the post-processing of simulation data from completed simulations with active processing of simulation results, generated as the simulation runs. Active processing of simulation results is very beneficial because it provides an active overview of the simulation, making it possible to stop the simulation early in case of errors, which, in turn, saves both computational resources and development time. Function objects in OpenFOAM can be selected by the user at runtime (when the simulation starts) based on their configuration in the `controlDict` dictionary file, so there is no need to maintain a multitude of different post-processing applications. A single post-processing application `postProcess` is available that re-uses (selects, configures, and runs) function objects to perform post-processing of generated simulation data. An example of function objects used for active processing is the case

`$FOAM_TUTORIALS/compressible/sonicFoam/RAS/nacaAirfoil`

In the `controlDict` file of this case, the force coefficients of the airfoil are defined by the function object entry shown in listing 19. When the simulation starts, the `forceCoeffs` function object will process the fields required to calculate force coefficients on the `wall_4` boundary patch. Resulting data is written in `postProcessing/forces/0/coefficient.dat`, and can be analyzed as the simulation runs, preferably using a Jupyter notebook. Using Jupyter notebooks to view and analyze active processing results in OpenFOAM simulations, generated by OpenFOAM function objects brings two important benefits. A Jupyter notebook can be refreshed in a web browser as the simulation runs, making it possible to update the data analysis and examine simulations as they run. Additionally, Jupyter notebooks can be started securely on a remote machine (an HPC cluster) and viewed in a web browser on a PC/laptop. Function objects that support this, can be executed after the simulation has completed, by running the `postProcess` application in the simulation case folder. Information about different post-processing function objects and

Listing 19 Example usage of a function object for computing force coefficients in `$FOAM_TUTORIALS/compressible/sonicFoam/RAS/-nacaAirfoil`.

```
functions
{
    forces
    {
        type           forceCoeffs;
        libs           (forces);
        writeControl   writeTime;

        patches
        (
            wall_4
        );

        rhoInf     1;

        CofR       (0 0 0);
        liftDir    (-0.239733 0.970839 0);
        dragDir    (0.970839 0.239733 0);
        pitchAxis  (0 0 1);
        magUInf    618.022;
        lRef       1;
        Aref       1;
    }
}
```

their usage is available in the Extended Code Guide so this section only provides this brief overview.

8.4 Custom post-processing applications

Custom post-processing applications are programmed when no similar application or function object exists in OpenFOAM that performs the required calculation.

This section covers the programming of an example post-processing application that computes a rising bubble area and velocity. The rising bubble in this example is approximated by a level-set of a field used to track the bubble. The rising bubble is simulated with the `interIsoFoam` solver [4, 5]. The `interIsoFoam` solver implements a variant of the unstructured Piecewise-Linear Interface Calculation (PLIC) Volume-of-Fluid method [3] (cf. [2] for a recent review). In the `interIsoFoam` solver, there is a

choice between the reconstructing an iso-surface from the signed distance (ψ_{RDF}) [5] and the volume fraction ψ_α. Since the difference in the re-constructed bubble is negligible, $\psi_\alpha \equiv \alpha$ is used as a level-set field in this example. The bubble is therefore a level-set

$$\Sigma(t) = \{\mathbf{x} : \tilde{\alpha}_c(\mathbf{x}) = 0.5\}, \quad \mathbf{x} \in \mathbb{R}^3. \tag{8.1}$$

The $\{\alpha_c\}_{c \in C}$ volume fractions are associated to the cell centers, and they are approximated as $\tilde{\alpha}_c := \tilde{\alpha}_c(\mathbf{x})$ at a point $\mathbf{x}$ for reconstructing the iso-surface. The iso-surface reconstruction algorithm in OpenFOAM implements the regularized marching tetrahedra algorithm [6], which uses linear approximation for $\tilde{\alpha}_c(x)$ along a mesh-edge intersected by $\Sigma(t)$. Inversed-Distance Weighted (IDW) interpolation is used to compute cell-corner volume fraction values from cell-centered volume fractions $\{\alpha_c\}_{c \in C}$.

The task of the iso-surface algorithm is to reconstruct the surface of a bubble as a triangular mesh, from an iso-surface field (volume fraction in this case). In this example, the area of the bubble is the area of the triangular iso-surface mesh, and the bubble centroid is likewise the centroid of the iso-surface mesh. The bubble velocity is calculated from the temporal evolution of the bubble centroid.

> INFO
>
> This chapter relies on the `cases/chapter04/risingBubble2D` simulation case from the `ofprimer` repository.

The example application is named `isoSurfaceBubbleCalculator` and it is available in the source code repository as well, in the

    applications/utilities/postProcessing/

folder of the code repository. The name of the library used by the `isoSurfaceBubbleCalculator` is `bubbleCalc` and its source code is placed in the

    src/bubbleCalc

folder.

The example starts with the implementation of the `bubbleCalc` library, followed by the `isoSurfaceBubbleCalculator` post-processing application.

Listing 20 Constructor declaration of the `isoSurfacePoint` class.

```
isoSurfacePoint
(
    const volScalarField& cellValues,
    const scalarField& pointValues,
    const scalar iso,
    const isoSurfaceParams& params = isoSurfaceParams(),
    const bitSet& ignoreCells = bitSet()
);
```

Modeling a rising bubble with an iso-surface is a prototype example of adding functionality to an existing class. In the case of the iso-surface, this class is called `isoSurfacePoint`, and its sources are located in the

```
$FOAM_SRC/sampling/surface/isoSurface
```

folder. Of course, the class that requires an extension is often unknown, but it can be found by searching the Extended Code Guide for classes with similar functionality,

The interface of the `isoSurfacePoint` class shows that the class constructor is responsible for reconstructing the iso-surface triangle mesh, as shown in listing 20

The `isoSurfacePoint` constructor prescribes following requirements on the rising bubble:

- an additional field should be calculated at cell corner-points (`pointValues`),
- a member function is not already available for area, centroid or rise-velocity calculation.

The above items impose an additional functionality to the `isoSurface-Point` class. Implementing the new calculations with global functions and data in a post-processing application is possible, but this makes it impossible to re-use them in another setting: a function object or another application.

Therefore, the iso-surface calculation is to be encapsulated into a new class named `isoBubble` and compiled into a dynamically loadable shared library `bubbleCalc`. Programming the post-processing code into a shared library makes it possible to share the functionality between different applications and other libraries that may require it. An example of this could be computing the signed distance between two bubbles.

The `isoSurfacePoint` requires an additional `pointField` associated to cell-corner points to compute the iso-surface, and the bubble reconstruction relies only on the cell-centered `volScalarField`, so the calculation of the `pointField` is encapsulated into a small class shown in listing 21. Inversed-Distance Weighted interpolation is used in this example, and can

Listing 21 The `isoPointFieldCalc` encapsulates the calculation of the cell-corner iso-surface field from the cell-centered iso-surface field.

```
class isoPointFieldCalc
{
    // Computed point iso-field.
    scalarField field_;

    public:

        isoPointFieldCalc(const volScalarField& isoField)
        {
            this->calcPointField(isoField);
        }

        virtual void calcPointField(const volScalarField& isoField)
        {
            volPointInterpolation pointInterpolation (isoField.mesh());
            field_ = pointInterpolation.interpolate(isoField)();
        }

        virtual ~isoPointFieldCalc() = default;

        const scalarField& field() const
        {
            return field_;
        }
};
```

be replaced with something else.

Listing 22 shows the private class interface of `isoBubble`. Starting at the class declaration, note that the `isoBubble` inherits from the `regIOob-ject` class. OpenFOAM encapsulates the IO operations for objects in the `regIOobject` class, which is registered to the *object registry*. The object registry is a class that calls the write member functions of the object in question when the a writing operation is executed in the application code. It also a global object in the program that issues write requests to all objects that are subscribed to it. This kind of design represents a common Object Oriented Design (OOD) pattern called "Observer Pattern" ([1]), and it fits very well with CFD solvers because of the following reasons (among others):

Listing 22 Private `isoBubble` class interface.

```
class isoBubble
    :
        public regIOobject
{
    isoPointFieldCalc isoPointField_;

    // Bubble geometry described with an iso-surface mesh.
    isoSurfacePoint bubbleSurf_;

    // Number of zeros that are prepended to the timeIndex()
    // for written VTK files.
    label timeIndexPad_;

    // Output format.
    word outputFormat_;

    fileName paddWithZeros(const fileName& input) const;
```

- an object-oriented CFD application may use dozens of objects so calling write for each one causes severe code bloat,
- the objects are not necessarily written out at every time step - the write frequency follows user-defined rules that are read from a dictionary file and applied by the registry.

Instead of having multiple write calls in the solvers (which would look something like `object.write()`) the class `Foam::Time` serves as the *Observer* or *Object Registry*, and dispatches the call to write to all subjects (registered objects).

A call `runTime.write()` causes the `Foam::Time` registry to loop over a list of pointers to the registered objects ("Subjects" in the Observer pattern) and forwards the write call to each registered object at the end of each time step. The actual writing occurs only if the time step in question is the time step designated for the output. For example, a user can prescribe the output to take place every N time steps.

In order to write the `isoBubble` object using the write controls available in OpenFOAM (e.g. every 5 time steps, or every 0.01 seconds), the `isoBubble` class inherits publicly from the `regIOobject` class. The `padWithZeros` private member function of the `isoBubble` expands the file names so that they are made readable as a time-sequence by the paraView visualization application. The actual output in the specified VTK format is then delegated from the `isoBubble` to the `isoSurface`

class. The files are written in an *instance* directory under the name given
as a sub-argument of the `IOobject` constructor argument.

The private attributes of the `isoBubble` class are

- `isoPointFieldCalc` that calculates the cell-corner iso-surface field
  values from cell-centered iso-surface field values,
- `isoSurfacePoint` that performs the iso-surface reconstruction and
  stores the surface mesh data,
- data nad functions required for a formatted output of the iso-surface.

The point field is not available by default in OpenFOAM simulations -
OpenFOAM uses cell-centered fields as dependent variables in a mathe-
matical model. The non-existent empty constructor of `isoSurfacePoint`
and a constructor that takes a cell-centered (vol)field as an argument, rule
out inheriting from `isoSurfacePoint`. Instead, composition is used, and
the `isoPointFieldCalc` object computes the point field required by
`isoSurfacePoint` constructor.

With the computed `isoPointField` and all the necessary arguments
passed to the constructor, the `isoSurfacePoint` class takes care of the
actual reconstruction of the iso-surface mesh, as shown in listing 23.

Listing 23 The `isoBubble` construtor reconstructs the iso-surface

```
isoBubble::isoBubble
(
    const IOobject& io,
    const volScalarField& isoField,
    scalar isoValue,
    bool isTime ,
    label timeIndexPad,
    word outputFormat,
    bool regularize
)
    :
        regIOobject(io, isTime),
        isoPointField_(isoField),
        bubbleSurf_(isoField, isoPointField_.field(), isoValue),
        timeIndexPad_(timeIndexPad),
        outputFormat_(outputFormat)
{}
```

The member functions of `isoSurfacePoint` can be used to perform the
center and area calculations for the bubble. We compute the area of the
bubble as the sum of the magnitudes of the iso-surface face area normal

vectors:

$$A_b = \sum_f \|\mathbf{S}_f\|. \tag{8.2}$$

The bubble center is calculated as the algebraic average of all the iso-surface mesh points:

$$\mathbf{C}_b = \frac{1}{N} \sum_N \mathbf{x}_N, \tag{8.3}$$

where N is the number of points of the iso-surface mesh. The computations are performed by the `isoBubble::area` and `isoBubble::center` member functions shown in listing 24. Having re-used the `isoSurface`

Listing 24 Iso-bubble area and center point calculation.

```
scalar isoBubble::area() const
{
    scalar area = 0;

    const pointField& points = bubblePtr_->points();
    const List<labelledTri>& faces = bubblePtr_->localFaces();

    forAll (faces, I)
    {
        area += mag(faces[I].normal(points));
    }

    return area;
}

vector isoBubble::center() const
{
    const pointField& points = bubblePtr_->points();

    vector bubbleCenter (0,0,0);

    forAll(points, I)
    {
        bubbleCenter += points[I];
    }

    return bubbleCenter / bubblePtr_->nPoints();
}
```

---

class and encapsulating our IO operations, as well as the computation of the bubble area and center, we can now easily write applications and

---

Listing 25 File `files` of the `bubbleCalc` library.

```
isoBubble.C

LIB = $(FOAM_USER_LIBBIN)/libbubbleCalc
```

Listing 26 File `options` of the `bubbleCalc` library.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/triSurface/lnInclude \
    -I$(LIB_SRC)/surfMesh/lnInclude \
    -I$(LIB_SRC)/surfMesh/MeshedSurfaceProxy/ \
    -I$(LIB_SRC)/sampling/lnInclude

EXE_LIBS = \
    -lmeshTools \
    -lfiniteVolume \
    -ltriSurface \
    -lsampling
```

---

instantiate as many bubble objects as we like. The `isoBubble` class is prepared for alternative computations, and can be extended with other bubble-related calculations.

In order to easily share the implementation of the `isoBubbble` with others (people or client programs), the `isoBubble` implementation is set to be a part of a library, namely the `bubbleCalc` library. Listing the `$PRIMER_EXAMPLES_SRC/bubbleCalc` directory shows the library files:

```
?> ls
isoBubble.C  isoBubble.dep  isoBubble.H  lnInclude  Make
```

The library build specifications are given in the `Make` folder, in files `options` and `files`. The file `files` lists the files that are compiled into the library object code The linker later uses this to bind to the function calls declared in the included header files and called in our application code. This file also specifies where the compiled library will be installed, in this case, and as recommended for user-defined libraries, the library is installed in the folder `$FOAM_USER_LIBBIN` to avoid polluting the library install directory of the OpenFOAM system.

The `options` file lists all the necessary paths to directories that hold the included header files used in `isoBubble`, as well as paths to the directories where the library code is installed. The pre-compiled library code allows us to extend existing code without always having to re-compile everything. The library is compiled with the OpenFOAM build system `wmake`, by issuing the command `wmake libso` within the `bubbleCalc` directory.

With the compiled library, the `isoBubbble` class can be used in the example post-processing utility `isoSurfaceBubbleCalculator`.

> **INFO**
>
> The example post-processing utility `isoSurfaceBubbleCalculator` is avalable in the code repository, in the `applications/utilities/` `postProcessing` folder.

First, the declaration of the `isoBubble` class must be included for the library to be used in an application. Additionally, to process existing simulation results, the time step selector is used. The corresponding code snippet is shown in listing 27. Next, a command line option is assinged to

---

Listing 27 Including `isoBubble` and `timeSelector` for iso-surface bubble calculation.

```
#include "fvCFD.H"
#include "timeSelector.H"
#include "isoBubble.H"

using namespace bookExamples;
```

---

the application, followed by the initialization of the root case, simulation time and the mesh, as shown in listing 28. The part of the application responsible for actual calculation is shown in listing 29

Inheriting from the `regIOobject` forces the client code to initialize the `isoBubble` object with an `IOobject`. The parameters of the `IOobject` constructor are defined as follows:

- `"bubble"` - name of the object (in our case, the name of the file we are using to save the iso-surface data),
- `"bubble"` - instance, or the directory where the object is stored (in this example same as the name of the file),

Listing 28 Parsing the `isoField` command line option and initalization.

```cpp
int main(int argc, char *argv[])
{
    argList::addOption
    (
        "isoField",
        "Name of the field whose level set is the bubble surface."
    );

    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"

    if (!args.found("isoField"))
    {
        FatalErrorInFunction
            << "Provide the name of the iso-surface field."
            << "Use '-help' for more information."
            << abort(FatalError);
    }
```

- `runTime` - object registry that the object registeres to - we want the IO operations for the `isoBubble` to be regulated by the simulation time,
- `IOobject::` - tokens (public enumerations) that define the runtime write/read operations for the `IOobject`,
- `inputField` - tracked iso-field.

The `timeSelector` class is then used to find the time step (iteration) directories among the files and folders in the simulation case directories. Within each iteration, the iso-field is read, the `isoBubble` object is reconstructed, the area and the center of the bubble are computed, and the bubble iso-surface mesh is written to the disk. In this sense, the `isoBubble` class is behaving in the same manner of any other Open-FOAM class.

Running the simulation by calling the `Allrun` script initializes the bubble, runs the `interIsoFoam` solver and calls

```
>  isoSurfaceBubbleCalculator -isoField alpha.air
```

to calculate the bubble iso surface. The bubble surface is stored as a series of "vtk" files in the `bubble` directory. The physical time, the bubble area and the coordinates of the bubble centroid are stored in `bubble.csv`.

Listing 29 Post-processing an iso-surface bubble.

```cpp
Foam::instantList timeDirs =
    Foam::timeSelector::select0(runTime, args);

OFstream bubbleFile("bubble.csv");
bubbleFile << "TIME,AREA,CENTER_X,CENTER_Y,CENTER_Z\n";
forAll(timeDirs, timeI)
{
    runTime.setTime(timeDirs[timeI], timeI);

    Info<< "Time = " << runTime.timeName() << endl;

    volScalarField isoField
    (
        IOobject
        (
            args.get<word>("isoField"),
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        ),
        mesh
    );

    isoBubble bubble
    (
        IOobject
        (
            "bubble",
            "bubble",
            runTime,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        isoField
    );

    const auto area = bubble.area();
    const auto center = bubble.center();

    Info << "Bubble area = " << bubble.area() << endl;
    Info << "Bubble center = " << bubble.center() << endl;

    bubbleFile << runTime.timeOutputValue() << ","
        << area << ","
        << center[0] << "," << center[1] << ","  << center[2] << "\n";

    bubble.write();
}
```

Visualization of the iso-surface in figure 8.3 shows some differences

between the 2D iso-surface algorithm in OpenFOAM and the one in ParaView (mentioned in the `isoSurface.H` file).
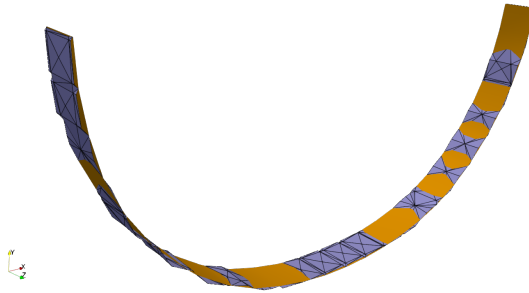

Figure 8.3: OpenFOAM and ParaView iso-surfaces for $\alpha = 0.5$, at the final time step.

The CSV output of the `isoSurfaceBubbleCalculator` in `bubble.csv` is processed in a Jupyter notebook `iso-bubble.ipynb` using the Pandas Python Data Analysis Library. The Python pandas code is listed in listing 30 This small example of using `pandas` is only here to point the reader to the pandas library: pandas can do much more powerful data processing than what is demonstrated in this example. Still, two advantages are visible that improve the understanding of data processing code: reading CSV files is trivial, columns are accessed by their names, calculation of derivatives is very straightforward. In addition to these simple advantages, the strongest benefit of `pandas` in the CFD context is its ability to easily store and manipulate results from parameter studies using `pandas.MultiIndex` with `pandas.DataFrame`.

The distribution of the bubble area over time is shown in figure 8.4. The area is almost halved when the bubble rises up to and spreads at the top wall of the solution domain. Figure 8.5 shows the distribution of the Y-coordinate of the bubble centroid calculated with equation (8.3). The distribution looks sufficiently smooth to the naked eye. However, using finite differences to compute the $Y-$ component of the rising velocity of the bubble from the Y-coordinate of the bubble's centroid in equation (8.3) is prone to error. This catastrophic loss in accuracy is confirmed by figure 8.6. Exercises suggested below improve the evaluation of the bubble rising velocity in the example post-processing calculation.

Listing 30 Python pandas code for processing the iso-surface bubble data.

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams["figure.dpi"] = 200
rcParams["text.usetex"] = True

bubble_csv = pd.read_csv("bubble.csv")
plt.plot(bubble_csv["TIME"], bubble_csv["AREA"])
plt.xlabel("Time in $s$")
plt.ylabel("Area in $m^2$")
plt.savefig("iso-bubble-area.png")

plt.plot(bubble_csv["TIME"], bubble_csv["CENTER_Y"])
plt.ylabel("Y-coordinate of the bubble center")
plt.xlabel("Time in $s$")
plt.savefig("iso-bubble-center-y.png")

y_velocity = bubble_csv["CENTER_Y"].diff() / bubble_csv["TIME"].diff()
plt.plot(bubble_csv["TIME"], y_velocity)
plt.ylabel("Bubble y-velocity in $m/s$")
plt.xlabel("Time in $s$")
plt.savefig("iso-bubble-center-y-velocity.png")
```
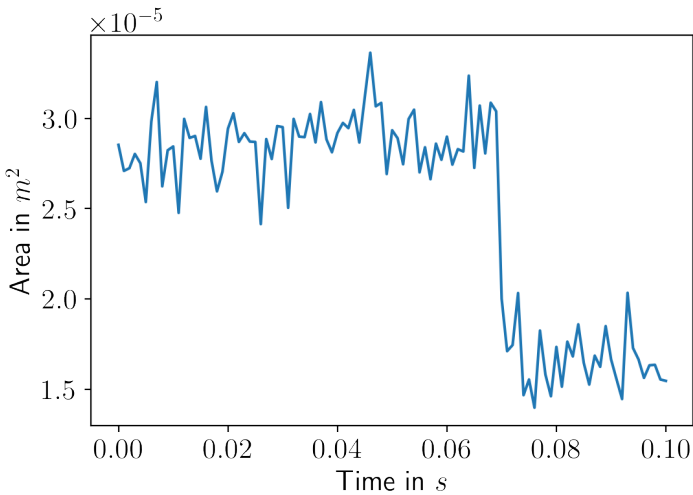

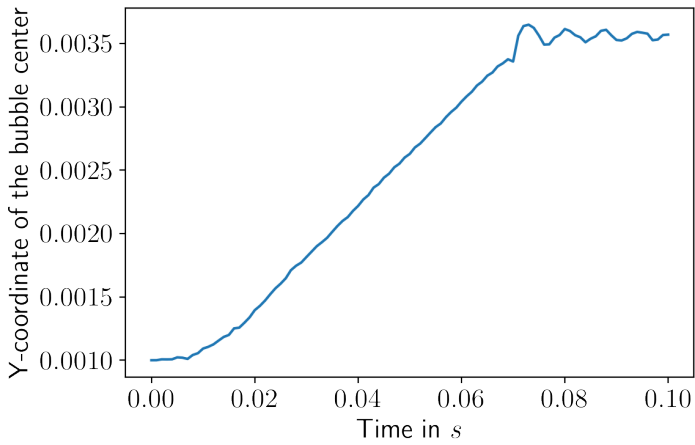Figure 8.4: Bubble area distribution over time.
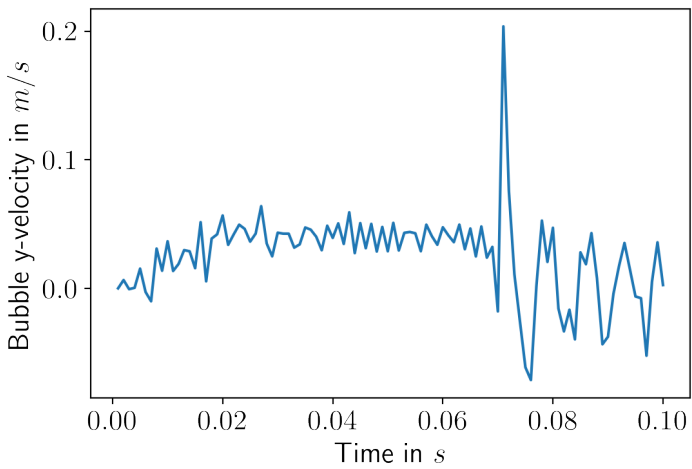
Figure 8.5: Y-coordinate of the bubble centroid.

Figure 8.6: Bubble Y-velocity.

EXERCISE

What happens if the rising velocity is computed with a more accurate time finite differencing scheme? Extend the `isoBubble` to compute the bubble velocity as the average velocity evaluated at centers of triangles of the bubble iso-surface? Hint: use `interpolationCellPoint` to interpolate velocities at triangle centers.

EXERCISE

Write another post-processing application that computes the bubble velocity using only the `alpha.air` and `U` fields. What are the differences in the calculated velocity, and where do these differences originate from?

Further reading

[1] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[2] Tomislav Marić, Douglas B Kothe, and Dieter Bothe. "Unstructured un-split geometrical Volume-of-Fluid methods–A review". In: *Journal of Computational Physics* 420 (2020), p. 109695.

[3] William J Rider and Douglas B Kothe. "Reconstructing volume tracking". In: *Journal of computational physics* 141.2 (1998), pp. 112–152.

[4] Johan Roenby, Henrik Bredmose, and Hrvoje Jasak. "A Computational Method for Sharp Interface Advection". In: *R. Soc. Open Sci.* 3.11 (2016). DOI: 10.1098/rsos.160405. URL: http://arxiv.org/abs/1601.05392.

[5] Henning Scheufler and Johan Roenby. "Accurate and efficient surface reconstruction from volume fraction data on general meshes". In: *J. Comput. Phys.* 383 (Apr. 2019), pp. 1–23. ISSN: 10902716. DOI: 10.1016/j.jcp.2019.01.009. arXiv: 1801.05382. URL: https://www.sciencedirect.com/science/article/pii/S0021999119300269.

[6] Graham M Treece, Richard W Prager, and Andrew H Gee. "Regularised marching tetrahedra: improved iso-surface extraction". In: *Computers & Graphics* 23.4 (1999), pp. 583–598.

9

Solver Customization

9.1 Solver Design

A solver application (solver) is not structurally different from any other OpenFOAM application: it uses algorithms implemented in different Open-FOAM libraries. Instead of arithmetically processing fields or manipulating the mesh somehow, a solver approximates the solution of Partial Differential Equations (PDEs) that model a physical process. A brief overview of the FVM used for PDE discretization in OpenFOAM is presented in chapter 1. OpenFOAM solvers are categorized into different categories, based on the physical processes simulated by the solvers. They can be located in the OpenFOAM installation either by using a predefined alias command

```
?> sol
```

which switches to the parent directory of all the solvers available in Open-FOAM, or by changing manually to the $FOAM_SOLVERS environmental shell variable:

```
?> cd $FOAM_SOLVERS
```

As an example solver, we have chosen the `interFoam` solver for DNS of two-phase flows. The `interFoam` solver implements a mathematical model that describes the flow of two separated immiscible fluids (phases) as a flow of a single fluid (continuum). The separation of phases is done by using an additional scalar field - the volume fraction field (α). This method is referred to as the Volume-of-Fluid (VoF) method and α

denotes the filling level of the cell with the first phase and takes on the values from the interval $[0, 1]$. More information on the VoF method can be found in [8].

The source code files of `interFoam` are located in `$FOAM_SOLVERS/ multiphase/interFoam`. Listing the contents of the `interFoam` directory, results in the following contents:

```
?> ls $FOAM_SOLVERS/multiphase/interFoam
alphaSuSp.H correctPhi.H createFields.H initCorrectPhi.H interFoam.C
interMixingFoam/ Make/ overInterDyMFoam/ pEqn.H rhofs.H UEqn.H
```

Every solver directory contains many different files that accompany the main solver implementation file, in this case `interFoam.C`. When the solver implements a *system* of PDEs that are coupled together by different terms, it is necessary to achieve a solution that does not only satisfy a single PDE, but all of them. The additional files implement parts of a so-called *solution algorithm*: the algorithm that determines how *coupled* PDEs are solved in OpenFOAM. Implementing equations in separate files and including them in the main solver application increases significantly the readability of the solution algorithm implemented by the solver application. In addition to a better overview of the solver application, separating the algorithms into different files makes them easily re-usable for other solvers, while avoiding copies of the source code. The supporting files that contain implementations of important for the VoF solver family can be found in

```
?> ls $FOAM_SOLVERS/multiphase/VoF/
aCourantNo.H alphaEqn.H alphaEqnSubCycle.H
teAlphaFluxes.H setDeltaT.H setRDeltaT.H
```

A shared file can easily be replaced by a file with the same name placed in the solver directory: the `wmake` build system searches the solver directory first for the files to be included. The files containing the "Eqn" suffix contain source code that define the equations of a mathematical model that the solver is implementing. This implementation is done on a very high level of abstraction, making it more human readable. Obviously, `interFoam` uses the momentum equation (`UEqn.H`), the pressure equation (`pEqn.H`), the volume fraction equation (`alphaEqn.H`) as well as other supporting parts of the solution algorithm. The `alphaEqn.H` is shared by different versions of `interFoam` and is therefore stored in `$FOAM_SOLVERS/multiphase/VoF`.

> **INFO**
>
> Whenever a source file is encountered with an *Eqn* suffix, that file contains an implementation of an equation of a mathematical model.

Source code that operates on global field variables and is not implemented in a form of a function or a class, is distributed in files that are included by the solver applications. Including the file `CourantNo.H` inside solver code, includes the calculation of the current Courant number in the solver application which is based on the field variables available at the global scope. In order to prevent code duplication, the number of such files that share common source code snippets is reduced to a minimum with the help of the build system. All shared included files are stored in specific folders such as:

```
?> ls $FOAM_SRC/finiteVolume/cfdTools/general/include
checkPatchFieldTypes.H fvCFD.H initContinuityErrs.H
readGravitationalAcceleration.H readPISOControls.H
readTimeControls.H setDeltaT.H setInitialDeltaT.H
volContinuity.H
```

and links to those files are created in the `$FOAM_SRC/finiteVolume/lnInclude` folder, so this folder can be used in `Make/options` to enable inclusion of the shared files in the solver application. By default, all OpenFOAM applications include the `finiteVolume/lnInclude` as the search directory for the compiler:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
```

and this directory contains the symbolic links to all header files in the `finiteVolume` directory and its sub-directories. Therefore, the included header files shared between different applications from `cfdTools` are by default made available to any OpenFOAM application.

Other globally available functionalities that are implemented in terms of header files are listed above, and they include the following:

- time step adjustment based on the CFL condition,
- correction for volume conservation,
- determination of continuity errors,
- reading the gravitational acceleration vector, ...

One could argue that the calculations implemented in terms of files that are included into the application (solver) code is not a pure object-oriented

Listing 31 Assembly of the momentum equation using DSL / equation mimicing.

```
fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(rhoPhi, U)
  + turbulence->divDevRhoReff(rho, U)
);
```

approach to software design. This is true, however, the very nature of the CFD simulation is procedural: process simulation input to compute the approximative solution and store the output. Therefore, some variables have been made global and thus allways accessible to the solution process, such as the fields and the mesh. In that case, encapsulating operations that are defined in the included files into classes and imposing hierarchy on them is still possible. For example, there is nothing preventing the user to implement a hierarchy of classes that implement different strategies to modify the time step based on the Courant number. Actually, calculations that are performed using the global field variables as arguments can be encapsulated very nicely into function objects, as covered in chapter 12. However, such encapsulation is not allways necessary, and in those situations, the included source files are used.

The high level of abstraction of the OpenFOAM DSL developed for mathematical models is observed in construction of the momentum equation as shown in listing 31. The discrete operators used by the unstructured FVM (divergence `div`, gradient `grad`, curl `curl` and the temporal derivative `ddt`) are implemented as function templates in the C++ language. Using the generic programming allows the programmer to retain the same function names (that are human readible) for functions taking completely different formal parameters. For this reason, there exists a single implementation of the gradient operator and not separate implementations for tensors of different ranks: scalar, vector, symmetric tensor, and the like.

OOD is used to encapsulate things such as interpolations required by the discrete operators, and keeps their interaction invisible to the user of OpenFOAM except as a named entry in the configuration files. As already pointed out in the previous chapters, the `fvSolution` and `fvSchemes` dictionaries are responsible for the control of the numerical schemes and solvers. The equations of the mathematical model invoke discrete

operators on fields such as the velocity, pressure, momentum and so forth. This is why different solvers need different entries in both the `fvSolution` and `fvSchemes` dictionaries, if the `default` entry is set to `none`. Hence neither the user nor the developer of a solver have to care about how any of the schemes are implemented, OpenFOAM selects the chosen schemes automatically based on the specification in the configuration file. As the RTS is enabled for the schemes, the solver must not be recompiled if a discretization changes.

> INFO

> OpenFOAM uses generic programming to implement the discrete differential operators. The operators dispatch the discretization and interpolation operations to a generic hierarchy of discretization and interpolation schemes, respectively. The interpolation/discretization class templates are instantiated and have been made runtime selectable. As a consequence, no change to the solver code is required when a different scheme is chosen.

9.1.1 Fields

As described in chapter 1, the unstructured FVM requires a discrete subdivision of the flow domain into a mesh of finite volumes, onto which the physical fields (e.g. pressure, velocity, temperature) are mapped. The fields, together with the mesh and the solution control (implemented by the `Time` class), are initialized by the solver at the beginning of the simulation but before the time loop starts. The initialization part of the solver code can be inspected in the main solver application file `interFoam.C` is shown in listing 32 Most of the above listed included (*header*) files are made globally available. Some header files are solver-specific and in that case they are not available to any application, but are stored in the solver source directory. Examples of solver-specific files are `createFields.H` and `readTimeControls.H`, which read the solver specific fields and solver related control structures. Whichever fields are initialized in the `createFields.H` will be expected by the solver to be present in the initial (0) directory of the simulation case when the solver is invoked.

Listing 32 A typical set of include files found in the start of OpenFOAM solvers

```
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"

pimpleControl pimple(mesh);

#include "initContinuityErrs.H"
#include "createFields.H"
#include "readTimeControls.H"
#include "correctPhi.H"
#include "CourantNo.H"
#include "setInitialDeltaT.H"
```

9.1.2 Solution Algorithm

The momentum conservation equation contains the term on the right hand side that models the volumetric force density of the pressure acting on the fluid. The pressure field is not known at the beginning of the simulation. The pressure acting upon the fluid enforces a change in the momentum, however, the mass conservation (continuity) equation needs to be satisfied at all times. For an incompressible flow, this condition causes a tight coupling between the continuity equation and the momentum equation. This equation coupling is usually referred to as *pressure-velocity coupling* and there are a multitude of algorithms developed in CFD whose sole purpose is handling the equation coupling while keeping the solution process stable.

The coupling might be addressed by a simultaneous solution of the algebraic equation system using *block-coupled* solvers. A starting point in the research of the block coupled solution algorithms on unstructured meshes in general would be the work by [2]. Developments targeting specifically OpenFOAM are those of [1] and [5], among others. Although it is intuitive to the strong nature of the equation coupling, this approach has historically not been used because of the prohibitive memory constraints of the computers at the time when the pressure-velocity coupling problem has been encountered in CFD. As a consequence, *segregated algorithms* have been developed that modify the original set of equations into a modified set that allows a separate solution of each equation.

Detailed information on CFD solution algorithms for pressure-velocity coupling can be found in textbooks on CFD like the book by [3] and [6]. OpenFOAM implements two major pressure-velocity coupling algorithms: the Pressure-Implicit with Splitting of Operators (PISO) algorithm originally developed by [4] and the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm, originally developed by [7].

Starting from OpenFOAM versions of 2.0 or newer the user interface to the pressure-velocity coupling algorithms has experienced a major refactoring which consolidates both existing algorithms into a single one named PIMPLE. Instead of reading the respective settings from `fvSolu-tion` in each solver explicitly, a new class named `pimpleControl` takes care of all of this on it's construction. This lead to a simplification of `readControls.H` for each solver. Please note that this solely affects the way the particular parameters are read from `fvSolution` and not the implementation of the algorithm in the solver itself. One can use `pim-pleControl` and still implement an entirely different pressure-velocity coupling algorithm instead.

9.2 Customizing the Solver

A majority of solver customization involves modifying the mathematical model equations in one way or the other. Entirely new model equations may be introduced, equation terms may be removed as their influence on the solution is neglected, and new terms can be added to account e.g. for new forces acting on the fluid. While it is impossible to cover all of the possible complexities and difficulties that can arise during the customization of solvers, there are a few examples that are covered in this chapter that should help in understanding problems related to solver customization and how to overcome them. A typical modification of a solver is adding passively transported fields and material properties. The new fields and material properties have to be read by the solver application from the simulation case. There are various files available in the simulation case, where different types of data are read from. Lets start with one of the more basic operations involved: looking up a value in a dictionary. OpenFOAM configuration (dictionary) files and the data structure they are based on are both described in detail in chapter 5. Still, basic information in working with dictionaries is provided also

---

Listing 33 Construction of an `IOdictionary`

```
Info<< "Reading transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        // Name of the file
        "transportProperties",
        // File location in the case directory
        runTime.constant(),
        // Object registry to to which the dict is registered
        mesh,
        // Read strategy flag: read if the file is modified
        IOobject::MUST_READ_IF_MODIFIED,
        // Write strategy flag: do not re-write the file
        IOobject::NO_WRITE
    )
);
```

---

in this chapter, to make the solver customization description more self sustained.

9.2.1 Working with Dictionaries

OpenFOAM configuration files are called *dictionaries* (dictionary files) and are used to provide configuration parameters for the solver application. Between the various dictionaries such as `transportProperties`, `controlDict`, or `fvSolution`, the user has complete control over solvers, material properties, time stepping, and the like. In this section, accessing an existing and a new dictionary as well as looking up a value and loading it into solver scope is covered. Examining how the `icoFoam` solver looks up the material properties from the `constant/transportProperties` dictionary will serve as a simple example of how to dictionaries are handled. To begin, open the `createFields.H` file located at `$FOAM_SOLVERS/incompressible/icoFoam/createFields.H`. At the beginning of this header file, there is an instantiation of the `IOdictionary` object as shown in listing 33. The `IOobject` declaration is commented out in order to indicate the purpose of each argument. More details on these arguments are provided later in this section. The code of listing 33 initializes the dictionary as a global variable, so retrieving values from it remains straightforward. The next few lines of the `cre-`

Listing 34 Example definition of nu in `transportProperties`

```
nu              0.01;
```

Listing 35 Instantiation of `twoPhaseMixture`

```
Info<< "Reading transportProperties\n" << endl;
immiscibleIncompressibleTwoPhaseMixture mixture(U, phi);

volScalarField& alpha1(mixture.alpha1());
volScalarField& alpha2(mixture.alpha2());

const dimensionedScalar& rho1 = mixture.rho1();
const dimensionedScalar& rho2 = mixture.rho2();
```

`ateFields.H` file to contain the code that initializes the viscosity (nu) with a value contained in the `transportProperties` the dictionary:

```
dimensionedScalar nu
(
    "nu",
    dimViscosity,
    transportProperties
);
```

The construction of the dimensioned scalar is done by the constructor

```
dimensioned<Type>::dimensioned<Type>
(
    const word&,
    const dimensionedSet&,
    const dictionary&
)
```

and the predefined dimension set `dimViscosity` is used to set the unit dimensions of the kinematic viscosity nu. The example `transportProp-erties` entry (listing 34) shows how the nu entry is defined. More information on how the dimension system in OpenFOAM works can be found in chapter 5. The following example shows how transport properties are looked up by the `interFoam` solver, which is a bit more complicated. The important source code is located at `$FOAM_APP/solvers/-multiphase/interFoam/` At the beginning of the `createFields.H` file, the `immiscibleIncompressibleTwoPhaseMixture` class is instantiated (see listing 35). There are two member functions (`rho1()`, `rho2()`), used to lookup the material properties of the `twoPhaseMixture` class. Examining the class source code is necessary, in order to find the code that performs the actual dictionary access.

EXERCISE

Figure out how the mixture viscosity and interface curvature are calculated starting from the `immiscibleIncompressibleTwoPhaseMixture` class using the Extended Code Guide.

9.2.2 The Object Registry and reglOobjects

As the fields and the mesh are used in the form of global variables in the OpenFOAM solver applications, tracking all of them and dispatching calls to their member functions explicitly would involve a lot of unnecessary code repetition. An example of such a clustered call dispatch is a request of the solver to write out all fields to the hard disk. In the case of using objects directly, the names of the objects would be hardcoded and changing the names would introduce a cascade of changes in the application code. Also, the code that implements such calls would need to be copied on multiple places. For example, the same code responsible for the field output would then be copied to every application that relies on the same set of fields with the same variable names. Using included header files as covered in section 9.1 would be possible for the solver family that relies on same field variables. But changing a single field variable would render such a header file unusable for the entire solver family. This approach hence represents a *stiff* software design, or a software design which *does not scale well*. A stiff or not-scaling software design is a design for which a single extension requires the modification of existing code, and furthermore, the modification occurs often at multiple places in the existing code base.

Because of this issue, the logic behind coordination of operations for multiple objects has been encapsulated into a class, that can then be re-used at many places in the OpenFOAM code. To this purpose, an *object registry* has been implemented: an object that registers other objects to itself and then dispatches (forwards) the call to its member functions to the registered objects. The object registry is an implementation of the *Observer Pattern* from the OOD and it is described in more detail in section 8.4. Additionally, there is a great review of the object registry as well as the registered object classes on the OpenFOAM Wiki page [1].

An example of the use of an object registry is a boundary condition

---

[1]`http://openfoamwiki.net/index.php/Snip_objectRegistry`

Listing 36 Total pressure boundary condition dispatch to object registry.

```
void Foam::totalPressureFvPatchScalarField::updateCoeffs()
{
   updateCoeffs
   (
      p0(),
      patch().lookupPatchField<volVectorField, vector>(UName())
   );
}
```

---

implementation where the boundary condition that operates on one field, requires access to another field. The total pressure boundary condition (`totalPressureFvPatchScalarField`) is such a boundary condition that requires access to multiple fields in order to update the field it is assigned to.

EXERCISE

Find out what is the `totalPressureFvPatchScalarField` boundary condition meant to be used for. Which member function performs the actual calculations? How are the alternative calculations implemented? Can you think of an alternative runtime selectable implementation for the alternative calculations?

The total pressure boundary condition updates the field it has been assigned to as it is done by all other boundary conditions in OpenFOAM, using the `updateCoeffs` member function as shown in listing 36. However, the `updateCoeffs()` dispatches the calculation to the overloaded `updateCoeffs`. The second argument of the `updateCoeffs` call shown in listing 36 accesses the `fvPatch` constant reference attribute of the boundary field using the `patch()` member function. On the other hand, the `fvPatch` class stores a constant reference to the finite volume mesh, which inherits from `objectRegistry` and is therefore *an* object registry. The `lookupPatchField` is defined as a template member function of the `fvPatch` class template, in the file `fvPatchFvMeshTemplates.H` and it is shown in listing 37. The return type declaration of the function is dependant on the template parameter `GeometricField` and therefore requires the `typename` keyword, in order to make the compiler aware that the `PatchFieldType` is indeed a type. The more important part of the member function template is the return statement that obviously makes use of the object registry functionality, which is inherited to the `fvMesh`

Listing 37 Looking up a geometric internal field from within a boundary mesh patch.

```cpp
template<class GeometricField, class Type>
const typename GeometricField::PatchFieldType&
Foam::fvPatch::lookupPatchField
(
    const word& name,
    const GeometricField*,
    const Type*
) const
{
    return patchField<GeometricField, Type>
    (
        boundaryMesh().mesh().objectRegistry::template
            lookupObject<GeometricField>(name)
    );
}
```

from the object registry class *objectRegistry*. Since `fvPatch` is a class template, the call to the base class member function is a bit convoluted. `lookupObject` is a member function template of the `objectRegistry` base class, and this must be specified at the member function call site using the `template` keyword. Looking past all the C++ template code, the finite volume mesh boundary patch accesses the entire boundary mesh, then the corresponding volume mesh and asks the volume mesh to look up a field with a specific name (`name`). This call path through the involved classes as described for this example allows the total pressure boundary condition to access a field from a mesh, based on the field name paremeter.

9.3 Implementing a new PDE

In this section a new solver is implemented by adding a new PDE is to the `interFoam` solver for simulating two incompressible immiscible fluid phases. The purpose of the PDE and its supporting code to show how the implementation is performed, and not to model a realistic physical transport process. Modeling of transport phenomena on and across fluid interfaces requires careful and rigorous derivation. For the most part, the equation stems from an implementation available in the `compress-ibleInterFoam` solver. There, the heat transfer across the interface is

accounted for, due to the use of equations of state which couple pressure, temperature and density. For the sake of simplicity, thermodynamic equations of state are omitted and laminar flow is assumed.

9.3.1 Additional Model Equation

As an example of a model of the unsteady passive scalar transport in a laminar two phase flow, the following equation is used

$$\frac{\partial \rho T}{\partial t} + \nabla \cdot (\rho \mathbf{U} T) - \nabla \cdot (D_{eff} \nabla T) = 0, \tag{9.1}$$

where T denotes the temperature, ρ is the density, $\mathbf{U}$ is the velocity, D_{eff} is the thermal conductivity coefficient of the phase mixture.

The `interFoam` solver implements two-phase Navier-Stokes equations in a single field formulation to model the two-phase flow of two immiscible incompressible fluid phases. This model describes the flow of two immiscible fluid phases as the flow of a single continuum, by introducing an additional scalar (volume fraction field) to distinguish between fluid phases. More information on two-phase flow modeling can be found in the book by [8].

Any cells with an α value in between with be considered interface cells with material properties weighted as a mixture of the two primary phases, since α is defined as the volume fraction by

$$\alpha = \frac{V_1}{V}, \tag{9.2}$$

where V_1 is the volume occupied by phase 1 in the cell that has the total volume V. The properties of the single continuum are then modeled using the volume fraction field, forming mixture quantities, such as the mixture viscosity

$$\nu = \nu_1 \alpha_1 + (1 - \alpha_1)\nu_2 \tag{9.3}$$

or mixture density

$$\rho = \rho_1 \alpha_1 + (1 - \alpha_1)\rho_2, \tag{9.4}$$

where ν_1, ν_2 and ρ_1, ρ_2 are the kinematic viscosities and densities of respective two fluid phases. The effective heat conduction coefficient in

equation 9.1 D_{eff} is modeled a similar way in this example, using the α_1 field

$$D_{eff} = \frac{\alpha k_1}{C_{v1}} + \frac{(1-\alpha)k_2}{C_{v2}}. \tag{9.5}$$

Here, k_1, k_2 and C_{v1}, C_{v2} represent the conduction coefficients and heat capacities of respective fluid phases.

The heat conduction coefficient modeled by equation 9.5 is based on the volume fraction field in the similar way as other phase properties. Using the volume fraction field values this way assignes for this example constant values of the physical properties to the bulk regions of the two fluid phases. The region of the fluid interface will have values of α_1 within the interval $[0,1]$ and will therefore define a transition region between the two constant values of the respective phase properties. The shape of the profile of the transition region will be determined by the nature of the model that describes the mixture property, like the conduction coefficient (see equation 9.5). How accurately this approach can be applied on the real physical problem of heat transfer accross a moving interface is not important for describing how to perform solver modifications in OpenFOAM and is therefore left out of scope.

9.3.2 Preparing the Solver for Modification

Before changing the source code of an existing solver, a new copy of the solver application must be created. The source code directory of the `interFoam` solver should be copied to the personal applications directory and renamed:

```
?> cp -r $FOAM_SOLVERS/multiphase/interFoam/ \
   $FOAM_RUN/../applications/
?> cd $FOAM_RUN/../applications/
?> mv interFoam heatTransferTwoPhaseSolver
?> cd heatTransferTwoPhaseSolver
?> mv interFoam.C heatTransferTwoPhaseSolver.C
```

> INFO
>
> Each OpenFOAM version exports the `$FOAM_RUN` variable for the user working directory, but does not generate it during the installation of OpenFOAM. In case this directory is not available, create it with `mkdir -p $FOAM_RUN`.

In the next steps all the files that are not to be modified must be deleted. This is done to prevent code duplication, that should always be avoided. If code is duplicated, maintaining the solver becomes problematic, because, as OpenFOAM evolves, the solver files need to be updated. That is why the changes introduced to existing files when implementing a solver should be kept to a minimum.

> **INFO**
>
> When developing a new solver, keep the changes introduced in shared files to a minimum. Introducing additional files and including them into your new solver application while re-using existing files makes your new solver easier to maintain, because re-using existing files will pick up changes in those files in new OpenFOAM versions.

The directory should be cleaned up from all the files that are not changed

```
?> rm -rf interMixingFoam/ overInterDyMFoam/
?> rm alphaSuSp.H createFields.H correctPhi.H
   initCorrectPhi.H rhofs.H pEqn.H UEqn.H
```

> **INFO**
>
> Alternatively if you know from the start which solver files must modified, only those files and the `Make` directory can be copied.

Since the solver source code files have been renamed, the `Make/files` must be modified so it reflects the changes. The `Make/files` config file should contain only the lines shown here:

```
heatTransferTwoPhaseSolver.C

EXE = $(FOAM_USER_APPBIN)/heatTransferTwoPhaseSolver
```

letting the `wmake` build system know that an executable application is to be built and installed in the user application binaries directory, from the renamed solver source code file. So, this will instruct `wmake` to compile the file `heatTransferTwoPhaseSolver.C` along with its dependencies and create an executable solver called `heatTransferTwoPhaseSolver`. Calling `wmake` at this point would fail: we have deleted the equation files, the field initialization, etc, and ended up with

```
heatTransferTwoPhaseSolver $ ls
heatTransferTwoPhaseSolver.C  Make
```

In order to inform the new solver to re-use the files from interFoam and shared files from `$FOAM_SOLVERS/multiphase/VoF`, `Make/options` must be modified into

```
INC = \
-I$(FOAM_SOLVERS)/multiphase/interFoam \
-I$(FOAM_SOLVERS)/multiphase/VoF \
...
```

This way `wmake` is instructed to look into the folder of the `interFoam` solver first for files shared with `interFoam`, then into `$FOAM_SOLVERS/multiphase/VoF` for files shared with the VoF solver family.

At this point, the new `heatTrasnferTwoPhaseSolver` can be built with `wmake` and it will behave exactly as `interFoam`.

9.3.3 Adding the temperature field and heat conduction coefficients

Two new required material phase properties must be looked up for each phase from the transport properties dictionary: the heat conduction coefficient k and the heat capacity Cv. Additionally, the new temperature T and the effective heat transfer coefficient D_{eff} fields need to be initialized. The initialization is placed in a new initialization file `createHeatTransferFields.H`, together with the initialization of the temperature and heat transfer coefficient fields, as shown in in listing 38. The new `createHeatTransferFields.H` shown in listing 38 should be inserted in new solver after `createFields.H` file, after the `immiscibleIncompressibleTwoPhaseMixture` has been initialized. The heat capacity of type `dimensionedScalar` will be loaded from each phases' subdictionary in the `transportProperties` dictionary file. With all of the dictionary lookup and declaration complete, the new model equation 9.1 is to be added to the solution algorithm.

9.3.4 Programming the temperature equation

The temperature equation is placed into a separate file `TEqn.H` and this file is then included in the solver application. The code defined in the `TEqn.H` file implements two operations. It calculates the coefficient D_{eff}

Listing 38 Looking up heat conduction coefficients from the `twoPhase-Properties` dictionary, initializing the temperature field and the effective heat transfer coefficient field.

```cpp
const dictionary& phase1dict = mixture.subDict ("phase1");
const dictionary& phase2dict = mixture.subDict ("phase2");

auto k1 (phase1dict.get<dimensionedScalar>("k"));
auto Cv1 (phase1dict.get<dimensionedScalar>("Cv"));
auto k2 (phase2dict.get<dimensionedScalar>("k"));
auto Cv2 (phase2dict.get<dimensionedScalar>("Cv"));

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

volScalarField Deff
(
    "Deff",
    (alpha1*k1/Cv1 + (1.0 - alpha1)*k2/Cv2)
);
```

Listing 39 Heat transport equation implementation.

```
Deff == alpha1*k1/Cv1 + (1.0 - alpha1)*k2/Cv2);
solve
(
   fvm::ddt(rho, T)
 + fvm::div(rhoPhi, T)
 - fvm::laplacian(Deff, T)
);
```

Listing 40 Including TEqn.H into the solution algorithm.

```
while (pimple.loop())
{
   ...
   mixture.correct();

   if (pimple.frozenFlow())
   {
      continue;
   }

   #include "TEqn.H"
   #include "UEqn.H"
   ...
}
```

and it solve the passive temperature transport PDE given by equation 9.1.

The D_{eff} coefficient is calculated as a linear weighted average based on the cell center `alpha1` value (equation 9.5). For this example, the same approach to computing the D_{eff} coefficient has been applied as the one used in the `compressibleInterFoam` solver. Since the D_{eff} coefficient will vary from cell to cell, it is implemented as as a `volScalarField`, as opposed to a single scalar. With the diffusion coefficient field initalized in `createHeatTransferFields.H`, the transport equation can be implemented as shown in listing 39. Now that `TEqn.H` is complete, it needs to be added to the solution algorithm implemented in the main solver application file `heatTransferTwoPhaseSolver.C`. The `TEqn.H` should be inserted above the internal PIMPLE loop, nex to the momentum equation as shown in listing 40. That completes the source code modifications required to add the heat transport equation to the solution algorithm of the `interFoam` solver. The solver directory should be cleaned up from the old files generated by the build process, and then the solver application is to be compiled:

```
?> wclean
?> wmake
```

Once the new solver is implemented a simulation case needs to be set up which is compatible with the new solver.

9.3.5 Setting up the Case

With the new solver created, new initial conditions, boundary conditions, material properties, and solver control input parameters are now required to handle the heat transfer computation. In the 0 directory, the temperature field is added in the form of the T file, with the appropriate dimensions

```
FoamFile
{
    version    2.0;
    format     ascii;
    class      volScalarField;
    location   "0";
    object     T;
}

dimensions    [0 0 0 1 0 0 0];
```

Boundary conditions should be set, and in this example we simply use zero-gradient boundary conditions. To set the initial condtions for the new temperature field, the setFieldsDict file in the system directory needs to be edited. For this example, we set the temperature of a rising bubble to a higher value than its environment:

```
defaultFieldValues
(
    volScalarFieldValue alpha1 0
    volVectorFieldValue U (0 0 0)
    volScalarFieldValue T 300
);

regions
(
    sphereToCell
    {
        centre (1 1 0);
        radius 0.2;
```

```
        fieldValues
        (
            volScalarFieldValue alpha1 1
            volScalarFieldValue T 500
        );
    }
);
```

The `transportProperties` file in the `constant` directory should also be modified to add `k` and `Cv` coefficient values to each phase. We have chosen the values arbitrarily:

```
e1

transportModel  Newtonian;

nu              nu  [ 0 2 -1 0 0 0 0 ] 1e-02;
rho             rho [ 1 -3 0 0 0 0 0 ] 100;
k               k   [ 1 1 -3 -1 0 0 0] 100;
Cv              Cv  [ 0 2 -2 -1 0 0 0] 100;
...

e2

transportModel  Newtonian;

nu              nu  [ 0 2 -1 0 0 0 0 ] 1e-02;
rho             rho [ 1 -3 0 0 0 0 0 ] 1000;
k               k   [ 1 1 -3 -1 0 0 0] 1000;
Cv              Cv  [ 0 2 -2 -1 0 0 0] 1000;
...
```

Since a new PDE is added to the solver, discretization schemes should be chosen for its differential operators in `system/fvSchemes`, in case the default options are not provided or applicable. In this example, the default CDS (Gauss linear) scheme is to be replaced with the upwind scheme:

```
divSchemes
{
    default       Gauss linear;
    div(rho*phi,U) Gauss limitedLinearV 1;
    div(rho*phi,T) Gauss upwind;
```

Finally, parameters involving the solution of the linear system resulting from the discretization of the temperature equation should be set. `TFinal` is added to the `fvSolution` file, for example

```
TFinal
{
    solver        PBiCG;
    preconditioner DILU;
    tolerance     1e-06;
```

```
    relTol      0;
}
```

These are the options that set the solver for the heat transfer equation, as well as its parameters. That should be the final case configuration change needed to run the new solver with an added new model equation. The solver can be run within the case directory and the results of the simulation can be analyzed.

9.3.6 Executing the solver

In order to run the case, make sure that the library and the application code of the example code repository is compiled automatically by invoking

```
?> ./Allwmake
```

in the main directory of the example code repository. To start the solver, simply execute the following command within the simulation case directory:

```
?> heatTrasferTwoPhaseSolver
```

Figure 9.1 shows the distribution of the volume fraction field α and the temperature field T for the simulation for the two-dimensional rising bubble. The bubble temperature is set to be higher than the temperature of the surrounding fluid, which should cause the bubble to release heat flux to its surroundings as it rises.

EXERCISE

Exact solutions exist for heat conduction in different geometries, for example a solid cylinder or a plate. Create a simulation case for a cylinder surrounded by hot still air, or two plates in contact and verify the new solver. Does the solution converge to the exact solution?
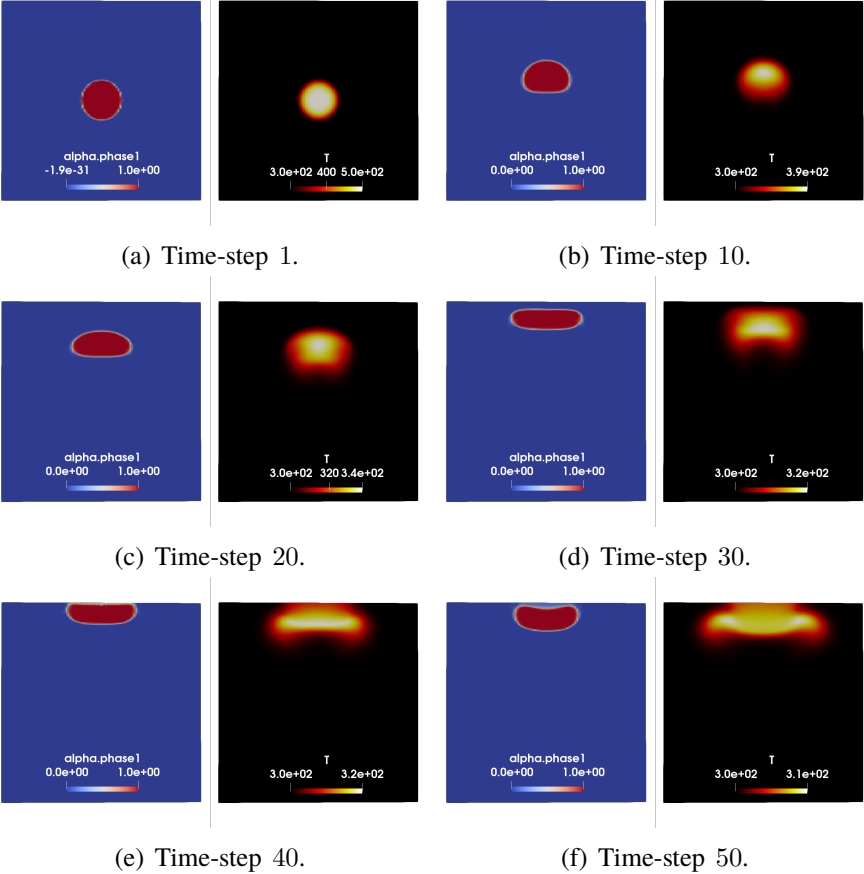
(a) Time-step 1.

(b) Time-step 10.

(c) Time-step 20.

(d) Time-step 30.

(e) Time-step 40.

(f) Time-step 50.

Figure 9.1: Temperature distribution for the 2D rising bubble simulation case for two chosen simulation times.

.

Further reading

[1] I Clifford and H Jasak. "The application of a multi-physics toolkit to spatial reactor dynamics". In: *International Conference on Mathematics, Computational Methods and Reactor Physics*. 2009.

[2] M Darwish, I Sraj, and F Moukalled. "A coupled finite volume solver for the solution of incompressible flows on unstructured grids". In: *Journal of Computational Physics* 228.1 (2009), pp. 180–201.

[3] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer, 2002.

[4] R. I. Issa. "Solution of the implicitly discretised fluid flow equations by operator-splitting". In: *Journal of Computational physics* 62.1 (1986), pp. 40–65.

[5] Kathrin Kissling et al. "A coupled pressure based solution algorithm based on the volume-of-fluid approach for two or more immiscible fluids". In: *V European Conference on Computational Fluid Dynamics, ECCOMAS CFD*. 2010.

[6] Suhas Patankar. *Numerical heat transfer and fluid flow*. CRC Press, 1980.

[7] S.V. Patankar and D.B. Spalding. "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows". In: *International Journal of Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806.

[8] G. Tryggvason, R. Scardovelli, and S. Zaleski. *Direct Numerical Simulations of Gas-Liquid Multiphase Flows*. Cambridge University Press, 2011. ISBN: 9780521782401.

10

Boundary conditions

For this chapter, a solid understanding of the unstructured Finite Volume Method (FVM). Chapter 2 covers the details of the unstructured mesh and section 1.3 summarizes the unstructured FVM. This chapter covers the use of boundary conditions in OpenFOAM simulations as well as their further development.

10.1 Boundary conditions in a nutshell

In the FVM (cf. section 1.3) a single algebraic equation is generated per each cell when an implicit method is used to discretize the model equation. The coefficients of the linear equation system are determined by the connectivity of the unstructured mesh and the interpolation schemes (`system/fvSchemes` dictionary). If the face belongs to the domain boundary, boundary conditions are applied because the face only has a single adjacent internal cell so there is nothing to interpolate from the other side. Without boundary conditions, the face-centered values on the boundaries cannot be determined and the linear system cannot be completed.

In the finite-volume mesh, each boundary face belongs to a boundary patch (a collection of boundary faces) and each boundary patch is defined to be of a certain type (cf. chapter 2). The boundary mesh is therefore a set of boundary patches. The type of the boundary patch does limit the choice of boundary conditions for all flow fields, but is not a boundary

condition in itself. An overview over the different boundary types is provided in chapter 2.

Any field file in an OpenFOAM case contains two different sections: `internalField` and `boundaryField`. The `boundaryField` describes how the values on the boundaries (boundary patch fields) are prescribed and the `internalField` does the same for the volume center values. The values on the boundary of other geometric fields (point field and surface field) are determined in a similar way. Details on boundary field operations are provided in chapter 2 as well as the [2].

10.2 Boundary condition design

Before going into details of how boundary conditions are implemented in OpenFOAM, the process of defining boundary conditions from a user's perspective is covered in this section.

10.2.1 Internal, boundary and geometric fields

The boundary of the domain and the respective boundary patches can be found in in the `constant/polyMesh/boundary` file. Each boundary-patch is a set of cell (finite volume) faces. The `boundary` file is rarely examined by the OpenFOAM user; it could be helpful to the programmer in understanding the data structures OpenFOAM uses to store the connectivity of the unstructured mesh.

OpenFOAM stores the fields as files in the initial time step directory (0). As an example, consider a part of the configuration file for the dynamic pressure field `p_rgh` of the simulation case `rising-bubble-2D`:

```
    internalField   uniform 0;

    boundaryField
    {
      bottom
      {
        type          zeroGradient;
      }
```

The `internalField` keyword relates to the values stored in the cell centers (uniform field with value 0), and the boundary condition for the `bottom` mesh patch is defined to be of type `zeroGradient` (cf. chapter 1 for the description of the numerics).

OpenFOAM manipulates tensor fields that are associated with different elements of the unstructured mesh. For example, a `volVectorField` for the velocity U and a `volScalarField` for the pressure field p are associated with cell-centers, while a `surfaceScalarField` stores the volumetric flux field `phi`, associated to face centers. A brief look into any solver shows the many different operations executed on those fields. In order to illustrate some common operations on a geometric field, the `volScalarField` p serves as the source of examples:

Accessing field values is simply done by passing the particular cell label to the access operator `[](const label&)` of the field. In this example we choose cell 4538:

```
const label cellI{4538};
Info<< p[cellI] << endl;
```

Accessing values on the boundary is slightly more complex, because the `volScalarField` does not store any values on cell faces directly. Boundary values are determined by the boundary conditions defined on the particular boundary. Calculating the maximum value of p on the first boundary patch of the mesh could be achieved using the following code:

```
const label boundaryI{0};
Info<< "max(p) = "
    << max(p.boundaryField()[boundaryI])
    << endl;
```

> **INFO**
>
> The order in which the boundary patches are stored in the boundary mesh is the same for all fields and it is determined by the way the patches are listed in the `polyMesh/boundary` file. This is, in turn, determined by the used mesh generator.

By default any field returns values from its `internalField`, when accessed with the access operator `operator[](const label&)`. To access values of the boundary field the `boundaryField()` member function must be called (`boundaryFieldRef()` for con-constant access). This returns a list of boundary patches, one boundary patch for each mesh

boundary. Each element is an abstract representation of the boundary condition chosen by the user for this patch. Depending on the type of field, this representation either inherits from `fvPatchField` or `point-PatchField`, though the first one is the one most commonly used.

> **INFO**
>
> It useful to look into the Extended Code Guide when working with classes in OpenFOAM to find what member functions are available and what are their interfaces.

As shown in the above example, OpenFOAM implements fields as so-called *geometrical fields*, that separate values into two sets: internal and boundary values. The geometrical fields are class templates in Open-FOAM as they store tensors of different ranks and associate them to different mesh elements. For example, if we imagine a geometrical mesh to be consisted of line segments, and call it a *line mesh*, the corresponding model of the geometrical field concept that works with the line mesh would be named *line field*. The line field would map tensorial values to the center of each line (internal field values), and to *two boundary end points* of the line mesh (boundary field values). The class template that implements the geometrical field concept is named `Geometrical-Field`, and its instantiations result in different geometrical field models, that map to different kinds of meshes. In OpenFOAM different models of geometrical fields are available:

1. The first category are the well known fields like `volScalarField` and `volVectorField`, that store the data in the cell center. On the boundary, boundary conditions must be applied to mimic values stored in the face center. The suffice the naming convention of `vol*Field`.
2. The second category are fields that store data in the face center, for each face of the mesh. This is not limited to the boundary of the domain. One of these field types is the `surfaceScalarField` that is used to define the flux ϕ between two adjacent cells. All fields of this category are named `surface*Field`.
3. The third category contains the `pointScalarField` or `pointVec-torField` types. Fields of this category are likely to be missed when talking about OpenFOAM fields: Fields that store the data in the points of the mesh. Each point in the mesh has it's own value and on the boundaries, boundary conditions must be defined for
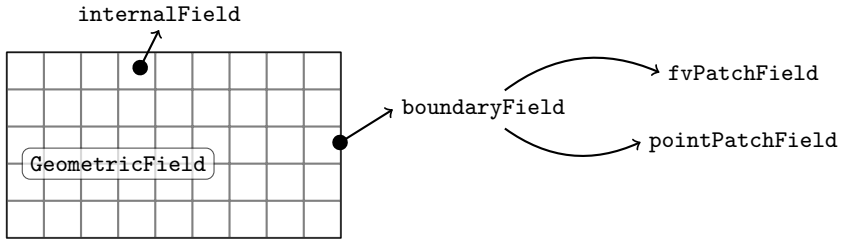
Figure 10.1: Composition of a `GeometricField` for a $9 \cdot 5$ cell mesh.

the points on the boundary and not for the face centers. Looking for `point*Field` in the source code will show all fields of this category.

After this clarification, we can dive into the relationship between fields and boundary conditions. From the design perspective, the boundary conditions in OpenFOAM encapsulate field values mapped to the domain boundaries together with member functions responsible for calculations that determine those boundary values based on internal values - the *boundary condition*. The `GeometricField` class template provides member functions which simplify the formulation and update of boundary conditions. Some functionalities of the member functions include computing values of the boundary fields and taking values stored in the adjacent internal cells as arguments. Both are key requirements to an implementation of any boundary condition. More information on which member functions are supposed to do which task is provided in the following section 10.2.2. The boundary fields - and thus the boundary conditions - are encapsulated together with the `internalField` to form `Geometric-Field` (see figure 10.1). Any of the three geometric field models, is a typedef for a `GeometricField` with appropriate template arguments.

The structure of boundary conditions can be investigated in the source code. For example, consider the declaration part of the `GeometricField` class template shown in listing 41. From listing 41 it is obvious that the `GeometricField` is derived from `DimensionedField`. This means that any default arithmetic operations executed on any instantiation of the `GeometricField` class template (a geometric field model) will be performed excluding the boundaries. This makes sense, because the values on the boundaries should only be determined by the respective boundary conditions. Using the additional assignment operator `GeometricField::op-`

Listing 41 `GeometricField` declaration.

```
template<class Type, template<class> class PatchField, class GeoMesh>
class GeometricField
:
    public DimensionedField<Type, GeoMesh>
{
```

`erator==`, the operations are extended to take the boundary fields into account as well. Values on the boundary can be overwritten from outside of the actual boundary condition, until the boundary conditions are evaluated again by the `GeometricField::correctBoundaryConditions()` member function.

Deriving from a class (`: public` in the above code snippet) results in inheriting its member functions. In case of the `GeometricField`, arithmetic operators are inherited from `DimensionedField`. The `Dimen-sionField` models the *internal field values*, since the boundary fields are *composed* by `GeometricField` with the `GeometricBoundaryField` attribute. Therefore, overloaded arithmetic operators from `Dimensioned-Field` *exclude boundary field values*.

> **INFO**
>
> Operator `GeometricField::operator==` is not a logical equality comparison operator in OpenFOAM, it extends the assignment of the `GeometricalField` to include boundary field values. It is *added* to the `GeometricField` interface to include arithmetic operations on the *composited* boundary fields.

The boundary field itself is declared as a nested class template declaration, and the `GeometricField` stores it as a private attribute. The boundary field class template is named `GeometricBoundaryField`, its declaration is found within the `GeometricField` class template and it contains a list of boundary patch fields, one patch field per each mesh boundary. Although the `GeometricBoundaryField` is encapsulated in `GeometricField`, a non-constant access is provided, which results in the ability of client code of `GeometricField` to change the value of the boundary fields. At the first glance, this breaks encapsulation of the boundary fields by the geometric field, however, the benefit outweighs the

---

Listing 42 Non-const access to the boundary fields in `GeometricField`.

```
Boundary& boundaryFieldRef(const bool updateAccessTime = true);

//- Return const-reference to the boundary field
inline const Boundary& boundaryField() const;
```

---

design principles, because this approach results in a much greater usage flexibility. For example, a thermodynamical model may alter field values in a way that is dependent on another geometric field. The change in this case is driven from outside `GeometricField`, as will often be the case, as the geometric fields are global variables in OpenFOAM, operated on by the solver that is implemented as a procedural sequence of calculations. Non-const access to the boundary field data member is shown in listing 42.

In contrast to the boundary field, the internal field is handled differently within the `GeometricField`. As the `GeometricField` is derived from `DimensionedField`, there is no need to return a different object. To access the internal field, a reference to `*this` is returned, as the `DimensionedField` implements the internal field and its arithmetic operations with dimension checking. Listing 43 clarifies how the geometric field provides access to the internal field. The `internalField()` member function returns the non-const reference to the `GeometricField` which, by inheritance, *is a* `DimensionedField`.

At this point you should have an overview of the fields involved in the simulation in OpenFOAM and why the `GeometricBoundaryField` is encapsulated in the `GeometricField`, with non-constant access prepared for manipulation done by client code of the `GeometricField` class template. Analyzing the class inheritance and collaboration diagrams, although certainly helpful, is not as efficient in gaining understanding as using the classes themselves, which is addressed in the following sections.

10.2.2 Boundary conditions

Boundary conditions, as their name implies, add functionality to values stored in the boundary field (`GeometricBoundaryField`). In Object Oriented Design (OOD), adding functionality usually implies extending

Listing 43 `GeometricField` providing access to internal dimensioned field.

```cpp
// GeometricField.H
//- Return internal field
InternalField& internalField();

// GeometricField.C
template
<
  class Type,
  template<class> class PatchField,
  class GeoMesh
>
typename
Foam::GeometricField<Type, PatchField, GeoMesh>::InternalField&
Foam::GeometricField<Type, PatchField, GeoMesh>::internalField()
{
    this->setUpToDate();
    storeOldTimes();
    return *this;
}
```

existing classes, which is done in the case of boundary conditions as well. In fact, the `GeometricBoundaryField` described above does not encapsulate only field values stored at the domain boundary, each of the fields are extended with virtual member functions that determine the boundary condition behavior.

The boundary conditions represent a hierarchical concept in the FVM - similar boundary conditions are grouped into boundary condition categories. For this reason, and to enable the user to select the boundary condition at runtime (RTS), they are modelled as a class hierarchy. The top parent abstract class `fvPatchField` defines the class interface to which each boundary condition must conform. Every boundary condition in OpenFOAM is either derived from `fvPatchField` or `pointPatch-Field`. The latter is mostly used for applications involving mesh motion or modification. Both have a constant private attribute that is called `internalField_` and is a reference to the internal field of the `Geometric-Field`, that was introduced in the previous section. The attribute provides access to values of the internal field, not only to the cells that are directly adjacent to the boundary mesh patch. In case of a `pointPatchField`, the `interalField_` attribute is declared as:

```cpp
const DimensionedField<Type, pointMesh>& internalField_;
```

The declaration of the internal field for a `fvPatchField` is the same as for the `fvPatchField`, but the second template argument to `DimensionedField` is `volMesh`, rather than `pointMesh`:

```
const DimensionedField<Type, volMesh>& internalField_;
```

Since the `pointPatchField` conforms to the same class interface as the `fvPatchField`, and the volume fields are most often encountered, the `fvPatchField` is covered in this section.

Before we go into detail about which member function of `fvPatchField` is of relevance, when it comes to the implementation of a new boundary condition, we try to conclude the overview of the connection between the `GeometricField` and the actual access to the boundary conditions. A graphical representation of this relation is given in figure 10.2. The geometrical field composes the geometrical boundary field, which inherits from (`FieldField`) and therefore is a collection of (boundary) fields. The composition of the geometrical boundary field is required because the modification of the internal field value requires an update of the boundary field values by the boundary conditions. Also, the boundary fields cannot be separated into objects distinct from the internal field. Internal and boundary fields are not only topologically attached to each other, via the mesh, the FVM requires the boundary field values when the equation is discretized to compute internal field values. The refinement of the mesh causes splitting of cell faces, so the lengths of internal and boundary fields are indirectly connected in this case as well. Having separate internal and boundary fields therefore would make no sense at all. It would introduce global variables that need to be explicitly synchronized, which would severely complicate the semantics of all field operations on the application level. The geometric field loops over the collection of boundary fields and updates each boundary field by delegating the update to the corresponding boundary condition. Figure 10.2 shows `PatchField` template parameter which, when instantiated (`fvPatchField` for a volume mesh), is a boundary condition.

An OpenFOAM application calls the `correctBoundaryConditions()` member function of the `GeometricField` when the internal field is modified and the boundary conditions are to be updated: after a PDE is solved for the field in a solver or the internal values are explicitly calculated in a pre-processing application. The implementation of the `GeometricField::updateBoundaryConditions()` member function is shown in listing 44.

Listing 44 The `GeometricField::updateBoundaryConditions()` member function.

```
// Correct the boundary conditions
template
<
  class Type,
  template<class> class PatchField,
  class GeoMesh
>
void Foam::GeometricField<Type, PatchField, GeoMesh>::
correctBoundaryConditions()
{
    this->setUpToDate();
    storeOldTimes();
    GeometricBoundaryField_.evaluate();
}
```

INFO

To understand how boundary conditions are updated by the `Geometric‑Field` the last line in listing 44 must be understood.

The last line in listing 44 calls the `evaluate()` member function of the `GeometricBoundaryField` that in turn performs a variety of tasks. This member function calls the `initEvaluate()` member function of `fvPatchField`, if the boundary condition has not been initialized. Otherwise the `evaluate()` member function is called. Due to the zero halo layer parallelism implemented by OpenFOAM, the parallel communication is handled handled by `GeometricBoundaryField::evaluate()`, as the process boundaries are also implemented as boundary conditions.

The `GeometricBoundaryField::updateCoeffs()` member function is the other major member function that triggers functionalities of the particular `fvPatchField` from the client code. Compared to `evaluate()`, the implementation of `updateCoeffs()` is shorter since no parallel communication is implemented. The implementation of the `Geometric‑Boundary‑Field::updateCoeffs` is shown in listing 45.

The `forAll` loop in listing 45 loops over all patches of the `Geometric‑BoundaryField` which is referred to as `*this`. The `updateCo‑effs()` member function of `fvPatchField` is called directly for each element of the domain boundary, using the `operator[]`.

Listing 45 The `GeometricBoundaryField::updateCoeffs` member function.

```
template
<
  class Type,
  template<class> class PatchField,
  class GeoMesh
>
void
Foam::GeometricField
<
  Type,
  PatchField,
  GeoMesh
>::GeometricBoundaryField::updateCoeffs()
{
    if (debug)
    {
        Info<< "GeometricField<Type, PatchField, GeoMesh>::"
            "GeometricBoundaryField::"
            "updateCoeffs()" << endl;
    }

    forAll(*this, patchi)
    {
        this->operator[](patchi).updateCoeffs();
    }
}
```
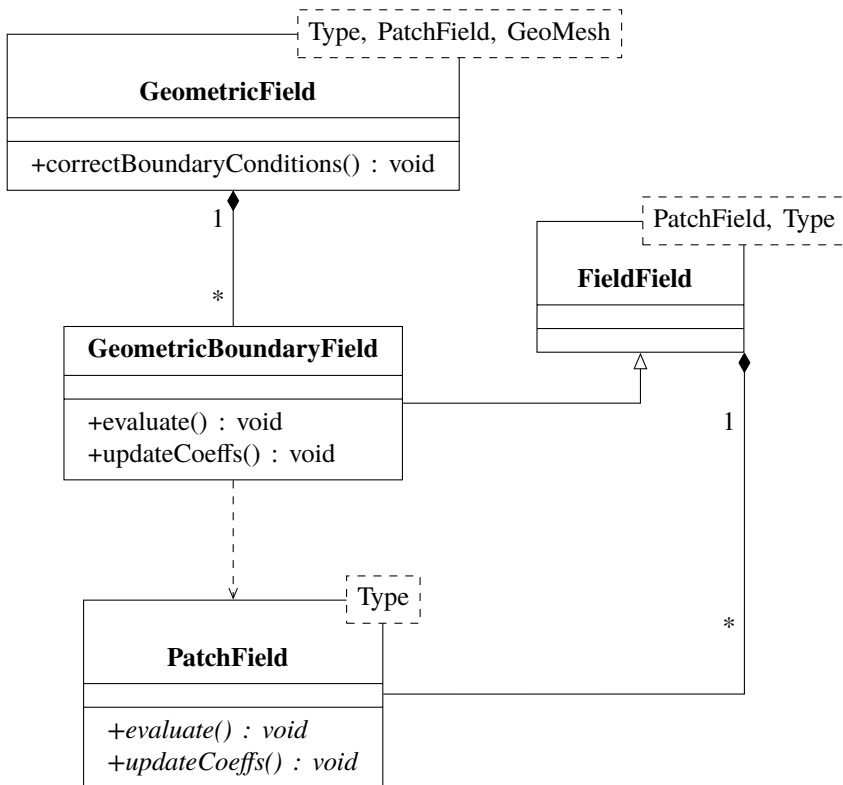
Figure 10.2: Class collaboration diagram for the boundary conditions.

The updateCoeffs() and evaluate() member functions represent the relevant part of the public interface to the fvPatchField that is accessed automatically from each solver. The major difference between both member functions is that evaluate() can be called an arbitrary number of times in a single time step, but is only executed once. On the other hand, updateCoeffs() does not check if the boundary condition is updated or not, it will perofrm the calculation as many times as it is called. Both are part of the general class interface provided by the base class fvPatch-Field, that can be used to program a customized boundary conditions, derived either directly or indirectly from fvPatchField.

Only a small number boundary conditions in the official release are derived directly from fvPatchField, such as the basic fixedValue-FvPatchField and zeroGradientFvPatchField. Most of the derived
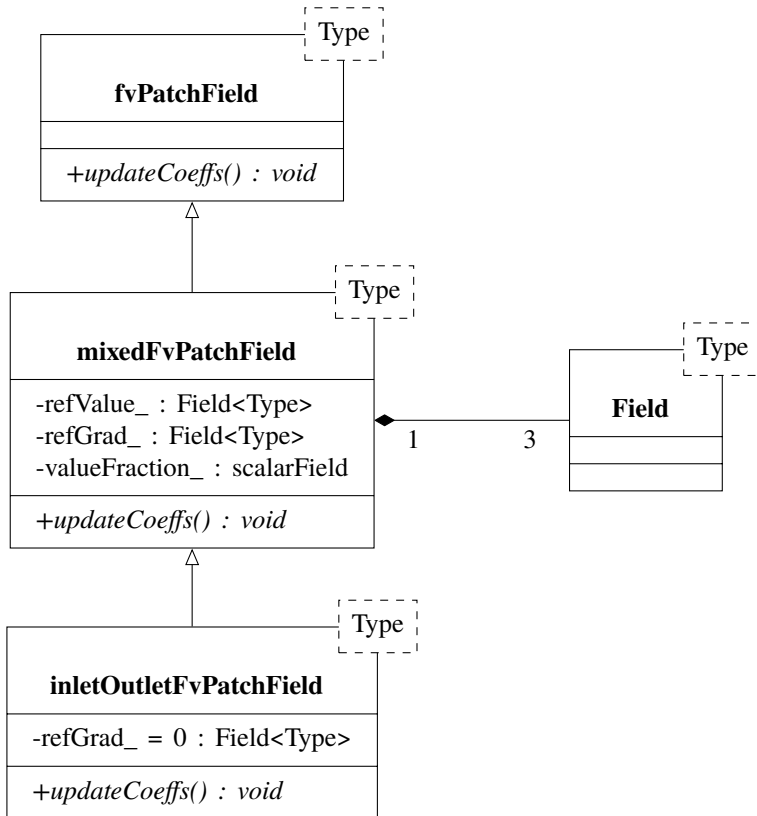
Figure 10.3: Class collaboration diagram for `mixedFvPatchField` and `inletOutletFvPatchField` boundary conditions.

boundary conditions inherit directly from basic boundary conditions. A popular base class is `mixedFvPatchField` which provides the functionality of blending between a user defined fixed value and a fixed gradient boundary condition. The class collaboration diagram of `mixedFvPatch-Field` is shown in figure 10.3. It contains three new private attributes shown also in listing 46, and it does not rely on the implementation of the fixed gradient and fixed value boundary conditions. Instead, the prescribed fixed gradient and fixed value boundary fields are stored as private attributes of type `Field`.

As these attributes are private, there are public member functions that provide const and non-const access to them. This enables the derived

Listing 46 Private attributes of the `mixedFvPatchField` boundary condition.

```
//- Value field
Field<Type> refValue_;

//- Normal gradient field
Field<Type> refGrad_;

//- Fraction (0-1) of value used for boundary condition
scalarField valueFraction_;
```

classes to use the attributes indirectly in order to implement different ways of blending between between the fixed value and zero gradient boundary condition. A commonly used boundary condition that is derived directly from `mixedFvPatchField` is the `inletOutletFvPatch-Field`. It switches between fixed value and zero gradient boundary condition, depending on the direction of the flux. If the flux is pointing out of the domain, it acts just as the zero gradient boundary condition (`zeroGradientFvPatchField`), otherwise it acts as a fixed value boundary condition (`fixedValueFvPatchField`). This is determined on a face-by-face basis, and based on the private attributes of the `mixed-FvPatchField` boundary condition. The gradient of the field is not prescribed by the user as in the `mixedFvPatchField` - it is set to the value of zero by the `inletOutletFvPatchField` constructor.

The fraction value used later by `mixedFvPatchField<Type>::update-Coeffs()` is computed in listing 47 by assigning the value 1 for faces that have a positive (outflow) volumetric flux, and value 0 otherwise. The implementation of the `updateCoeffs()` member function of `in-letOutletFvPatchField` is shown in listing 47. Only the values for the `valueFraction()` are set by this member function, the calculation of the boundary field is delegated to the parent `mixedValueFvPatchField`. The function `pos` is shown in listing 48. It returns 1 if the value of the scalar s is greater or equal than zero and 0 otherwise. The actual assignment of `zeroGradientFvPatchField` and `fixedValueFvPatch-Field` is done by the call to `mixedFvPatchField::updateCoeffs()` and hence must not be re-implemented.

Listing 47 The `updateCoeffs()` member function of the inlet/outlet boundary condition.

```cpp
template<class Type>
void Foam::inletOutletFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const Field<scalar>& phip =
        this->patch().template lookupPatchField
        <
            surfaceScalarField,
            scalar
        >
        (
            phiName_
        );

    this->valueFraction() = 1.0 - pos(phip);

    mixedFvPatchField<Type>::updateCoeffs();
}
```

Listing 48 The `pos` function.

```cpp
inline Scalar pos(const Scalar s)
{
    return (s >= 0)? 1: 0;
}
```

Reading boundary condition data

During the process of programming a custom boundary condition that may have new parameters, corresponding new parameter names and values need to be read from a field file in the 0 directory. Therefore, the programmer must add the necessary parameters and their values in that file. The as the boundary conditions are read from files in the form of dictionaries, the dictionary is read and passed to the boundary condition constructor.

Some boundary conditions may use the `dictionary` class member function that looks up data while providing default values. In that case, switching the type of the boundary condition and not providing the appropriate parameters will not result in a runtime error. Operations on the dictionary class are covered in chapter 5 and should be well understood before proceeding to the next section where programming of a new boundary condition is explained.

10.3 Implementing a new boundary condition

The previous section provided an overview of the implementation of boundary conditions in OpenFOAM. In this section the impelmentation of two new boundary conditions is described.

OpenFOAM provides a large number of different boundary conditions to choose from, and community developments have been done in this part of the code as well, with the most prominent one being the `groovyBC` boundary condition in the `swak4Foam` contribution[1]. Before writing a new boundary condition that fits your requirements, it is wise to take a look if this functionality is already available in the code base or can be modelled by the `groovyBC` boundary condition.

There is quite a large amount of information on how to write your own boundary condition in OpenFOAM already available on the internet. Examples in this chapter have been prepared independent of the already available material. The first example shows how to extend the functionality of any boundary condition in OpenFOAM without modifying it, with a concrete purpose of reducing recirculation at the boundary. The

---

[1]`http://openfoamwiki.net/index.php/Contrib/swak4Foam`

second example shows how develop a new `pointPatchField` that applies a predefined motion to a patch. This predefined motion is calculated from tabulated data, that must be provided by the user. Both examples emphasise the correct use of the class interface fixed by the root abstract base classes for the boundary conditions in OpenFOAM, i.e. the `fvPatchField` and `pointPatchField` classes.

10.3.1 Recirculation control boundary condition

In this example a method is presented how an existing boundary condition can be extended by an additional computation (functionality) during runtime. Imagine a boundary condition that updates the boundary field values in a certain way as a simulation runs. At some point, based on the simulation results, the conditions of the simulation (e.g. a pressure at another boundary) signal that an additional boundary operation is necessary. When this happens, the new extended boundary condition *enables* the additional calculation at runtime, and modifies the boundary field values accordingly.

A good technical example would be a heated closed container filled with an ideal gas. When heat flux enters the container, the pressure in the container rises. An extended boundary condition would measure the pressure at the lid of the container, and open the lid when the pressure reaches a certain value. Numerically speaking, this boundary condition would *change its type* during the simulation, from an impermeable wall, to an outlet boundary condition, based on the pressure at the lid.

In the example presented in this chapter, recirculation is measured at a boundary. When recirculation occurs, an additional calculation modifies the nature of the boundary condition into an *inflow* condition. As a result, the recirculation is *pushed out* by the inflow.

> INFO
>
> The recirculation boundary condition example may not work in a parallel execution, depending on the fact if the modified boundary is a part of the processor domain. The goal of this example is to show how the mesh, the object registry and the fields connect with each other, and not how to parallelize boundary condition updates.

Hypothetically, such an extension could be achieved using inheritance only. However, it would require an extension of each boundary condition in OpenFOAM to account for the RTS-enabled additional calculation that needs to be performed. Extending each boundary condition by using multiple inheritance would result in modifications of the existing boundary conditions, only to take into account *possible extension* (i.e. recirculation control) which may, or may not be used at runtime, depending on the user's choice. Obviously, this is by no means a versatile method and it makes the extension difficult to achieve during runtime without modifying existing code. When a new functionality needs to be added during runtime to an already existing hierarchy of classes without modifying the models of that hierarchy, the object oriented design pattern *Decorator Pattern* can be used. Details on the Decorator Pattern and other OOD patterns are given in the book by [1]. Figure 10.4 illustrates the working principle for a zeroGradient boundary condition, that is decorated with an

delegates calculation when a condition is met

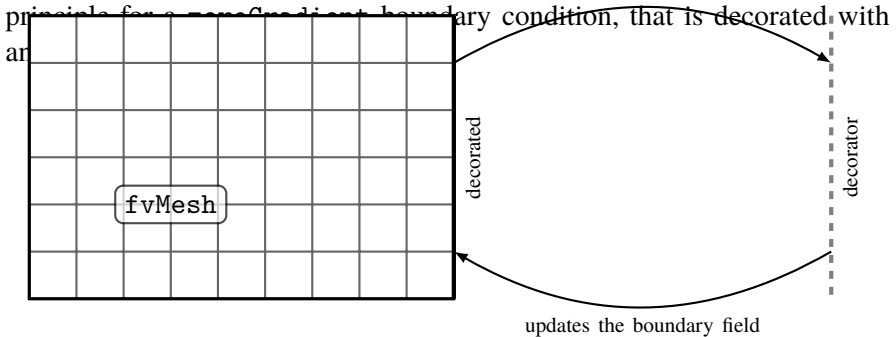decorated

decorator

fvMesh

updates the boundary field

Figure 10.4: Working principle of the decorated boundary condition. The boundary condition operates in a standard way until such conditions are fulfilled that the extended computation is required and delegated to the decorator. At that point, the boundary condition acts as if it switched to the decorator type.
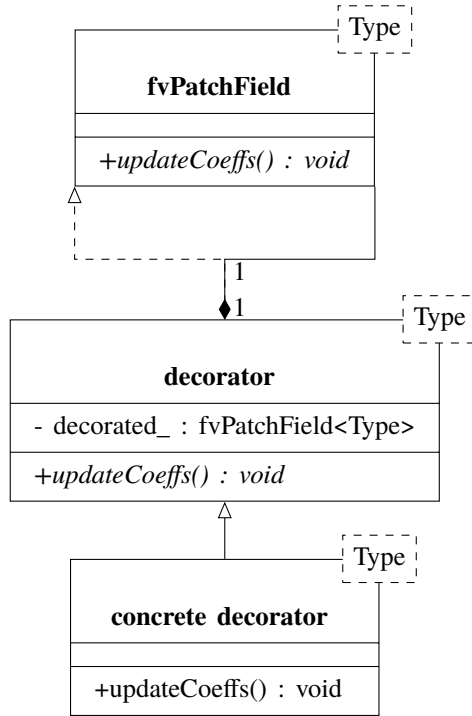
Figure 10.5: Decorator for the boundary conditions in OpenFOAM

Adding functionality to a BC using the Decorator Pattern

A boundary conditition decorator *is a* boundary condition itself, since it modifies the boundary field as well. Therefore, the decorator inherits from the `fvPatchField` abstract base class of all boundary conditions in OpenFOAM. In addition to inheriting from the `fvPatchField`, the decorator composes object of it's own base class (`fvPatchField`).

To clarify things, the Unified Modeling Language (UML) class collaboration diagram of the Decorator Pattern applied on the boundary conditions hierarchy is shown in figure 10.5. As shown on in that diagram, the decorator is a part of the class hierarchy as other boundary conditions. Hence, for boundary conditions in OpenFOAM, imposing such an `is-a` relationship with the abstract base class `fvPatchField` makes the boundary condition decorator act as a boundary condition to the rest of the Open-FOAM client code. This is exactly the same principle as plain inheritance, only with the extension of storing an instance of the object inherited from.

The decorator must implement all the pure virtual methods prescribed by the abstract class `fvPatchField`. As it composites an ordinary boundary condition as well, the decorator willl delegate the function calls to the decorated boundary condition, based on the conditions prescribed by the programmer. The extensions provided by the decorator can be designed as the programmer desires. By combining both inheritance and composition as shown in figure 10.5, it is possible for a decorated boundary condition to switch its own type during runtime, as it delegates the computation to the decorator.

Adding recirculation control to a boundary condition

As an example of adding runtime functionality to any boundary condition in OpenFOAM we have chosen to flow recirculation as the control parameter and inflow velocity as the imposed action of the boundary condition. The flow recirculation is recognized by an alternating sign the volumetric flux at the boundary. Alternating sign in the volumetric flux on a domain boundary signals an that an eddy (vortex) crosses the boundary. Therefore, the fluid flows into the domain over a part of the boundary, and flows out of the over the other boundary part.

For the example simulation case, the boundary condition checks if the flow of the extended boundary has recirculation and tries to control the decorated boundary condition in order to reduce it. This type of boundary condition decorator is most likely to be set for some of the boundary conditions where outflow is expected. This control in this example is done in a fairly straightforward manner: by modifying the decorated boundary condition as such that its field is overwritten with increasing inflow values.

Of course, this example is specific to inflow/outflow situations, but the goal of this example is not to deal with flow control. The recirculation control boundary condition decorator is different than the standard boundary conditions in OpenFOAM. It modifies field values calculated by another boundary condition directly, regardless of the type of boundary condition that is decorated. A finished recirculation control boundary condition is already available in the example code repository. To ease the understanding of the presented example, the description should be followed with the code of the recirculation control boundary condition

from the example code repository opened in a text editor. In the following discussion the same names are used for the files and classes as those present in the example code repository.

> **INFO**
>
> The boundary conditions have to be compiled as shared libraries. They should never be implemented directly into application code, as this limits their usability significantly as well as sharing with other OpenFOAM programmers.

The libraries that the boundary conditions are compiled into are dynamically linked with solver applications during runtime. At the start of the tutorial, the library directory needs to be created:

```
?> mkdir -p primerBoundaryConditions/recirculationControl
?> cd !$
```

and the class files of an existing boundary condition need to be copied, which we will use as skeleton files for the recirculation control boundary condition:

```
?> cp \
$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic/zeroGradient/* .
```

All instances of the string "zeroGradient" are to be renamed into "recirculationControl" in file names as well as in class names. Once the names are modified, exit the `recirculationControl` directory, and create the compilation configuration folder `Make`:

```
?> cd ..
?> $FOAM_SRC/../wmake/scripts/wmakeFilesAndOptions
```

and modify the `Make/files` to take into account that a library is to be compiled, and not an application, so the line

```
EXE = $(FOAM_APPBIN)/primerBoundaryConditions
```

needs to be replaced with the line

```
LIB = $(FOAM_USER_LIBBIN)/libofPrimerBoundaryConditions
```

Note that the script `wmakeFilesAndOptions` will insert all `*.C` files into the `Make/files` file, which means that the line

```
recirculationControl/recirculationControlFvPatchField.C
```

Listing 49 The `Make/files` file of the primerBoundaryConditions library.

```
recirculationControl/recirculationControlFvPatchFields.C

LIB = $(FOAM_USER_LIBBIN)/libofPrimerBoundaryConditions
```

Listing 50 The `Make/options` file of the primerBoundaryConditions library.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

must be removed before compilation, since it containts class template definition for the recirculation control boundary condition. This will cause class re-definition errors during compilation, since the actual boundary condition classes are instantiated from the recirculation control class template for the tensorial properties using the macro

```
makePatchFields(recirculationControl);
```

Once the file `recirculationControlFvPatchField.C` is removed from `Make/files`, the library is ready for the first compilation test. The compilation can be started by issuing

```
?> wmake
```

from within the `primerBoundaryConditions` directory. Listing 49 contains final version of the `Make/files` and listing 50 contains the final version of the `Make/options` file. With the successful compilation, a skeleton implementation of a self-contained shared boundary condition library is created. Before further development of the decorator functionality is started, the boundary condition is to be tested with an actual simulation run. At this point the boundary condition has the same functionality as the `zeroGradientFvPatchField` which it is based upon, but with a different name. It's functionality can be tested by running any simulation case, provided you define the required linked libraries in the `system/controlDict` file:

```
libs ("libofPrimerBoundaryConditions.so")
```

This loads the boundary condition shared library and links it to the OpenFOAM executable.

> **TIP**
>
> Performing such integration tests is a strongly recommended step in the process of writing a custom library. Using a version control system (Chapter 6) improves the workflow as well. As an example, try applying the recirculation control boundary condition to the `fixedWalls` patch of the velocity field in the `cavity` tutorial case using the `icoFoam` solver.

At this point, the integration of the library with OpenFOAM is tested and considered working. The implementation of the design shown in figure 10.5 may begin. The strict object-oriented way of implementing the Decorator Pattern would be to start with an abstract Decorator class for the `fvPatchField`. In that case, the `recirculationControl` must be derived from it as a concrete decorator model. Instead, a concrete decorator is used in this example as a starting point and the abstract implementation is left to the reader as an exercise at the end of the section. Implementing the exercise allows the addition of different functionalities to any boundary condition in OpenFOAM without modifying their implementation. This is an additional benefit of doing the exercise besides improving the understanding of boundary conditions in OpenFOAM.

The first modification to the recirculation control boundary condition is applied to the class template declaration file `recirculationControlFv-PatchField.H`. It inherits from `fvPatchField`, which is done by `ze-roGradientFvPatchField` already. The important parts of `recircu-lationControlFvPatchField`'s class definition are shown in code listings.

In listing 51, the decorator pattern can be seen: the inheritance from `fv-PatchField` is present but also a private attribute of `fvPatchField` is composited by the recirculation control boundary condition class. Instantiating the local copy in `baseTypeTmp_` is done using the smart pointer object `tmp` of OpenFOAM, as it provides additional functionalities such as garbage collection. This smart pointer relies on the RAII C++ idiom and greatly simplifies the handling of pointers. The remaining class attributes that are declared as constant will be read from the boundary

Listing 51 The declaration of the recirculation control boundary condition.

```cpp
template<class Type>
class recirculationControlFvPatchField
:
    public fvPatchField<Type>
{
    protected:

        // Base boundary condition.
        tmp<fvPatchField<Type>> baseTypeTmp_;

        const word applyControl_;
        const word baseTypeName_;
        const word fluxFieldName_;
        const word controlledPatchName_;
        const Type maxValue_;

        scalar recirculationRate_;
```

condition's dictionary. In order to instantiate the boundary condition that should be decorated, the particular name must be passed via the boundary condition's dictionary and is stored in the `baseTypeName_` attribute. To provide an option to either apply the control of the recirculation or not, the `Switch applyControl_` is implemented. If it is false, nothing will be imposed upon the boundary, and so switching it off and simply using the decorated boundary condition is easy. In addition it will report the amount of recirculation on the extended boundary condition. The type of the extended boundary condition is determined by the `baseTypeName_` variable, which is then used to create a concrete extended boundary condition based on a class name. This boundary condition is stored in `baseTypeTmp_`.

In order to compute the recirculation, the boundary condition needs to know the name of the volumetric flux field which is what the `fluxFieldName_` attribute defines. The name of the patch field which is controlled is defined by the `controlledPatchName_`. The maximal value it may assume is defined by the member `maxValue_`, and the `recirculationRate_` stores the percentage of the negative volumetric flux on the extended boundary condition.

Having added the new class attributes, the constructors of `recirculationControlFvPatchField` will have to initialize them. The common

Listing 52 Constructor of the recirculation control boundary condition.

```cpp
template<class Type>
Foam::recirculationControlFvPatchField<Type>::
recirculationControlFvPatchField
(
    const recirculationControlFvPatchField<Type>& ptf,
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fvPatchField<Type>(ptf, p, iF, mapper),
    baseTypeTmp_(),
    applyControl_(ptf.applyControl_),
    baseTypeName_(ptf.baseTypeName_),
    fluxFieldName_(ptf.fluxFieldName_),
    controlledPatchName_(ptf.controlledPatchName_),
    maxValue_(ptf.maxValue_),
    recirculationRate_(ptf.recirculationRate_)
{
    // Instantiate the baseType based on the dictionary entries.
    baseTypeTmp_ = fvPatchField<Type>::New
    (
        ptf.baseTypeTmp_,
        p,
        iF,
        mapper
    );
}
```

approach is to expand and modify the constructor initialization lists to take into account the new private class attributes. To keep the description concise, only the dictionary constructor is shown in the following text. The full implementation is available in the source code repository.

The constructor of the recirculation control boundary condition is shown in listing 52. In the initialization list, `baseTypeTmp_` is not assigned a value and hence takes an arbitrary value. In the constructor itself, a `fvPatchField` is created and assigned to `baseTypeTmp_`, using the `New` selector. This way of constructing objects is called Factory Method (New Selector) and is defined in the abstract `fvPatchField` class. It selects the base class for `recirculationControlFvPatchField` during runtime, based on the name provided in the dictionary `dict` is used by the constructor to initialize the decorated boundary condition. After this constructor has been executed, the protected attributes needed by the control function will be initialized, as will the decorated boundary

Listing 53 Recirculation control boundary condition delegating a member function call.

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::recirculationControlFvPatchField<Type>::valueInternalCoeffs
(
    const Foam::tmp<Foam::Field<scalar>> & f
) const
{
    return baseTypeTmp_->valueInternalCoeffs(f);
}
```

condition.

The remaining constructors of `recirculationControlFvPatchField` must be modified similarly, accounting for the new class attributes. Once the constructors are modified, the member functions listed in figure 10.5 need to be modified as well. They need to delegate the work to the decorated boundary condition, that is stored in `baseTypeTmp_`. The implementation is identical for all four member functions, so only one of them is presented in listing 53.

This kind of delegation needs to be present in all parts of the decorator implementation where the control of the flow *is not performed*. In that case the boundary condition should act as the decorated boundary condition. This is illsutrated as the concrete boundary condition model in figure 10.5.

As the final step of implementing the recirculation control boundary condition, the member function responsible for the boundary condition operation (`updateCoeffs`) needs to be implemented. The implementation is devided into two major parts. The first part checks if the boundary condition has already been updated, which is common practice in Open-FOAM boundary conditions. If this is not so, the flux field is looked up using the user-defined name of the flux field `fluxFieldName_`.

The positive and negative volumetric fluxes are computed by the extended boundary condition as shown in listing 54. Once the positive and negative volumetric fluxes are computed, the recirculation rate (ratio of the negative flux and the total flux) is calculated with the code shown in listing 55.

Listing 54 Computing positive and negative fluxes by the recirculation boundary condition.

```
if (this->updated())
{
    return;
}

typedef GeometricField <Type, fvPatchField, volMesh>  VolumetricField;

// Get the flux field
const Field<scalar>& phip =
this->patch().template lookupPatchField
<
  surfaceScalarField,
  scalar
>(fluxFieldName_);

// Compute the total and the negative volumetric flux.
scalar totalFlux = 0;
scalar negativeFlux = 0;

forAll (phip, I)
{
    totalFlux += mag(phip[I]);

    if (phip[I] < 0)
    {
        negativeFlux += mag(phip[I]);
    }
}
```

In conditions where no recirculation occurs, the recirculation control and the decorated boundary condition behave identically. The modification of the field is thus delegated to the encapsulated concrete boundary condition model. Note that because the boundary condition decorator is itself a boundary condition, each time an update happens, the boundary condition state must be set to up-to-date. As usual the member function `updateCoeffs` of the `fvPatchField` abstract class must be invoked for this purpose. The part of the `updateCoeffs` member function that is responsible for the control of the recirculation is shown in listing 56.

This boundary condition will need a non-constant access to the field of another boundary condition, since it will modify boundary field values. For this reason, it *breaks the encapsulation* of the `objectRegistry` class, by casting away constness of the `VolumetricField` provided by the registry forcefully. The non constant access to the other boundary

---

Listing 55 Computing recirculation rate on the boundary.

```
// Compute recirculation rate.
scalar newRecirculationRate = min
(
  1,
  negativeFlux / (totalFlux + SMALL)
);

Info << "Total flux " <<  totalFlux << endl;
Info << "Recirculation flux " << negativeFlux << endl;
Info << "Recirculation ratio " << newRecirculationRate << endl;

// If there is no recirculation.
if (negativeFlux < SMALL)
{
    // Update the decorated boundary condition.
    baseTypeTmp_->updateCoeffs();
    // Mark the BC updated.
    fvPatchField<Type>::updateCoeffs();
    return;
}
```

---

field is shown in listing 57.

> **INFO**
>
> Casting away constness this way should be avoided wherever possible: it invalidates the point of encapsulation - the object state that can be modified only by the class member functions. This example uses the const cast only to point to a possible collaboration between the geometrical field and the object registry.

If the interface of the `objectRegistry` provides only constant access to registered object, casting away constness and modifying the object state is deceiving the programmer. He/she will not expect the change of the `VolumetricField` object state to be possible via the interface of the `objectRegistry`. Algorithm 1 clarifies the recirculation control algoritm in pseudocode.

This boundary condition is meant primarily as an example of applying the Decorator Pattern to the boundary condition class hierarchy in OpenFOAM. However, the design applied here may very well be used in situations where inlet/outlet boundary conditions are present. In some situations a reduction of recirculation can be achieved by increasing e.g.

the inlet pressure or velocity. Still, note that this is only an example that illustrates two major things: First it shows how the design of the `VolumetricField` may be used to couple functionality between different boundary conditions. Secondly, it illustrates how to program a new boundary condition in OpenFOAM, that is derived from `fvPatchField`.

Testing the recirculation control boundary condition

Once the `updateCoeffs` method has been implemented, the boundary condition is ready to be used. As a test case, a simple backward channel with a backward facing step is used. The recirculation control is applied on the outlet of the channel, with the backward facing step wall as the controlled boundary condition.
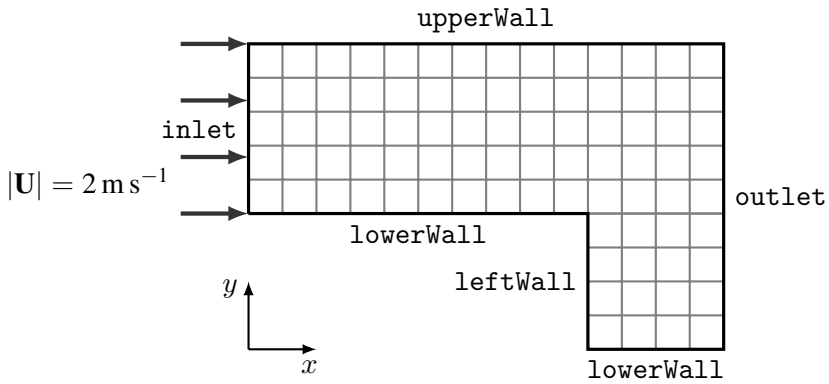

Figure 10.6: Geometrical setup and initial conditions for the recirculation control test case

Figure 10.6 depicts the initial configuration of the flow, where the velocity of the backward step is set to zero as it is an impermeable wall. The recirculation control boundary condition will override the zero velocity of the backward facing step the moment recirculation appears at the outlet. Further on, it will switch the boundary condition applied to the `leftWall` boundary into an inflow, in order to drive the recirculation out.

The comparison of the velocity field without and with the recirculation control boundary condition is shown in figure 10.7.
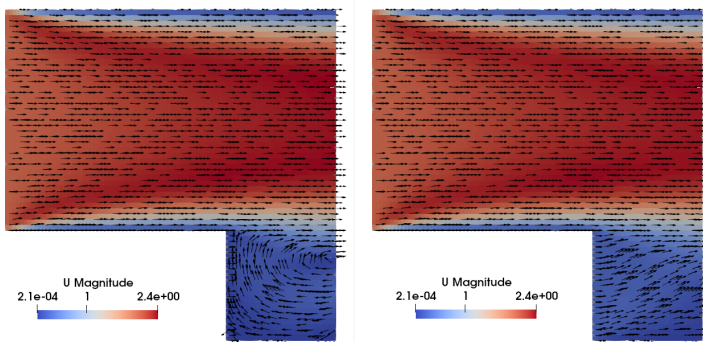
Figure 10.7: Velocity field of the recirculation control test case, without and with recirculation control, respectively with zero-gradient and recirculation outlet.

Listing 56 Recirculation control execution.

```cpp
if (
    (applyControl_ == "yes") &&
    (newRecirculationRate > recirculationRate_)
)
{
    Info << "Executing control..." << endl;

    // Get the name of the internal field.
    const word volFieldName = this->internalField().name();

    // Get access to the regitstry.
    const objectRegistry& db = this->db();

    // Find the GeometricField in the registry using
    // the internal field name.
    const VolumetricField& vfConst =
      db.lookupObject<VolumetricField>(volFieldName);

    // Cast away constness to be able to control
    // other boundary patch fields.
    VolumetricField& vf = const_cast<VolumetricField&>(vfConst);

    // Get the non-const reference to the boundary
    // field of the GeometricField.
    typename VolumetricField::Boundary& bf =
        vf.boundaryFieldRef();

    // Find the controlled boundary patch field using
    // the name defined by the user.
    forAll (bf, patchI)
    {
        // Control the boundary patch field using the recirculation rate.
        const fvPatch& p = bf[patchI].patch();

        if (p.name() == controlledPatchName_)
        {
            if (! bf[patchI].updated())
            {
                // Envoke a standard update first to avoid the field
                // being later overwritten.
                bf[patchI].updateCoeffs();
            }
            // Compute new boundary field values.
            Field<Type> newValues (bf[patchI]);

            scalar maxNewValue = mag(max(newValues));

            if (maxNewValue < SMALL)
            {
                bf[patchI] == 0.1 * maxValue_;
            } else if (maxNewValue < mag(maxValue_))
            {
                // Impose control on the controlled inlet patch field.
                bf[patchI] == newValues * 1.01;
            }
        }
    }
}
```

Listing 57 Getting non-const access to another boundary field, from the recirculation boundary condition.

```
// Cast away constness to be able to control other boundary patch fields.
VolumetricField& vf = const_cast<VolumetricField&>(vfConst);

// Get the non-const reference to the boundary field of the GeometricField.
typename VolumetricField::GeometricBoundaryField& bf = vf.boundaryField();
```

Algorithm 1 Recirculation control algorithm.

---

if control is applied **and** recirculation is increasing **then**
    get the name of the controlled field
    get access to object registry
    find the `VolumetricField` in the object registry
    cast away constness of the `VolumetricField`
    for boundary conditions **do**
        if controlled boundary condition found **then**
            compute the new values
            if new values are zero **then**
                set new values to 10% of the maximum
            else if new values are positive and smaller than prescribed maximum **then**
                increase the old values by 1 %
            end if
        end if
    end for
end if

10.3.2 Mesh motion boundary condition

This sub-section illustrates the construction of a new boundary condition used for the motion of the mesh. The motion of the mesh relies on displacements or velocities defined for the points of the mesh in a form of a boundary condition that is associated to the boundary points that belong to the mesh boundary, a so-called `pointPatchField` in OpenFOAM. Unlike the boundary conditions that are based on the `fvPatchField`, `pointPatchField`-type boundary conditions do not store the boundary values in a boundary field, they are used to modify the values of the internal field. Altering the internal field values and any other operations are handled by either the solver or other classes using these boundary conditions. Vector quantities used by the mesh motion boundary conditions will either define a velocity or a displacement of the particular mesh point, depending on the choice of the mesh motion solver.

In this section, a mesh motion solver will use the `dynamicFvMesh` library and in a simulation example, it will solve a Laplace's equation for the point displacement with the displacement at the boundary defined by the new boundary condition. Since the Laplace's equation is used to model diffusive transport, the displacement prescribed at the mesh boundary will be smoothly diffused to the surrounding mesh, which ensures higher quality of the deformed cells. For this type of application, the field that stores the deformation of the points is called `pointDisplacement`.

The boundary condition presented in this section reads the position and orientation of the patch's centre of gravity from an input file and applies the displacement to the mesh boundary, with respect to its previous position and it must be used in conjunction with dynamic meshes, otherwise the `pointDisplacement` field will not be read. The boundary condition functionality consists of two existing components of OpenFOAM:

1. Computing the position of the patch's centre of gravity (COG), which is calculated based on the prescribed motion that is contained in a dictionary. This prescribed motion must be present in tabulated form and gets interpolated linearly between each of the data points. All of this is already implemented in the `tabulated6DoFMotion`, which is a `dynamicFvMesh` class that moves the entire mesh based on the prescribed motion.

2. Assigning vectorial values to a `pointPatchField`. One example for this type of boundary condition is the `oscillatingDisplacementPointPatchVectorField`.

As for boundary conditions derived from `fvPatchField`, only values on the domain boundary are changed. No additional changes to the simulation or field are performed by any boundary condition and the functionalities are encapsulated in a logical manner. For `fvPatchField` boundary conditions, the field variables are calculated by the flow solver, using the boundary values. The same principle applies to boundary conditions derived from `pointPatchField`. Only the velocity or displacement is prescribed by the boundary condition, whereas the actual mesh changes are performed by a dedicated mesh motion solver that is a part of the `dynamicFvMesh` library. In the following, both components are briefly described and the parts that are relevant for the new boundary condition are highlighted.

Reading the motion data

A brief search in the existing code base of OpenFOAM reveals that there is a mesh motion solver that reads the motion data from a tabulated file, similar to what is planned for this boundary condition. However, in that case, the same displacement is applied to all mesh points, resulting in a mesh motion which moves the mesh as a rigid body: no mesh deformation occurs and the relative position of the mesh points does not change. For the purpose of this tutorial, the calculation of motion can be used and later be applied to the patch points making the boundary of the mesh move as a rigid body. This kind of mesh motion may be beneficial when the relative motion of the body is small with respect to the mesh (flow domain). In this case, the if the motion is propagated into the flow domain in a diffusion-like manor, the motion of the mesh far away from the boundary with prescribed displacement will be near zero. How fast the displacement vanishes away from the body is determined by the magnitude of the displacement diffusion coefficient.

The code implementing the rigid body mesh motion based on tabular data is contained within the `tabulated6DoFMotionFvMesh` class which is derived from `solidBodyMotionFunction` and can be found here:

```
$FOAM_SRC/dynamicMesh/motionSolvers/displacement/solidBody/\
solidBodyMotionFunctions/tabulated6DoFMotion/
```

There is an example case in the official release that employs the `tabu-lated6DoFMotion` to prescribe the motion of a closed tank. This tutorial can be found here:

```
$FOAM_TUTORIALS/multiphase/interFoam/laminar/sloshingTank3D6DoF
```

The dictionary containing the motion data (points in time) is set up as a list (using the `List` data structure) which is consisted of translation and rotation vectors in time t. An example for this data can be found in `constant/6DoF.dat` located in the tutorial mentioned above. The snippet below illustrates the principle of the way that dictionary is set up.

```
(
(t1 ((Translation_Vector_1) (Rot_Vector_1))
(t2 ((Translation_Vector_2) (Rot_Vector_2))
...
)
```

A spline based interpolation is performed to obtain position and orientation data between the data points of the dictionary. The translation and rotation vectors are defined with respect to the original coordinate system and as shown in figure 10.8.

As chapter 13 deals solely with dynamic meshes, only a brief summary of the working principles of dynamic meshes of type `solidBodyMotionFvMesh` is provided for better clarity:

- dynamic meshes that are derived from `solidBodyMotionFvMesh` deal only with the motion of solid bodies,
- no topological changes can be performed, nor is a solid body patch supposed to change it's shape,
- the motion itself is not defined by the `solidBodyMotionFvMesh`, that is what the `solidBodyMotionFunction` and derived classes are for,
- this simplyfies seperating the caluclation of motion from the actual mesh motion algorithm,
- the `tabulatedSixDoF` class inherits from `solidBodyMotionFunction` which is the base class for all solid body motion functions in OpenFOAM.
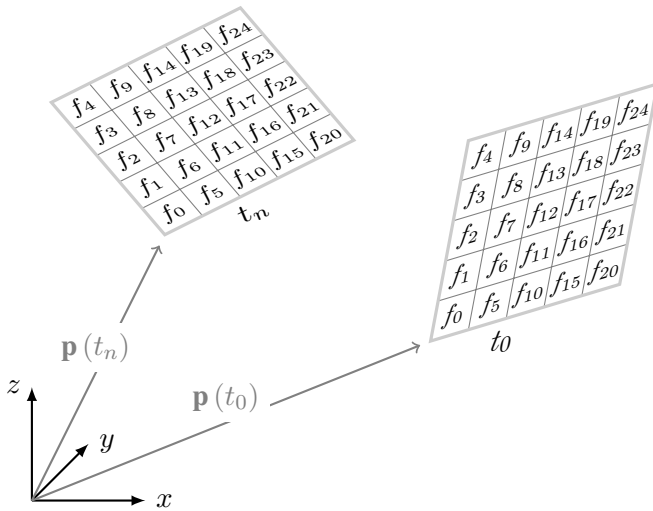
Figure 10.8: Illustration of an example patch moved from position at t_0 to the position at t_n with respect to the global coordinate system

The `solidBodyMotionFvMesh` is instantiated during construction of the `dynamicFvMesh` using runtime selection, if a dynamic mesh of type `solidBodyMotionFvMesh` is selected in the `dynamicMeshDict`. The particular parameters are read from a sub-dictionary of the `constant/dynamicMeshDict`, which is a private class attribute referred to as `SBMFCoeffs_`. The relevant parts of the program code for this boundary condition are described in the following text.

Reading the tabulated data from an input dictionary is shown in listing 58. The data and the filename are read from the sub-dictionary of the `dynamicMeshDict` into a list of type `Tuple2`. The `Tuple2` class is a data structure that stores two objects that can be of different type. In the above example, instances of a `scalar` and a `translationRotationVectors` are stored in the `Tuple2`, with the first one being the time and the second a nested vector, one for translation and one for rotation. Hence, the contents of the motion file are stored in that data structure directly. To provide easy access to that data, separate lists are created, as shown in the last half of the above code snippet.

Calculation of the transformation from the input data is performed by the public member function `transformation()` and the relevant content is

Listing 58 Reading tabulated data from an input dictionary.

```
fileName newTimeDataFileName
(
    fileName(SBMFCoeffs_.lookup("timeDataFileName")).expand()
);
IFstream dataStream(timeDataFileName_);
List<Tuple2<scalar, translationRotationVectors> > timeValues
(
    dataStream
);

times_.setSize(timeValues.size());
values_.setSize(timeValues.size());

forAll(timeValues, i)
{
    times_[i] = timeValues[i].first();
    values_[i] = timeValues[i].second();
}
```

Listing 59 Transformation member function of the tabulated motion boundary condition.

```
scalar t = time_.value();
// -- Some lines were spared --
translationRotationVectors TRV = interpolateSplineXY
(
    t,
    times_,
    values_
);

// Convert the rotational motion from deg to rad
TRV[1] *= pi/180.0;

quaternion R(TRV[1].x(), TRV[1].y(), TRV[1].z());
septernion TR(septernion(CofG_ + TRV[0])*R*septernion(-CofG_));
```

shown in listing 59. The second assignment interpolating the position and orientation data for the current time t. All angles must be converted to radians and finally a representation of the transformation is assembled, using quaternions and septernions.

Constructing an object is of course done by means of the constructor, that takes two arguments as shown in the listing below. The first one is a reference to the dictionary that contains the data required by tabu-

`lated6DoFMotion`, which is the path to the data file. Passing a reference to `Time` simplifies the interpolation between the data points:

```
tabulated6DoFMotion
(
    const dictionary& SBMFCoeffs,
    const Time& runTime
);
```

After having found the code responsible for the point motion, the next step is to find a boundary condition that is derived from `pointPatchField` and performs a similar task than the one we plan to implement: from this we plan build our own boundary condition.

Adapting an Existing Boundary Condition

The already existing `oscillatingDisplacementPointPatchVector-Field` boundary condition is a good starting point to derive the mesh motion boundary condition from. It applies displacement values according to a time dependent sinusoid to each of the *values* stored in the points on the boundary. The source code of `oscillatingDisplacementPoint-PatchVectorField` can be found in a subdirectory of `fvMotionSolver`:

```
?> $FOAM_SRC/fvMotionSolver/pointPatchFields/\
      derived/oscillatingDisplacement/
```

As discussed in the beginning of this chapter with regard to `fvPatch-Field`-type boundary conditions, the actual functionalilty of boundary conditions is implemented in either `evaluate()` or `updateCoeffs()` member functions. In case of the `oscillatingDisplacementPoint-PatchVectorField` boundary condition, it is the member function up-dateCoeffs() that calculates the displacement vector for each point of the boundary mesh. The displacement vector is defined like this:

```
amplitude_*sin(omega_*t.value())
```

with both `amplitude_` and `omega_` being scalar values (omega is the angular rotation) read from a dictionary. The implementation of this method can be performed as shon in listing 60.

The line

```
Field<vector>::operator=(amplitude_*sin(omega_*t.value()));
```

Listing 60 Computing the displacement vector for the oscillating displacemenet boundary condition.

```
void oscillatingDisplacementPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {

        const polyMesh& mesh =
            this->dimensionedInternalField().mesh()();

        const Time& t = mesh.time();

        Field<vector>::operator=(amplitude_*sin(omega_*t.value()));

        fixedValuePointPatchField<vector>::updateCoeffs();
    }
    fixedValuePointPatchField<vector>::updateCoeffs();
}
```

of the `updateCoeffs()` method uses the assignment operator of `Field` to assign the proper displacement values to the boundary points of the mesh. This instruction is followed by a call to `updateCoeffs()` of the parent class. By assigning the displacement of the patch to the patch points, the dynamic mesh solver takes care of the actual mesh motion.

Assembling the Boundary Condition

A working version of this boundary condition can be found in the example code repository distributed along with the book, bundled together with the recirculation control boundary condition of the previous chapter into the library `primerBoundaryConditions`. For easier understading, you may want to have the source code of the ready-to-use mesh motion boundary condition open in your text editor while following the steps described here. Similar to other programming examples the boundary contion of this example should be compiled into a dynamic library using `wmake libso`. As usual the first step is to create a new directory to store the boundary condition in:

```
?> mkdir -p \
    $WM_PROJECT_USER_DIR/applications/tabulatedRigidBodyDisplacement
?> cd $WM_PROJECT_USER_DIR/applications/tabulatedRigidBodyDisplacement
```

The next step is to copy the `oscillatingDisplacementPointPatchVec-torField` to the new directory:

```
?> cp $FOAM_SRC/fvMotionSolver/pointPatchFields/\
    derived/oscillatingDisplacement/* .
```

To keep the `tabulatedRigidBodyDisplacement` boundary condition named properly, all occurrences of `oscillatingDisplacement` must be replaced by `tabulatedRigidBodyDisplacement`. This applies both for the filenames as well as matches inside the source files themselves. After deleting `*.dep` files the remaining `C` and `H` files have to be renamed accordingly.

```
?> rm *.dep
?> mv oscillatingDisplacementPointPatchVectorField.H \
    tabulatedRigidBodyDisplacement.H
?> mv oscillatingDisplacementPointPatchVectorField.C \
    tabulatedRigidBodyDisplacement.C
?> sed -i "s/oscillating/tabulatedRigidBody/g" *.[HC]
```

The first thing to check is if the renamed `oscillatingDisplace-mentPointPatchVectorField` still compiles properly. To check this the OpenFOAM typical `Make/files` and `Make/options` files need to be created:

```
?> mkdir Make
?> touch Make/files
?> touch Make/options
```

The content of `files` is short, as the boundary condition consists of only one source file:

```
tabulatedRigidBodyDisplacementPointPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libtabulatedRigidBodyDisplacement
```

For sake of simplicity, all example boundary conditions provided in the code repository are compiled into one library. This library's name differs from the one defined in the above code snippet.

This source file has plenty of dependencies, though. Various other libraries and their header files have to be linked to this library:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/fileFormats/lnInclude
```

```
LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -lfileFormats
```

Test if the boundary condition still compiles, by issuing a `wmake libso` in the directory that contains the `Make` folder. If everything compiles without errors and warnings, open both the header and the source file in an editor of your choice and apply the following changes. First, header files have to be included in `tabulatedRigidBodyDisplacementPoint-PatchVectorField.H`:

```
#include "fixedValuePointPatchField.H"
#include "solidBodyMotionFunction.H"
```

And the source file must include some more header files:

```
#include "tabulatedRigidBodyDisplacementPointPatchVectorField.H"
#include "pointPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "Time.H"
#include "fvMesh.H"
#include "IFstream.H"
#include "transformField.H"
```

The actual implementation of the desired functionality is done in the `updateCoeffs()` member function. Although one private attribute is used there that is not implemented, yet: a constant dictionary that contains all data defined in the boundary condition's dictionary in the `0/` directory. This dicitonary is added to the private attributes in the header file:

```
//- Store the contents of the boundary condition's dicitonary
const dictionary dict_;
```

Each of the boundary condition's constructors must initialize the new private attribute, which is usually done by calling the null-constructor of `dicitonary`. An example is the constructor that constructs the boundary condition from a `pointPatch` and a `DimensionedField`, and it is shown in listing 61. In case the boundary condition is constructed by reading the particular file from the `0` directory, the following constructor is called. As a matter of fact, the dicitonary of the boundary condition is passed to the constructor and it must be stored in the boundary condition for later processing:

Listing 61 Constructor of the point displacement boundary condition.

```
tabulatedRigidBodyDisplacementPointPatchVectorField::
tabulatedRigidBodyDisplacementPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(p, iF),
    dict_()
{}
```

```
tabulatedRigidBodyDisplacementPointPatchVectorField::
tabulatedRigidBodyDisplacementPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchField<vector>(p, iF, dict),
    dict_(dict)
{
    updateCoeffs();
}
```

This constructor is the only one that calls `updateCoeffs()` during construction. The `updateCoeffs` member function is shown in listing 62 The most important line is the line that defines the `SBMFPtr`. It constructs an `autoPtr` for a `solidBodyMotionFunction`, based on a `dictionary` and an object of `Time`. As this code originates from the `dynamicFvMesh` library, the dictionary passed to the constructor is a subdictionary of the `dynamicMeshDict`. This subdicitonary contains all the parameters required by the `solidBodyMotionFunction` selected by the user in the `dynamicMeshDict`. As the definition of the motion parameters for this boundary condition should be performed on a per-boundary basis and not a global-basis, the dictionary that is passed to the constructor should be read from the boundary condition in the 0 directory, rather than the `dynamicMeshDict`. This is what the private member `dict_` is for: It is read only once and passed to the `solidBodyMotionFunction` in each call to `updateCoeffs()`.

The next lines are similar to the ones that can be found in the `tabulated6DoFMotion`. The absolute positions of the patch points after

Listing 62 The updateCoeffs member function of the tabulated rigid body motion boundary condition.

```cpp
void tabulatedRigidBodyDisplacementPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh = this->dimensionedInternalField().mesh()();
    const Time& t = mesh.time();
    const pointPatch& ptPatch = this->patch();

    autoPtr<solidBodyMotionFunction> SBMFPtr
    (
        solidBodyMotionFunction::New(dict_, t)
    );

    pointField vectorIO(mesh.points().size(),vector::zero);

    vectorIO = transform
    (
        SBMFPtr().transformation(),
        ptPatch.localPoints()
    );

    Field<vector>::operator=
    (
        vectorIO-ptPatch.localPoints()
    );

    fixedValuePointPatchField<vector>::updateCoeffs();
}
```

the transformation is applied is stored in `vectorIO`. As the actual motion is relative to the previous point positions, the difference must be computed which is done directly in the call to the assignment operator The transformation is performed using septernions, which are superior to transformation matrices in many regards.

Now that the source code has been prepared, compile the library again.

```
?> wclean
?> wmake libso
```

Executing the Simulation in the Example Case

The execution of the simulation in the example case should be done at first with the prepared and tested code from the example code repository. Once the distributed code is run, the neccessary input parameters in the dictionary, as well as the modifications applied to the fields should be clear and the implemented boundary condition can be tested. The example case `tabulatedMotionObject` is located in the example case repository. This three dimensional case is a simple demonstration of the functionality of the newly implemented tabular rigid body motion boundary condition. Figure 10.9 shows a sketch of the case setup, including a cubic domain with a volume cut out of the middle. The boundary created by this cutting process is represented by the `movingObject` patch.

The `movingObject` boundary will be displaced by the new boundary condition according to the data contained in the 6DoF.dat file in constant/. Compared to a basic OpenFOAM case, there are new configuration files that re required to execute the new boundary condition: The `0/pointDisplacement` and `constant/dynamicMeshDict` files. Initial boundary values are set for the `movingObject` patch of the point-Displacement field using the new boundary condition, as shown in listing 63. In order to use the boundary condition, the instruction to dynamically load the new library at runtime needs to be added to the `system/controlDict` configuration file. Please note, if you used the code presented in this section to compile the boundary condition, the library is named differently ("libofPrimerBoundaryConditions.so").

The `dynamicMeshDict` is needed to control both the mesh motion solver. For this case the `displacementLaplacian` smoother is used, which is based on Laplace's equation and smooths the point displacements. If the
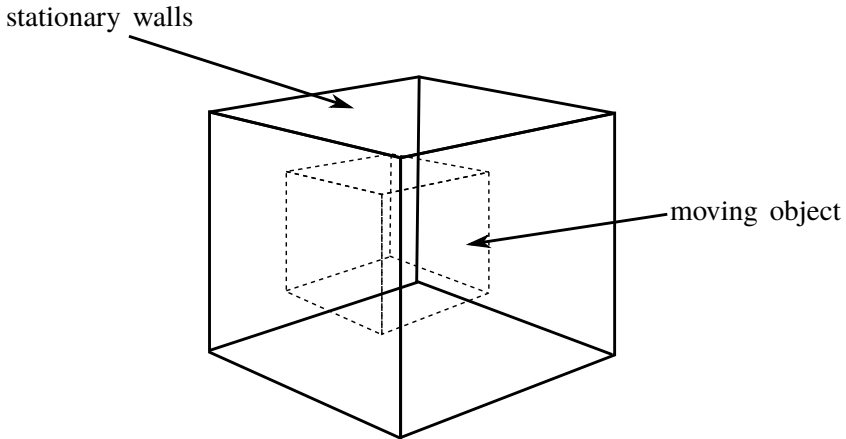
stationary walls

Figure 10.9: Illustration of the domain and moving patch of the tabulat-
edMotionObject example case.

Listing 63 The `movingObject` dictionary.

```
movingObject
{
    type            tabulatedRigidBodyDisplacement;
    value           uniform (0 0 0);
    solidBodyMotionFunction tabulated6DoFMotion;
    tabulated6DoFMotionCoeffs
    {
        CofG            ( 0 0 0 );
        timeDataFileName    "constant/6DoF.dat";
    }
}
```

---

surrounding points did not move with the patch, proximal cell would be
quickly deformed and crushed. The contents of the `dynamicMeshDict`
are shown in listing 64.

The simulation can be run by executing the `Allrun` script within the
simulation case directory. This script performs all necessary steps, such as
mesh generation, to complete this example hassle-free. The utility solver
`moveDynamicMesh` only calls dynamic mesh routines and is devoid of
any flow related computations, which makes it relatively fast compared
to the usual flow solvers with dynamic mesh capabilities. In addition to
performing mesh motion operations it executes numerious mesh related

---

Listing 64 The `dynamicMeshDict` for the tabulated motion example.

```
    dynamicFvMesh      dynamicMotionSolverFvMesh;

    motionSolverLibs ("libfvMotionSolvers.so");

    solver          displacementLaplacian;

    displacementLaplacianCoeffs
    {
        diffusivity     inverseDistance (floatingObject);
    }
}
```

---

quality checks.

Post-processing the case is straightforward and can be performed visually: the case can be inspected in Paraview. Animating the case shows the internal patch tumbling and swaying around within the domain, similar to the data provided in the input file. Cutting the domain in half using the `Clip` filter with the option `Crickle Clip`[2] clarifies some interesting things with regards to dynamic meshes: The way the point displacement emposed by the boundary condition is dissipated by the mesh motion solver. The Laplace equation dissipates the displacement to internal mesh points, moves the points accordingly and maintains good cell quality while the patch translates and rotates. Figure 10.10 holds the image of the final patch position before and after transformation.

Summary

In this chapter the design and implementation of boundary conditions for the FV method and mesh motion in OpenFOAM was described. Both families of boundary conditions are modelled as two separated class hierarchies, `fvPatchField` and `pointPatchField`, respectively. Dynamic polymorphism allows different extensions and combinations of existing implementations. Having boundary conditions implemented as a class hierarchy in conjunction with the fields stored in the internal mesh points allows OpenFOAM library to determine the type of boundary condition during runtime, without recompiling the code each time a new boundary condition is set for a field. The mechanism for loading of dynamic

---

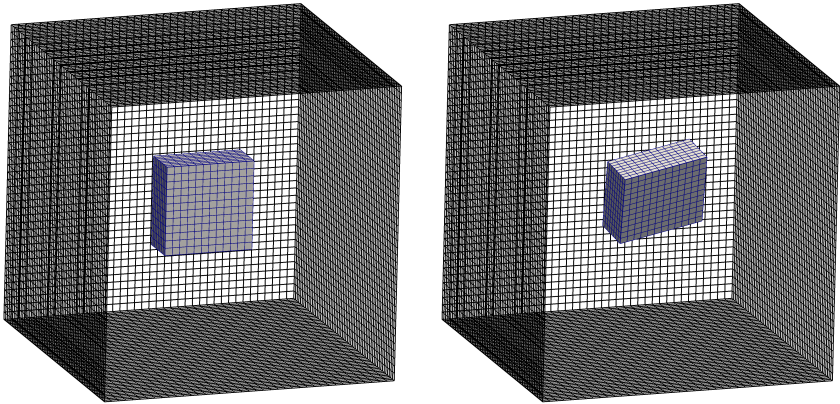[2]Available in ParaView version 4.0.0 or higher.

Figure 10.10: Illustration `tabulatedMotionObject` case before and after patch transformation.

libraries allows new implementations of boundary conditions being compiled into separate libraries (to e.g. the `finiteVolume` library). This in turn results in a straightforward way to add new boundary conditions, without having to re-compile the numerical library. These advantages allow the programmer to extend the boundary conditions, and other parts of OpenFOAM, by developing (and sharing) self-sustained libraries.

Developing new boundary conditions involves finding a point of entry in the class hierarchy, from which hierarchy can be extended. The choice of class as a starting point for further developments for a boundary condition depends on the functionality the desired boundary condition implements, and if a similar boundary condition already exists. The computational part the boundary conditions is located in the `updateCoeffs` member function. Apart from renaming the source files appropriately and adapting the new class for inheritance, the programmer will most likely do most of his work implementing the body of this function.

Further reading

[1]   Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[2]   *OpenFOAM User Guide*. OpenCFD limited. 2016.

11

Transport models

This chapter covers the implementation of viscosity models in Open-
FOAM, other transport models in OpenFOAM have a similar structure
and can be developed further based on the information presented in this
chapter.

11.1 Numerical background

Viscosity models for various kinds of fluids are available in OpenFOAM,
so this section provides a brief introduction to viscosity models' physical
and numerical aspects. The implementation of viscosity models is covered
in section 11.2. More information on viscosity is available in [3, 4], and
similar fluid dynamics textbooks.

A flow configuration often used for describing the viscosity of fluids is
the so-called *Couette flow*: the flow between two parallel plates separated
with a distance d. The lower plane is fixed in space and does not move,
whereas the upper plane moves with a constant velocity **u**.

A velocity gradient between both plates develops over time, as outlined
in figure 11.1, resulting in the stress τ, that acts upon the upper plate
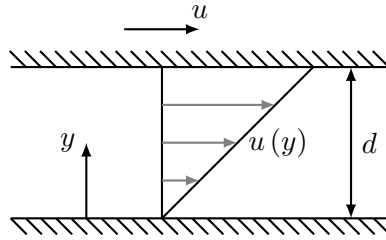that [3], namely

$$\tau = \mu \frac{du}{dy}, \tag{11.1}$$

Figure 11.1: Couette flow with velocity distribution

with μ denoting the dynamic viscosity that relates to the *kinematic* viscosity ν in the following manner:

$$\nu = \frac{\mu}{\rho}. \tag{11.2}$$

The kinematic viscosity ν is the fluid property specified by the Open-FOAM user. Most fluids can are Newtonian: in Newtonian fluids, the viscous stresses correlate linearly with the strain rate (velocity gradient). Non-Newtonian fluids exhibit non-linear stress-strain models. To facilitate simulations of a wide variety of fluids, OpenFOAM includes various viscosity models along with the standard Newtonian model:

Newtonian for incompressible Newtonian fluids with $\nu = const$,
BirdCarreau for incompressible Bird-Carreau non-Newtonian fluids,
CrossPowerLaw for incompressible Cross-Power law that is based non-Newtonian fluids,
Casson for Casson non-Newtonian fluids,
HerschelBulkley for Herschel-Bulkley non-Newtonian fluids,
powerLaw for power-law based non-Newtonian fluids,
Arrhenius where viscosity is a function of some other scalar, typically temperature.

The viscosity model is not the only model that affects the viscosity: the turbulence model changes the so-called *effective* viscosity (cf. chapter 7). If turbulence modeling is disabled, the viscosity model exclusively determines ν.

11.2 Software design

The OpenFOAM transport model library consists of two major components: transport models and viscosity models. The viscosity models provide straightforward object-oriented access to viscosity related calculations and data. This section covers the class relationship for viscosity models in OpenFOAM using simplified UML diagrams for the major classes, as shown in figure 11.2.

The base class is called `transportModel` and inherits from `IOobject`. The `transportModel` class is initialized in the constructor to read the `constant/transportProperties` dictionary of the respective OpenFOAM case. Listing 65 shows the constructor of `transportModel` and the initialization of the `IOobject`:
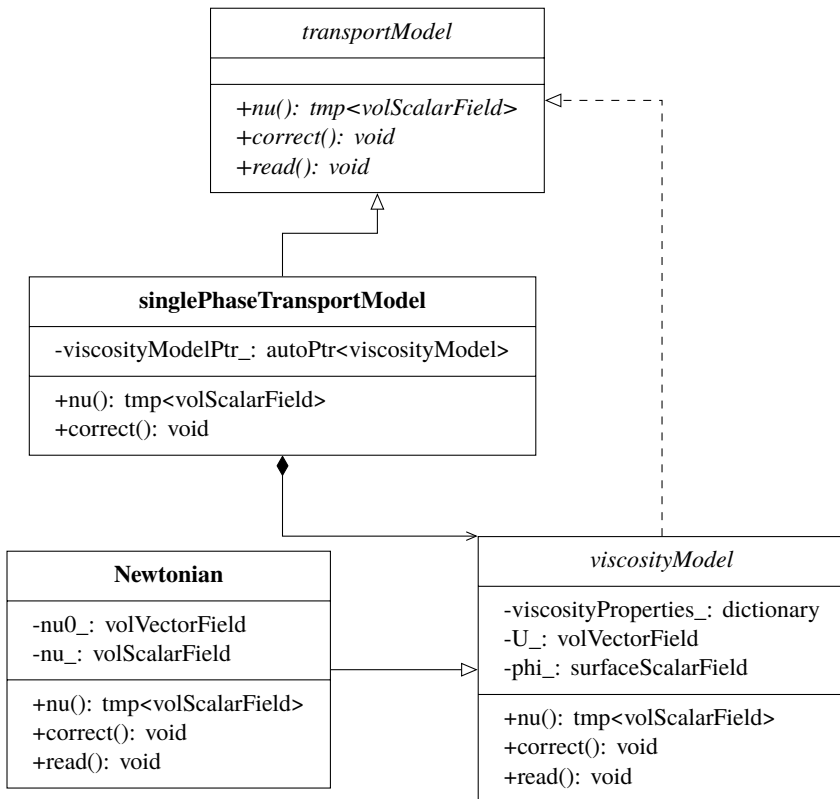

Figure 11.2: Class collaboration diagram for the transport models.

Listing 65 Constructor of `transportModel`

```
Foam::transportModel::transportModel
(
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    IOdictionary
    (
        IOobject
        (
            "transportProperties",
            U.time().constant(),
            U.db(),
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE
        )
    )
{}
```

The implementation of transport models separates models for single-phase flows from models for two-phase flows into two derived classes of the `transportModel` class: `singlePhaseTransportModel` and `incompressibleTwoPhaseMixture`. This division is due to the natural differences between single-phase and two-phase flows.

In OpenFOAM, all single phase solvers employ the `singlePhaseTransportModel` class to gain access and load the kinematic viscosity nu. This parameter must be provided by the user in the `constant/transportProperties` dictionary. The `singlePhaseTransportModel` and other transport models delegate the process of reading and updating this dictionary, which helps keep the code clean. Within the single-phase solvers, the `singlePhaseTransportModel` is instantiated as follows:

```
singlePhaseTransportModel laminarTransport(U, phi);
```

Note that the constructor of the turbulence model requires the `laminarTransport` object as an argument because a turbulence model will require access to the *laminar* viscosity, that in turn is described by the `singlePhaseTransportModel`.

Both, `transportModel` and `singlePhaseTransportModel` define a public member function `nu()` which returns the viscosity as a `volScalarField`. Because `transportModel` is an abstract base class, it does not

implement this member function whereas the `singlePhaseTransport-Model` class must implement this member function (see figure 11.2). It's implementation, however, delegates the functionality to the `viscosity-Model`:

```
Foam::tmp<Foam::volScalarField>
Foam::singlePhaseTransportModel::nu() const
{
    return viscosityModelPtr_->nu();
}
```

The `viscosityModelPtr_` is a private member of `singlePhaseTransportModel` and is defined as an `autoPtr` to a `viscosityModel`. This class attribute is defined in the header file as:

```
autoPtr<viscosityModel> viscosityModelPtr_;
```

For now, it is only important that the `viscosityModel` somehow determines the kinematic viscosity. Its actual implementation and how the `viscosityModel` is selected are both discussed below.

The two phase transport model is modelled by the `incompressibleTwoPhaseMixture` class, used by the `interFoam`-type solvers in OpenFOAM that do not involve phase change. In a similar manner to most single phase solvers, `interFoam` type solvers instantiate an object of that class in the following way:

```
incompressibleTwoPhaseMixture twoPhaseProperties(U, phi);
```

The class collaboration for the incompressible two-phase mixture model is shown in figure 11.3. This instantiation is similar to the approach thas has been used for the `singlePhaseTransportModel`, however, the `incompressibleTwoPhaseMixture` class stores additional data. Rather than having one `viscosityModel`, two are instantiated and stored by that class to account for each fluid phase.

The class definition of `incompressibleTwoPhaseMixture` is shown in listing 66. Using an `autoPtr` to store objects of any derived classes of `viscosityModel` is mandatory, because there are several different classes for different fluid types, that are derived from `viscosityModel`. Each of them is run-time selectable and finally determines what kind of fluid is selected, which in turn defines the viscosity and hence the return value of the public member function nu. To differentiate between both fluid phases, a new `volScalarField` is introduced: `alpha1_`. This field is used in the VoF member function and is effectively a blending value
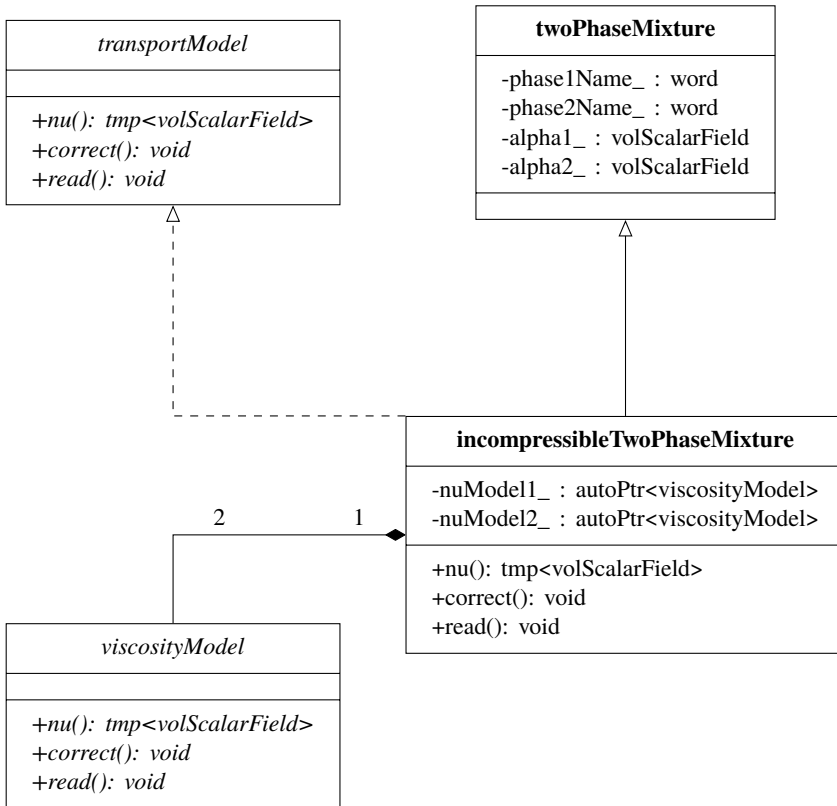
Figure 11.3: Class collaboration diagram for the incompressible two phase mixture model.

between both phases' properties such as viscosity and density. Unlike the `singlePhaseTransportModel`, where a call to the public member function nu is delegated directly to the chosen `viscosityModel`, the return value of this public member function is composed differently. A copy of the private member `nu_` is returned, which is calculated based on both phases' `viscosityModels` and the current `alpha1_` field. This calculation is performed by the private member `calcNu` and triggered when the public member function `correct` is called as shown in listing 67.

Both `viscosityModels` are updated and the volume fraction `alpha1_` is limited to be bounded between 0 and 1. Finally, the kinematic viscosity is calculated using the dynamic viscosity `mu` and density distribution. The latter is computed using the mixture's rule in terms of each phase's

---

Listing 66 Class definition of `incompressibleTwoPhaseMixture`

```cpp
class incompressibleTwoPhaseMixture
:
    public transportModel,
    public twoPhaseMixture
{
protected:

    // Protected data

        autoPtr<viscosityModel> nuModel1_;
        autoPtr<viscosityModel> nuModel2_;

        dimensionedScalar rho1_;
        dimensionedScalar rho2_;

        const volVectorField& U_;
        const surfaceScalarField& phi_;

        volScalarField nu_;
```

Listing 67 Calculation of cell centered viscosity for a two phase mixture.

```cpp
void Foam::incompressibleTwoPhaseMixture::calcNu()
{
    nuModel1_->correct();
    nuModel2_->correct();

    const volScalarField limitedAlpha1
    (
        "limitedAlpha1",
        min(max(alpha1_, scalar(0)), scalar(1))
    );

    // Average kinematic viscosity calculated from dynamic viscosity
    nu_ = mu()/(limitedAlpha1*rho1_ + (scalar(1) - limitedAlpha1)*rho2_);
}
```

---

density and the volume fraction `alpha1_`. The dynamic viscosity `mu` is calculated similarly, using a bounded `alpha1_` field and the kinematic viscosities `nu` of each `viscosityModel`.

Up to this point, the focus was placed on the `transportModel` and its derivatives, while `viscosityModels` were only briefly mentioned. The `viscosityModels` classes are responsible for modeling and computing the viscosity, following the OOD pattern Single Responsibility Principle (SRP). A list of viscosity models and their description is available in section 11.1. Similar to the structure of `transportModel`, the `viscosityModel` class is an abstract base class for the actual viscosity models.

As outlined in figure 11.2, the `viscosityModel` implements a public member function `nu`, which finally returns the kinematic viscosity. This member function provides access to viscosity in any of the `transportModels`. In the base class `viscosityModel`, this is a virtual member function and needs to be implemented by each derived class:

```
virtual tmp<volScalarField> nu() const = 0;
```

With `singlePhaseTransportModel` and `incompressibleTwoPhaseMixture` hard coded into the particular solver, the `viscosityModels` are the final types which are selected by the user. Hence, they have to be run-time selectable using specific entries in the `constant/transportProperties` dictionary. Because of this, the base class must implement the Open-FOAM RTS mechanism.

As an example for a `viscosityModel` the `Newtonian` class is selected. It describes the viscosity for an incompressible Newtonian fluid and solely inherits from `viscosityModel`. Additional data is stored in the following private member variables:

```
dimensionedScalar nu0_;

volScalarField nu_;
```

The `nu` member function must be implemented by this class, as it's base class definition is virtual. This implementation is short and returns `nu_`. The constructor is shown in listing 68.

Of course the base class' constructor is called at the first position of the initialization list, followed by initializations for `nu0_` and `nu`.

Listing 68 Constructor of the `Newtonian` class

```
Foam::viscosityModels::Newtonian::Newtonian
(
    const word& name,
    const dictionary& viscosityProperties,
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    viscosityModel(name, viscosityProperties, U, phi),
    nu0_("nu", dimViscosity, viscosityProperties_),
    nu_
    (
        IOobject
        (
            name,
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        U_.mesh(),
        nu0_
    )
{}
```

INFO

Even though the `transportModel` is the abstract base class for all transport models, specialistic solvers will select other model classes in the hierarchy. An example is the `incompressibleTwoPhaseMixture`, which *is a* `transportModel`, but is also a `twoPhaseMixture`, specifically meant to be used by the two-phase solvers.

11.3 Implementation of a new Viscosity Model

The implementation of a custom `viscosityModel` is built on an example where the viscosity depends on the shear rate. Blood is taken as an example fluid, and the experimental data that relate the viscosity to the shear rate are given in [1, figure 2]. The experimental data is digitized from [1] into the CSV file using `engauge` digitizer, with the first column being the strain rate and the second the corresponding effective dynamic viscosity μ.

The kinematic viscosity is calculated by dividing μ with $\rho = 1060$, the approximate density of blood at room temperature, and is shown in fig-

---

INFO

The example focuses on implementing a new viscosity model in Open-FOAM, not on physical correctnes of the model.

Listing 69 Pure virtual functions of `viscosityModel`.

```
//- Return the laminar viscosity
virtual tmp<volScalarField> nu() const = 0;

//- Return the laminar viscosity for patch
virtual tmp<scalarField> nu(const label patchi) const = 0;

//- Correct the laminar viscosity
virtual void correct() = 0;

//- Read transportProperties dictionary
virtual bool read(const dictionary& viscosityProperties) = 0;
```

---

ure 11.4. This example covers implementing a new viscosity model class that reads the kinematic viscosity and the local strain-rate and returns the effective viscosity. This data table-based approach contrasts with the other more common viscosity models that use analytical expressions of the strain-rate vs. viscosity relationship. An example of a set of measurements from a shear rheometer (effective viscosity vs. strain rate) is shown in figure 11.4. Because there are only discrete data points in the rheometry table, an interpolation scheme must be used to interpolate between them and assign a viscosity to any given strain rate. OpenFOAM provide two-dimensional spline-based interpolation which can be used for this purpose: `interpolateSplineXY`. The source code for this library is available in the example repository, in

> `ofbook/ofprimer/src/ofBookTransportModels`

Here, only the implementation of virtual member functions responsible for the viscosity calculation is covered. More information about the implementation is available in the source code.

The new viscosity model inherits, as expected, from `viscosityModel`, so the interface of `viscosityModel` shown in listing 69 must be satisfied by the new viscosity model.

The `correct` member function performs the calculation (update) of the kinematic viscosity, while the `nu` member functions provide access.
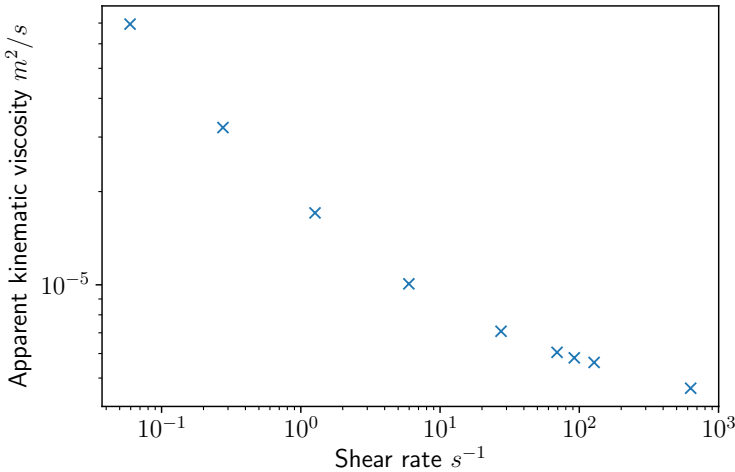
Figure 11.4: Experimental effective viscosity profile for a non-Newtonian fluid from [1, figure 2].

INFO

In OpenFOAM, as in every other large-scale object-oriented software, the alternative implementation of *virtual functions* extend the functionality of a library. In other words, when extending a library in OpenFOAM, find out what virtual functions do in other derived classes, as those member functions are most likely to be modified in your new class as well.

The viscosity model extension starts with the public inheritance from `viscosityModel` and the definition of private data members, required for the viscosity calculation, shown in listing 70. The private member function `loadViscosityTable` in listing 70 is responsible for reading the experimental data from the CSV file, which is converted to scalar fields `rheologyTableX` and `rheologyTableY`, used as input for the spline-based interpolation in OpenFOAM.

The declaration of the pure virtual member functions inherited from `viscosityModel` is shown in listing 71. The kinematic viscosity field is a private data member (wrapped in a `tmp` smart pointer), so the viscosity field and the patch viscosity field can be returned easily. The main part of the model implementation therefore lies in `loadViscosityTable` and

---

Listing 70 Inheriting from `viscosityModel`.

```
class interpolatedSplineViscosityModel
:
    public viscosityModel
{
    // Private data

        //Dictionary for the viscosity model
        dictionary modelDict_;

        //x and y entries of the rheology data for
        //   use with the spline interpolator
        fileName dataFileName_;
        scalarField rheologyTableX_;
        scalarField rheologyTableY_;

        tmp<volScalarField> nuPtr_;

    // Private Member Functions

        // Load the viscosity data table
        void loadViscosityTable();
```

Listing 71 Virtual interface of the new viscosity model.

```
//- Return the laminar viscosity
tmp<volScalarField> nu() const
{
    return nuPtr_;
}

//- Return the laminar viscosity for patch
virtual tmp<scalarField> nu(const label patchi) const
{
    return nuPtr_->boundaryField()[patchi];
}

//- Correct the laminar viscosity
virtual void correct();

//- Read transportProperties dictionary
bool read(const dictionary& viscosityProperties);
```

Listing 72 The `loadViscosityTable` member function.

```
void interpolatedSplineViscosityModel::loadViscosityTable()
{
csvTableReader<scalar> reader(modelDict_);

List<Tuple2<scalar, scalar>> data;
reader(dataFileName_, data);

// Resize to experimental data
rheologyTableX_.resize(data.size());
rheologyTableY_.resize(data.size());

forAll(data, lineI)
{
   const auto& dataTuple = data[lineI];
   rheologyTableX_[lineI] = dataTuple.first();
   rheologyTableY_[lineI] = dataTuple.second();
}
```

`correct`.

The `csvTableReader` class is used to read the CSV file that contains the experimental data from figure 11.4. The `loadViscosityTable` is called once in the body of the constructor and once in the `read` member function. The `correct` member function obtains the strain rate from `viscosityModel` in each cell, then interpolates the apparent kinematic viscosity from the experimental data read by `loadViscosityTable` using a spline-based interpolation. The implementation is outlined in listing 73.

The definition of the new viscosity model is in the `constant/transportProperties` dictionary file:

```
transportModel  interpolatedSplineViscosityModel;
interpolatedSplineViscosityModelCoeffs
{
   dataFileName "constant/viscosityData/anand2004kinematic.csv";
   hasHeaderLine false;
   refColumn 0;
   componentColumns (1);
}
rho         1060;
```

The input of the new viscosity model contains the path to the file with the experimental data, as well as information that defines the structure of the CSV file. The CSV file does not contain a header, the reference

Listing 73 The `correct` member function interpolates the kinematic viscosity from the strain rate using experimental data from [1].

```
void interpolatedSplineViscosityModel::correct()
{
    // Interpolate kinematic viscosity in each cell from tabular data.
    volScalarField& nu = nuPtr_.ref();
    const volScalarField cellStrainRate = strainRate();
    forAll(cellStrainRate, cellI)
    {
        nu[cellI] = interpolateSplineXY
        (
            cellStrainRate[cellI],
            rheologyTableX_,
            rheologyTableY_
        );
    }
    nu.correctBoundaryConditions();
}
```

column is 0, and there is only a single component column 1, with the experimental apparent kinematic viscosity from [1].

11.3.1 Example Case

A falling "blood" droplet impacting an impenetrable wall is the example case for the spline-based viscosity model. The case itself is included in the repository, at

```
ofbook/ofprimer/cases/chapter11/falling-droplet-2D
```

A two-phase VoF solver `interIsoFoam` [2] is used to simulate a non-Newtonian liquid droplet falling through air and impacting a solid surface. Note that the simulation is two-dimensional and spatially under-resolved; it still serves well as an example simulation. The square domain has three impenetrable walls and one open atmosphere boundary condition as shown in figure 11.5. The case can be run to completion using the prepared `Allrun` script that requires 4 CPU cores.

The droplet of diameter 1 mm starts with zero velocity, centered in the domain. Gravity accelerates the droplet downward, where it eventually impacts the no-slip wall. As the droplet impacts the wall, high local strain rates alter the effective viscosity in the liquid. The non-Newtonian effects are visualized in figure 11.6.
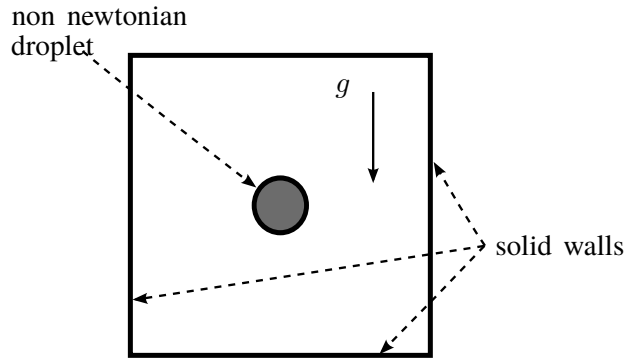
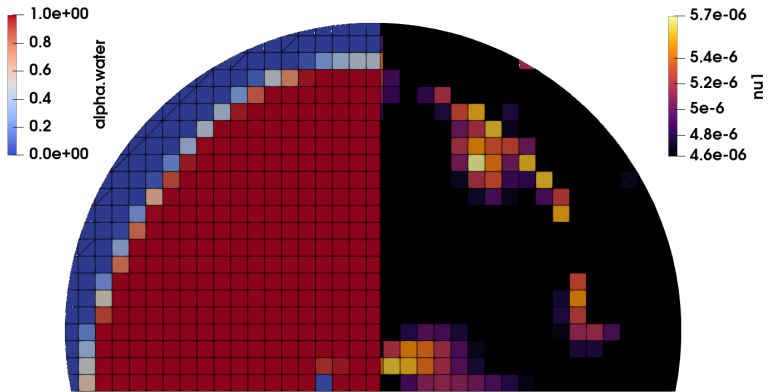Figure 11.5: Solution domain of the example case.

Figure 11.6: Comparison of the volume fraction and the local shear-dependent viscosity of the droplet impacting a wall.

Further reading

[1] M Anand and Kr R Rajagopal. "A shear-thinning viscoelastic fluid model for describing the flow of blood". In: *Int. J. Cardiovasc. Med. Sci.* 4.2 (2004), pp. 59–68. URL: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/taos-10/publications/MAKRR2004.pdf.

[2] Henning Scheufler and Johan Roenby. "Accurate and efficient surface reconstruction from volume fraction data on general meshes". In: *J. Comput. Phys.* 383 (Apr. 2019), pp. 1–23. ISSN: 10902716. DOI: 10.1016/j.jcp.2019.01.009. arXiv: 1801.05382. URL: https://www.sciencedirect.com/science/article/pii/S0021999119300269.

[3] Hermann Schlichting and Klaus Gersten. *Boundary-Layer Theory*. 8rd rev. ed. Berlin: Springer, 2001.

[4] D. C. Wilcox. *Basic Fluid Mechanics*. 3rd rev. ed. DCW Industries Inc., 2007.

12

Function Objects

A function object is solver-independent code compiled into a dynamic library and executed either during the simulation time loop or by a post-processing application after the simulation has been completed. Function objects most often perform post-processing calculations as the simulation runs, but they can also manipulate the fields or the simulation parameters. An OpenFOAM function object library can contain an arbitrary number of function objects that can be linked to the application during runtime: OpenFOAM provides a mechanism for loading dynamic libraries in every simulation case. The function objects' encapsulated functionality is accessed from the solver in a predefined way, so the function objects must adhere to a fixed class interface.

Like transport models or boundary conditions, OpenFOAM function objects are implemented as classes, organized into class hierarchies, and compiled into dynamically linked libraries. This allows the user to select any number of different function objects via configuration files. Another benefit of the modular hierarchical structure of function objects in Open-FOAM is the ability to share function object libraries easily with others. Function objects implement fully self-sustained computations and do not involve any modification of the application code at all. No changes must be applied to the solvers or utility applications, and there is no need to recompile them to access new function objects. This improves the re-usability of the code following the OOD pattern of "Open-Closed": functionality is open for extension, and the existing implementation is closed for modification.

From a user's perspective, function objects typically provide functionalities that do not affect the solution: they perform runtime post-processing tasks that should generally be independent of the selected solver. Their purpose is to implement general post-processing methods, such as calculating average values of any field variable, separately from the solver code.

As a hypothetical example, consider a parametric CFD optimization study with a pressure drop between the inlet and outlet boundary of a steady-state simulation as the target function. Using function objects, it is possible to evaluate this pressure drop after each iteration and terminate the simulation as soon as the pressure drop satisfies some desired conditions. Contrary to function objects, using post-processing applications would require each simulation to be finalized, possibly involving many more iterations than necessary for a convergent macroscopic pressure drop, depending on the solution algorithm's convergence criteria.

12.1 Software design

This section covers the software design of the C++ and OpenFOAM function objects. There are differences between C++ and OpenFOAM function objects that should be made clear because they may lead to confusion.

12.1.1 Function Objects in C++

Describing in detail the implementation and usage of function objects in C++ is beyond the scope of this book and described extensively in [2], [3] and [1], among others. A brief overview of C++ function objects is covered, sufficient to distinguish them from OpenFOAM function objects.

As their name indicates, function objects are objects that can behave like functions. In the C++ programming language, an object can behave like a function when the operator `operator()` is overloaded for its class. A very simplified example a C++ function object is shown in listing 74. Overloading arithmetic operators (`+`,`-`,`*`,`/`) allows the class to implement arithmetic operations on objects, though the implementation is not limited

---

Listing 74 Simplified form of a C++ function object implementation

---

```cpp
class CallableClass
{
    public:

        void operator()() {}
};

int main(int argc, const char *argv[])
{
    CallableClass c;

    c();

    return 0;
}
```

---

to arithmetic operations. This C++ language feature is used extensively for algebraic field operations in OpenFOAM itself. In a similar way, overloading `operator()()` for a class enables the function-like behaviour of its objects.

The fact that function objects are objects results in different advantages:

- The function object can store additional information about its state.
- It is implemented as a type which is a fact often used in generic programming.
- Its execution will be faster than the code which involves passing a function pointer because function objects are generally inlined.

A function object can accumulate information while processing arguments in the `operator()()` and store the accumulated information as data members for later use.

The following example of C++ function objects shows how to implement an OpenFOAM class that selects mesh cells with the help of the C++ STL function objects, instead of relying on OpenFOAM data structures. This class is named `fieldCellSet` and is available in the example code repository, in the `ofBookFunctionObjects` folder.

Cell selection in OpenFOAM often topological, such as selecting labels of cells whose centers lie within a sphere. The `fieldCellSet` class implemented in this example uses field values of a `volScalarField` to collect cells from the mesh. If the field values satisfy a specific criterion, the cell selector will add the particular cell label to a set of labels.

The class `fieldCellSet` is decoupled from the cell selection criterion, following the SRP, by defining the cell selection function as a function template parametrized on the selection criterion. In the following two code snippets, this template parameter is of generic type `Collector` and the parameter `col` stores an object of this type. Hence `fieldCellSet` uses template arguments to describe the actual selection criterion in `collectCells`.

```
//- Edit
template<typename Collector>
void collectCells(const volScalarField& field, Collector col);
```

The ability of `fieldCellSet` to work with any `Collector` function object that has the `operator()()` implemented, takes a scalar value and returns a boolean is a result of this. The concept of the `Collector` template parameter can be examined in the implementation of the `collectCells` member function template, defined in file `fieldCellSet-Templates.C`:

```
template<typename Collector>
void fieldCellSet::collectCells
(
    const volScalarField& field,
    Collector col
)
{
    forAll(field, I)
    {
        if (col(field[I]))
        {
            insert(I);
        }
    }
}
```

As the above snippet shows, the `collectCells` member function simply iterates over all field values and expects `Collector` to return a `boolean` variable as a result of operating on the field value `field[I]`. This specifies the collector template parameter concept:

callable: a function object is the natural choice
unary: it must allow at least one function argument
predicate: it returns a boolean variable

To summarize, the small cell collector class `fieldCellSet` has a member function template that takes a `volScalarField` and a function object `col` as arguments. It uses the function object to check if the cell should

be added to the cell set based on the value of the field. However, it is irrelevant how this comparison is implemented, as long as the `operator()` of `col` is implemented and returns a boolean.

This straightforward implementation allows *any* function object to be passed to the `collectCells` member function template. In the following example, a function object of the STL is used. The implementation of the test application for a single `fieldCellSet` class is available in the example code repository, under the `applications/test` sub-directory and it is named `testFieldCellSet`. The part of the `testFieldCellSet` that relates to function objects in C++ is a single line of code:

```
fcs.collectCells(field, std::bind1st(std::equal_to<scalar>(), 1));
```

The `collectCells` method is called on the `fcs` object of the `fieldCellSet` class.

The function object used in this example is the `equal_to` STL function object template. This generic function object simply checks whether two values of type `T` are identical. Since it uses *type lifting* aspect of generic programming, it may only be applied on instances of types that have the equality `operator==` defined. However, the `equal_to` function object requires two arguments for the comparison. The question that remains is how it can be called for the field value within the `collectCells` member function template. For this simple example, the answer is fairly simple: The first argument of the `equal_to` function object has been bound to the value of 1. Meaning that the `testFielCellSet` application should create a cell set consisted of all mesh cells with the field values equal to 1.

Object oriented design allows the `fieldCellSet` class to delegate the storage of the cell labels as well as delegate the output operation which is performed at each new time step. This was achieved using multiple inheritance:

```
class fieldCellSet
:
    public labelHashSet,
    public regIOobject
{
```

In the above snippet, the `labelHashSet` is an instantiation of the OpenFOAM class template `HashSet` using `Foam::label` as key values.

Inheriting from `regIOobject` allows the class to be registered in an object registry and written to disk automatically as time increment operator (`Time::operator++()`) is envoked. The writing is performed in such a way that the output file contains the necessary OpenFOAM file header information required by OpenFOAM to reread the data later on.

The `testFieldCellSet` application can be tested in the example case from chapter 11, namely

```
/ofprimer/cases/chapter11/falling-droplet-2D
```

by calling

```
?> blockMesh
?> setAlphaField
?> testFieldCellSet -field alpha.water
```

The resulting cell set is stored in the 0 directory and to visualize it using the paraView application, it must be converted to the VTK format using `foamToVTK`. Before this can be done, the cell set must be copied from the 0 directory to `constant/polyMesh/sets`:

```
?> mkdir -p constant/polyMesh/sets
?> cp 0/fieldCellSet !$
```

Once this is done, the cell set is converted with

```
?> foamToVTK -cellSet fieldCellSet
```

This generates the VTK directory within the `falling-droplet-2D` case. Figure 12.1 shows the `alpha1` field of the falling droplet cell set for the initial time step. The computation and visualization of the field based cell set could have easily be done in the paraView application, but the point is to have the understanding of function objects in C++, and how they may be easily used when programming OpenFOAM code. Function objects in C++ are very much like extended functions. The function objects in OpenFOAM have similar functionality to C++ function objects, but they differ in their design.
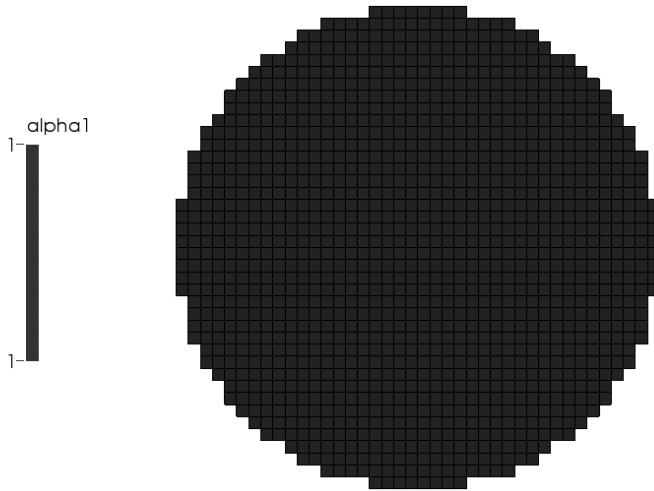
Figure 12.1: Field `alpha1` of the field based cell set comprising the rising bubble in the initial time step.

12.1.2 Function Objects in OpenFOAM

Function objects in OpenFOAM have a different class interface than standard C++ function objects. Instead on relying on overloading the `operator()`, they adhere to a class interface defined by the `functionObject` abstract class, shown in listing 75. The class hierarchy of OpenFOAM function objects is therefore structurally similar to the hierarchy used for boundary conditions (chapter 10) and transport models (chapter 11). An abstract base class (`functionObject`) prescribes the interface of all function objects in OpenFOAM. Compared to the standard C++ function objects that overload the call operator, the OpenFOAM function objects use various virtual functions that make it possible for them to be called at different steps of the simulation loop.

The execution of the `functionObject` member functions is related to either changes of the simulation time or changes of the mesh, as shown in listing 75. Simulations in OpenFOAM are controlled by the `Time` class and the majority of the member functions of `functionObject` have something to do with the change in the simulation time. Thus the `Time` class is made responsible for invocation of `functionObject` member functions, depending on the event. Two member functions that relate to mesh motion (`movePoints`) and mapping of fields (`updateMesh`) are

---

Listing 75 Class interface of the `function object` abstract base class

```
// Member Functions

    //- Name
    virtual const word& name() const;

    //- Called at the start of the time-loop
    virtual bool start() = 0;

    //- Called at each ++ or += of the time-loop.
    //  forceWrite overrides the outputControl behaviour.
    virtual bool execute(const bool forceWrite) = 0;

    //- Called when Time::run() determines that
    //  the time-loop exits.
    //  By default it simply calls execute().
    virtual bool end();

    //- Called when time was set at the end of
    //  the Time::operator++
    virtual bool timeSet();

    //- Read and set the function object if
    //  its data have changed.
    virtual bool read(const dictionary&) = 0;

    //- Update for changes of mesh
    virtual void updateMesh(const mapPolyMesh& mpm) = 0;

    //- Update for changes of mesh
    virtual void movePoints(const polyMesh& mesh) = 0;
```

---

called from within the mesh class `polyMesh`, using the constant access to simulation time.

As the result of those requirements, the `Time` class composits all function objects loaded for each particular simulation in a list of function objects (`functionObjectList`) as shown in the diagram in figure 12.2. The `functionObjectList` implements the interface of `functionObject` and delegates the member function invocation to the composited function objects. As the simulation starts and the `runTime` object of the `Time` class gets initialized, the simulation control dictionary `controlDict` is read. The constructor of `functionObjectList` reads the `controlDict` and parses the entries in the `functions` sub-dictionary. Each entry in the `functions` sub-dictionary defines parameters of a single function object, which are then passed to the function object selector. The selector (`functionObject::New`) implements the "Factory Pattern" known from OOP and uses the dictionary parameter `type` to initialize a concrete model of the `functionObject` abstract class during runtime. Finally, the selected function object gets appended to the list of function objects. This mechanism also relies on the RTS mechanism in OpenFOAM, allowing the user to select and instantiate different function objects for different simulation cases.

Since the function objects are initialized during runtime and rely on dynamic polymorphism (virtual functions), the member functions that implement the main functionality have an overhead when resolving which virtual member function is called (dynamic dispatch). However, the calculations performed by OpenFOAM function objects take orders of magnitude more computational time than dynamic dispatch, so the overhead of using dynamic dispatch can be neglected safely.

The function objects in OpenFOAM are also objects that perform function-like operations during the simulation, so that property justifies their name. The class declaration of `functionObject` serves as the other example to describe further differences:

```cpp
// Private Member Functions

    //- Disallow default bitwise copy construct
    functionObject(const functionObject&);

    //- Disallow default bitwise assignment
    void operator=(const functionObject&);
```
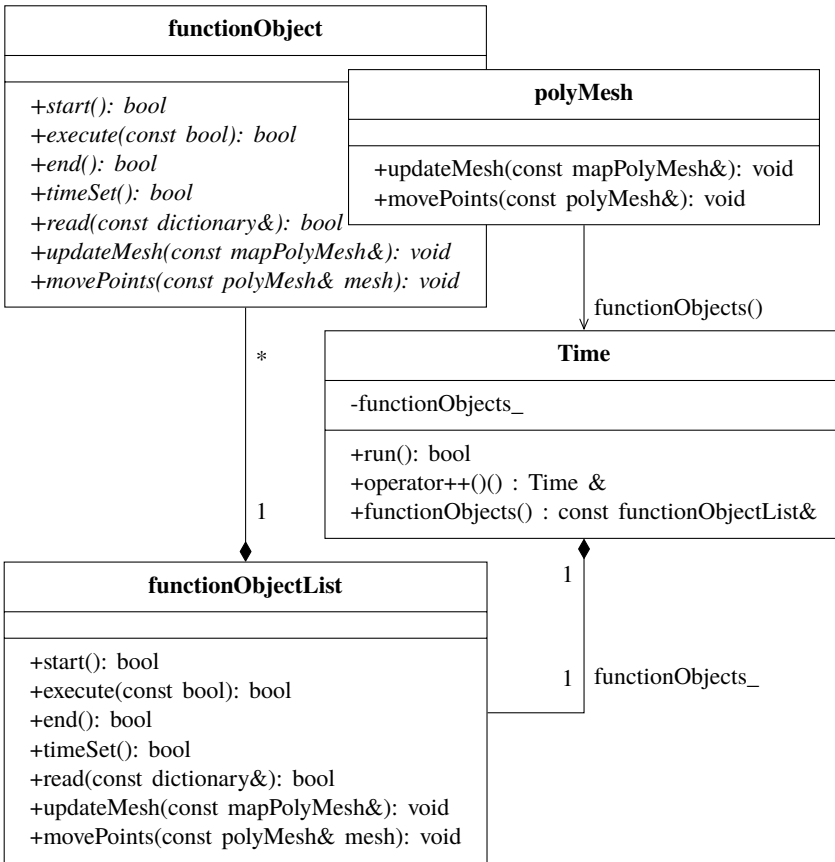
Figure 12.2: Composition of function objects in the `Time` class.

Contrary to the C++ function objects, the assignment and copy construction are both prohibited for function objects in OpenFOAM, making it impossible to pass them as value arguments. The prohibited assignment and copy-construction have no negative impact on OpenFOAM function objects: the central point of use of function objects in OpenFOAM is the private attribute `functionObjectList` in the `Time` class, so there is no need to instantiate them manually in a list or pass them around by value.

With the design of OpenFOAM function objects and their interaction with the `Time` class covered, the remaining point to be covered is the programming of new function objects. Basically any class that inherits from `functionObject`, implements its interface and whose `type` en-

try as well as necessary parameters are listed within the `controlDict` simulation control directory will be interpreted as a function object by OpenFOAM. The process of initializing function objects in a simulation is performed by the `Time` class and happens automatically, provided a function object configuration sub-dictionary is added, and the dynamically linked library that implements the function object is loaded in the `system/controlDict` file. However, before developing a new function object, it is advisable to check if the desired functionality is already available in OpenFOAM or a community contribution.

12.2 Using OpenFOAM Function Objects

Before starting with the development of a new function object, it is recommended to check if a similar function object already exists in Open-FOAM, or an OpenFOAM-related project. The swak4Foam project, developed by Bernhard Gschaider, contains different useful function objects.

Using function objects from OpenFOAM and swak4foam is done in this chapter using the falling droplet example from the previous chapter

```
ofprimer/cases/chapter11/falling-droplet-2D
```

12.2.1 OpenFOAM Function Objects

The organization of function objects in OpenFOAM is described in detail in the Extended Code Guide, so this information is omitted here.

The starting point for examining the source code of function objects in OpenFOAM is `$FOAM_SRC/postProcessing/functionObjects`.

Function objects from OpenFOAM are enabled by approxpriate entries in the `system/controlDict` file in the simulation folder. Parameters required by the function object and the library which implements the function object must be specified in the `controlDict`.

To usage of OpenFOAM function objects can be demonstrated using the `CourantNo` function object, that computes the Courant number for every cell in the mesh and stores it as a `volScalarField` and writes it so that it can be visually inspected. To use the `courantNumber` function

---

Listing 76 Definition of the `CourantNo` function object

```
tions

courantNo
{
    type CourantNo;
    phiName phi;
    rhoName rho;
    writeControl outputTime;
    libs ("libfieldFunctionObjects.so");
}
...
```
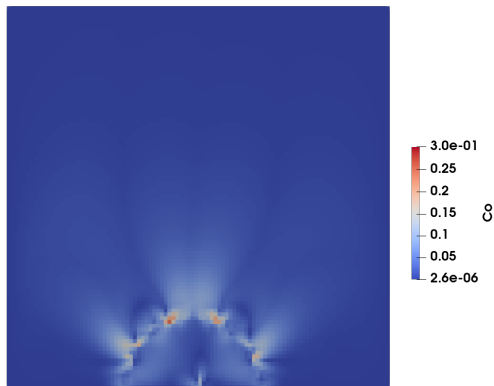

Figure 12.3: Courant number distribution for the droplet impacting a wall.

object, the the dynamically loaded library and the `functions` entries in the `system/controlDict` file of the `falling-droplet-2D` simulation case need to be defined as shown in listing 76. For the `courantNo` function object, as well as for other function objects in OpenFOAM, the output is independent from the simulation output. Because of this, the additional entry `writeControl` is necessary to prescribe the output control of the function object. Setting `writeControl` to `outputTime` writes the Courant number field using the same output frequency as the output frequency of other fields in the simulation. Figure 12.3 shows the resulting Courant number distribution for the droplet simulated with the `interIsoFoam` solver at 0.04 seconds.

12.3 Implementation of a custom Function Object

Implementing a new function object requires defining the class interface of the `functionObject` class and preparing files necessary for the compilation of the function object library. These tasks repeat themselves when different function objects are implemented. To save development time, a shell script is available in OpenFOAM that generates the necessary file structure of a new function object library. The script is named `foam-NewFunctionObject`, it generates a directory structure, function object class files (`.C` and `.H`) and the essential build configuration for a function object library that contains a single function object.

12.3.1 Function object generator

> **INFO**
>
> To use the function object generator, make sure that the OpenFOAM environment is set.

Following commands create a new function object library with the `foam-NewFunctionObject` script

```
?> foamNewFunctionObject myFuncObject
?> cd myFuncObject
```

In OpenFOAM-v2012 there is a syntax error in the template file, that has been fixed on the master branch. Since the book describes release versions, a small modification in the constructor of the generated function object should be done. The line with the bracket error

```
Foam::functionObjects::myFuncObject::myFuncObject
(
    const word& name,
    const Time& runTime,
    const dictionary& dict
)
:
    fvMeshFunctionObject(name, runTime, dict),
    // Bracket error.
    boolData_(dict.getOrDefault<bool>("boolData"), true),
```

should be modified like this:

```
Foam::functionObjects::myFuncObject::myFuncObject
(
    const word& name,
    const Time& runTime,
    const dictionary& dict
)
:
    fvMeshFunctionObject(name, runTime, dict),
    // Bracket error fixed.
    boolData_(dict.getOrDefault<bool>("boolData",true)),
```

After this small syntax correction, the new library can be compiled with

```
myFuncObject > wmake libso
```

The `foamNewFunctionObject` script generates one OpenFOAM library per function object, as defined in `myFunctObject/Make/files`

```
myFuncObject > cat Make/files
myFuncObject.C

LIB = $(FOAM_USER_LIBBIN)/libmyFuncObjectFunctionObject
```

The library is named automatically as `myFuncObjectFunctionObject` and stored in the OpenFOAM folder that contains user-defined library binaries.

It is impractical to generate a different function object library for each new function object we introduce, because this requires adding a library entry in the `system/controlDict` file for each function object generated with `foamNewFunctionObject`. Function objects that can be categorized into a group, belong to the same library. To organize function objects generated with `foamNewFunctionObject` into groups, multiple function object folder structures can be generated with `foamNewFunctionObject`, and compiled into a single library. For example,

```
?> mkdir myFunctionObjects && cd myFunctionObjects
?> foamNewFunctionObject myFuncObjectA
?> foamNewFunctionObject myFuncObjectB
```

creates two function objects, `myFuncObjectA` and `myFuncObjectB`, that should be compiled into the same library, `myFunctionObjects`. The files `Make/files,options` from any function object generated by `foamNew-FunctionObject` can be used as a start for the build configuration of the library. For example, using the build configuration from `myFuncObjectB`,

```
myFunctionObjects > mv myFuncObjectB/Make .
myFunctionObjects > rm -rf myFuncObjectA/Make
```

its `Make` folder now contains the build configuration of the `myFunc-tionObjects` library, that needs to be modified. Specifially, the `my-FunctionObject/Make/files` should look like this:

```
myFuncObjectA/myFuncObjectA.C
myFuncObjectB/myFuncObjectB.C

LIB = $(FOAM_USER_LIBBIN)/libmyFunctionObjects
```

which basically lists the two implementation (`.C`) files of the two function objects contaires in the `myFunctionObjects` library, and specifies the location (`$FOAM_LIBBIN`), and the name of the new function object library.

> **INFO**
>
> If you are on the latest commit on the master branch, the syntax error present on the OpenFOAM-v2012 tag has been fixed.

To compile the library, fix the small syntax erro in the constructor of the both function objects as discussed above, and execute `wmake libso`. Once the library is compiled successfully, function objects from the `myFunc-tionObjects` library can be used like any other OpenFOAM function object, with any solver and any simulation case, provided that the `my-FunctionObjects` library is listed as a dynamically loaded library in the `controlDict` file. For example, the new library is used in the `cavity` tutorial case with the `icoFoam` solver as follows:

```
?> mkdir -p $FOAM_RUN && cd !$
?> mkdir myFunctionObjectsTest && cd myFunctionObjectsTest
?>  cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity .
?> cd cavity
```

The new function object `myFuncObjectA` from the new library `myFunc-tionObjects` is activated by the following entries in `system/control-Dict` file:

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

functions
{
    funcA
    {
        type myFuncObjectA;
    }
}
```

The first line specifies the new library that should be dynamically loaded. OpenFOAM automatically searches for $FOAM_LIBBIN and $FOAM_USER_ LIBBIN for available libraries. The functions entry is a dictionary may contain arbitrary many dictionary entries, each specifying a different function object. In the above snippet, funcA sub-dictionary configures the myFuncObjectA function object. The name funcA is arbitrary: any user-defined name is acceptable.

Generating the mesh with blockMesh and starting the icoFoam solver, leads to the error

```
--> FOAM FATAL IO ERROR: (openfoam-2012)
Entry 'labelData' not found in dictionary
"path/to/cavity/system/controlDict.functions.funcA"
```

This is expected: function object generator script foamNewFunctionObject prepares a skeleton implementation that does nothing except from initialization of some dummy data members (bool, label and scalar). If the data for those data members are not provided in system/controlDict, the function object will complain.

If, on the other hand, we misname the function object, OpenFOAM delivers a warning:

```
--> FOAM Warning :
Unknown function type myFuncObjectAA
Valid function types :
2(myFuncObjectA myFuncObjectB)
```

Therefore, there are two ways in which using function objects may go wrong: we misname the function object, or we do not provide the data required for its initialization. If we misname a function object, the solver will run without it. If a data entry required to initialize the function object is missing, the solver does not run and the user is promted to enter required data in system/controlDict. The complete definition of the funcA sub-dict in system/controlDict is

```
functions
{
    funcA
    {
        type myFuncObjectA;
        boolData false;
        labelData 0;
        scalarData 0;
    }
}
```

Once the template function object implementation is compiled into a library and can be used (tested) with a solver, the default implementation can be extended.

The dummy data members generated by `foamNewFunctionObject`:

```cpp
class myFuncObjectA
:
    public fvMeshFunctionObject
{
    // Private Data

        //- bool
        bool boolData_;

        //- label
        label labelData_;

        //- word
        word wordData_;

        //- scalar
        scalar scalarData_;
```

will be replaced by data members that support the computation of the new function object. Member functions generated by `foamNewFunctionObject` do practically nothing:

```cpp
bool Foam::functionObjects::myFuncObjectA::read(const dictionary& dict)
{
    dict.readEntry("boolData", boolData_);
    dict.readEntry("labelData", labelData_);
    dict.readIfPresent("wordData", wordData_);
    dict.readEntry("scalarData", scalarData_);

    return true;
}

bool Foam::functionObjects::myFuncObjectA::execute()
{
    return true;
}

bool Foam::functionObjects::myFuncObjectA::end()
{
    return true;
}

bool Foam::functionObjects::myFuncObjectA::write()
{
    return true;
}
```

and should be implemented.

12.3.2 Implementing the Function Object

When the function object implements a generally useful calculation, it makes sense to encapsulate the function objects' computation in a separate class and re-use it inside the function object. Sometimes this decoupling of the calculation from the runtime processing capability leads to a more clean implementation with separated concerns: the class that calculates something is separate from the function object class that deals with the runtime execution. When a new function object is programmed, one should consider re-using existing classes in OpenFOAM within function objects instead of programming everything from scratch, as this may shorten development time. In this section's example, the function object class implements the calculation and inherits the runtime operation from `functionObject`.

As described above, the function object's member functions are called at specific places within the simulation loop in OpenFOAM:

- `start` : executed at the beginning of the time loop,
- `execute` : executed when the time is increased,
- `end` : executed when the time-loop ends (time has reached the value of `endTime`).

In case when the calculation is not different at the beginning and the end of the time loop, member functions `start` and `end` call the member function `execute`. The function object described in this section keeps track of all the mesh cells that have contained a specific phase during a multiphase simulation. This example is not especially useful, its only purpose is to demonstrate the calculations usually performed by function objects, involving the mesh, the fields, and the simulation time control.

The code for the function object example is available in the example code repository in

```
src/ofPrimerFunctionObjects/phaseCellsFunctionObject
```

Viewing the code in a text editor may help with understanding the below described example.

To start programming the library, create a directory named `ofPrimer-FunctionObjects` for the new function object library and generate the function object from the template using `foamNewFunctionObject`:

```
?> mkdir ofPrimerFunctionObjects && cd ofPrimerFunctionObjects
?> foamNewFunctionObject phaseCellsFunctionObject
?> mv phaseCellsFunctionObject/Make .
```

Edit `Make/files` so that the function object is compiled into the new library

```
phaseCellsFunctionObject/phaseCellsFunctionObject.C
LIB = $(FOAM_USER_LIBBIN)/libofPrimerFunctionObjects
```

If the git tag OpenFOAM-v2012 is used, correct the bracket error in constructor line

```
// Fixed bracket error.
boolData_(dict.getOrDefault<bool>("boolData", true)),
```

compile the library

```
ofPrimerFunctionObjects > wmake libso
```

then test it on the falling droplet test case

```
cases/chapter11/falling-droplet-2D
```

by adding the following entry to `system/controlDict`

```
markPhaseCells
{
    type phaseCellsFunctionObject;
    libs ("libofPrimerFunctionObjects.so");
    boolData false;
    labelData 0;
    scalarData 0;
}
```

The `Allrun` script generates the mesh and starts the simulation, and the function object generated from the template does nothing at first.

> **INFO**
>
> The implementation of the function object is described below and it assumes some prior knowledge of the C++ programming language: classes / encapsulation, declaration vs definition, virtual functions, inheritance, etc.

The name of the function object used for the Runtime Type Selection (RTS) in OpenFOAM is generated by `foamNewFunctionObject` as "name" + "FunctionObject", where "name" is the argument given to `foamNewFunctionObject`. The type name of the generated function object can be changed in `phaseCellsFunctionObject.H`, from

---

Listing 77 Private attributes of `phaseCellsFunctionObject`

```cpp
// Private data

//- Time.
const Time& time_;

//- Mesh const reference.
const fvMesh& mesh_;

//- Name of the phase indicator field.
word fieldName_;

//- Reference to the phase indicator.
const volScalarField& alpha1_;

//- Phase cells field.
volScalarField phaseCells_;

//- Phase cell tolerance.
scalar phaseTolerance_;

//- Percent of cells that have contained the phase.
scalar phaseDomainPercent_;
```

```cpp
//- Runtime type information
TypeName("phaseCellsFunctionObject");
```

to

```cpp
//- Runtime type information
TypeName("phaseCells");
```

This change requires re-compilation of the library and the modification of the `type` attribute in the function object's sub-dictionary in `system/-controlDict`.

The computation of cells that have contained a specific phase during a simulation requires data: the name of the phase indicator field (e.g., volume fraction), the tolerance used to determine if the cell contains a specific phase, and the indicator field for marking cells that have contained the phase, implemented as private attributes of the `phaseCells-FunctionObject`, as shown in listing 77. Once the private attributes are declared, they need to be initialized by the class constructor, as shown in listing 78. Of course, this definition requires a respective declaration in `phaseCellsFunctionObject.H`. The runtime type selection (RTS) of the `phaseCellsFunctionObject` uses the above mentioned type name

Listing 78 Initialization of data members of `phaseCellsFunctionObject`

```cpp
phaseCellsFunctionObject::phaseCellsFunctionObject
(
    const word& name,
    const Time& time,
    const dictionary& dict
)
:
    functionObject(name),
    time_(time),
    mesh_(time.lookupObject<fvMesh>(polyMesh::defaultRegion)),
    fieldName_(dict.lookup("phaseIndicator")),
    alpha1_
    (
        mesh_.lookupObject<volScalarField>
        (
            fieldName_
        )
    ),
    phaseCells_
    (
        IOobject
        (
            "phaseCells",
            time.timeName(),
            time,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar
        (
            "zero",
            dimless,
            0.0
        )
    ),
    phaseTolerance_(dict.get<scalar>("phaseTolerance")),
    phaseDomainPercent_(0)
{}
```

```
//- Runtime type information
TypeName("phaseCells");
```

The `foamNewFunctionObject` applies the OpenFOAM macros necessary for runtime type selection (RTS): the only thing that can be modified is the above-mentioned type name of the function object. The function object is selected by providing the `type` entry in the respected sub-dict `system/controlDict` dictionary.

Afte the attribute initialization, the actual calculation is implemented in the `phaseCellsFunctionObject::execute`

```
bool phaseCellsFunctionObject::execute()
{
    calcWettedCells();
    calcWettedDomainPercent();
    report();
    return true;
}
```

Here the sub-calculations of the `execute()` member function have been separated. This is not necessary in this small example. However, it is good practice in software engineering when the function calculation is large, and it becomes difficult to understand what the function actually does. Organizing sub-calculations into sub-functions makes the implementation modular: variations of sub-calculations can be implemented, and the class can be extended with inheritance without modifying the existing implementation. Additionally, organizing larger algorithms into sub-algorithms and implementing them as sub-functions improves the readability of the main algorithm, as seen above for the `execute()` member function.

The `calcWettedCells` member function marks the cells that are "wetted",

```
void phaseCellsFunctionObject::calcWettedCells()
{
    forAll (alpha1_, cellI)
    {
        if (isWetted(alpha1_[cellI]))
        {
            phaseCells_[cellI] = 1;
        }
    }
}
```

under the condition

```
bool isWetted(scalar s)
{
    return (s > phaseTolerance_);
}
```

The percentage of cells that were "wetted" by the specific phase is computed as

```
void phaseCellsFunctionObject::calcWettedDomainPercent()
{
    scalar phaseCellsSum = 0;

    forAll (phaseCells_, cellI)
    {
        if (phaseCells_[cellI] == 1)
        {
            phaseCellsSum += 1;
        }
    }

    phaseDomainPercent_ =
        100. * phaseCellsSum / alpha1_.size();
}
```

The report is made to the standard output streem with

```
void phaseCellsFunctionObject::report()
{
    Info << "Phase " << fieldName_ << " covers " << phaseDomainPercent_
        << " % of the solution domain." << endl << endl;
}
```

Since the function object contains data members that can be written down during the simulation for later inspection, the `write()` member function is implemented as

```
bool phaseCellsFunctionObject::write()
{
    if (time_.writeTime())
    {
        phaseCells_.write();
    }
    return true;
}
```

This ensures that the `phaseCells` field is only written to the disk at the write frequency defined by the user of the simulation and controlled by `Foam::Time`.

The following sub-dictionary of the `functions` dictionary within `system/controlDict` enables the function object:

```
markPhaseCells
{
    type phaseCells;
    libs ("libofPrimerFunctionObjects.so");
    phaseIndicator alpha.water;
    phaseTolerance 1e-06;
}
```

This defines the dictionary entries of the constructor (see listing 78), and the dynamically linked library of the function object. Executing `Allrun` in `cases/chapter11/falling-droplet-2D` results in the `phaseCells` being written to disk and the following addition to the solver output

```
Phase alpha.water covers 5.4 % of the solution domain.
Phase alpha.water covers 5.4 % of the solution domain.
Phase alpha.water covers 5.4 % of the solution domain.
Phase alpha.water covers 5.48 % of the solution domain.
Phase alpha.water covers 5.48 % of the solution domain.
```

Further reading

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] Nicolai M. Josuttis. *The C++ standard library: a tutorial and reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. 1st ed. Addison-Wesley Professional, Nov. 2002.

13

Dynamic mesh handling

Generally there are two main dynamic mesh handling capabilities in OpenFOAM: *mesh motion* and *topological changes* ([4], [5]). Mesh motion solely involves displacement of the mesh points without altering mesh topology: the connectivity between mesh elements such as points, edges, faces, and cells. Displacing mesh points might seem like a rather trivial task, but depending on what motion is desired, the required operations can be more complex than expected. Geometrical information used by the unstructured FVM, such as face centers and face area normal vectors, is built from the list of unique mesh points. Therefore, the displacement of mesh points triggers the computation of the geometrical information required by the unstructured FVM. After the mesh has been deformed, the fields whose values still relate to the initial mesh need to be mapped to the new mesh. This mapping must be applied for cell-centered and face-centered fields because the field values in FVM represent values averaged over the volume of a cell or the area of a cell-face, and faces and cells may experience modifications by mesh refinement and de-refinement.

Changing the mesh topology often includes adding or removing mesh elements: points, edges, faces, or cells, which ultimately changes the connectivity between the mesh elements. Therefore, the operations involving topological mesh changes are rather complex compared to mesh motion, as they involve more complex algorithms and data structures. There are two main problem categories for which topological changes may be required to obtain a fast and accurate solution: moving mesh boundaries and large gradients in the solution. When an object moves significantly

inside the domain, and relative motion exists between mesh points, the cells are likely to get either too distorted or compressed. An example of this is a piston moving in a cylinder: layers of cells are added or removed, parts of the mesh can be disconnected from each other and be reconnected later on.

The second problem category requires the simulation to have higher accuracy in those domain regions that are unknown a priori, e.g., at a point when the mesh is generated. For simulations that involve shocks in the simulation domain where the shock position is a part of the solution process, topological changes are applied based on, e.g., pressure gradient to achieve local static refinement in the region the shock appears. A different example is an interface between two immiscible liquid phases in a two-phase simulation: physical properties change abruptly in values across the interface, but the interface position is known at the start of the simulation. However, to obtain a more accurate solution, the mesh is locally and dynamically refined near the fluid interface. This refinement follows the interface as it moves across the computational domain.

Keeping the dynamic mesh operations on a higher level of abstraction by encapsulating them into classes in a class hierarchy allows the Open-FOAM user to detach the dynamic mesh operations from a flow solver and combine them. Dynamic mesh classes can also be combined using object-oriented design principles to extend the flow solver with multiple dynamic mesh operations to increase the accuracy and flexibility of a numerical simulation.

An overview of the existing types of dynamic meshes is provided in this chapter, and a set of chosen dynamic mesh engines available in OpenFOAM are described in more detail. Extensive details about the design of the base classes' design and some selected dynamic meshes can be found in section 13.1. Of more interest for the reader interested in using dynamic mesh handling is section 13.2, which presents some usage examples. Extending the existing dynamic mesh handling available in OpenFOAM by combining solid body motion and hexahedral mesh refinement is described in section 13.3.

13.1 Software design

The design of the dynamic mesh classes varies depending on the functionality of each particular dynamic mesh. In this chapter, a brief overview of the functionality and the design of dynamic meshes in OpenFOAM is provided, involving mesh motion and topological changes of the mesh.

13.1.1 Mesh motion

There are two main types of mesh motion operations implemented in OpenFOAM: *solid body mesh motion* and *mesh deformation*.

As the name suggests, the solid body motion displaces mesh points while retaining their relative position. Solid-body mesh motion involves translation and rotation of the mesh or a combination of both, and it can either be prescribed or computed as a solution of ordinary differential equations that model the solid-body dynamics.

Contrary to the mesh motion, the mesh deformation distributes the displacements unequally, from the mesh boundary into the inner parts of the solution domain. The displacement distribution utilizes different methods: either an algebraic interpolation or the solution of a transport (usually Laplace) equation for the point displacements or velocities.

Solid-body mesh motion

Solid-body mesh motion is sketched in figure 13.1. In OpenFOAM, the solid body motion is defined by a variety of classes, all derived from their common base: `solidBodyMotionFunction`. The motion function returns a `septernion` which describes the motion of the body. This motion is a combination of a vector for the translation and a `quaternion` for the rotation. The `septernion` displaces each mesh point with the same vector transformation. More information about quaternions can be found in the book by [2].

Classes derived from the abstract base class `solidBodyMotionFunction` define what type of motion is present. Ranging from simple ones, such as `linearMotion` and `rotatingMotion` over more sophisticated functions, like `tabulatedMotion`, to complex and rather specific motions
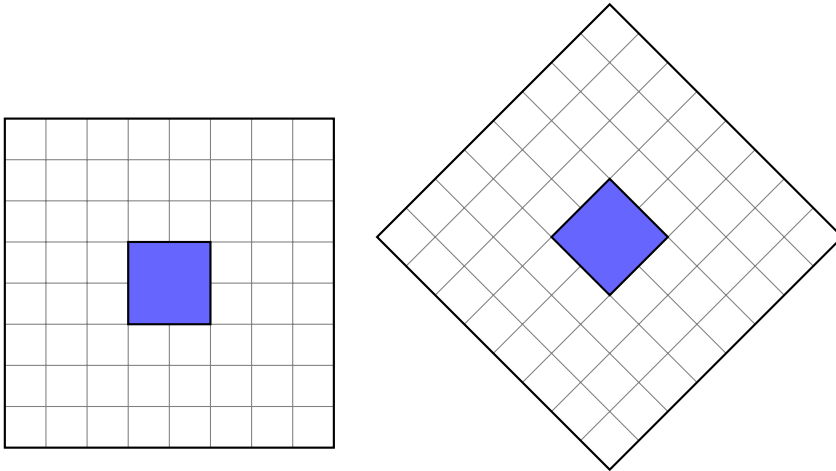
Figure 13.1: The filled body is subjected to a solid body motion, so the mesh points do not move relatively to each other.

(SDA for seakeeping investigations of ships). As can be seen from the UML diagram in figure 13.1, the `solidBodyMotionFunction` defines the virtual method `transformation` that in turn is implemented by each derived class. It calculates the septernion used to transform the mesh points between the initial position and the position at the current time and represents both the translation and the rotation.

The transformation function is encapsulated in a class, which enables runtime selection and re-use. None of the classes derived from `solid-BodyMotionFunction` change the location of the mesh points; they only provide the transformation.

The dynamic mesh class performs the actual solid body mesh motion is named `solidBodyMotionFvMesh` and inherits from `dynamicFvMesh`. The relationship between `solidBodyMotionFvMesh` and `solidBodyMo-tionFunction` is shown on the class diagram in figure 13.2. The trans-formation function is implemented using the Strategy Pattern ([1]) for the `solidBodyMotionFvMesh` class. Consequently, the solid body mo-tion function can be re-used by other library parts as a standalone entity. To compute the solid body motion, the relationship between a class and the solid body mesh motion must not be known. Additionally, the `solid-BodyMotionFunction` is designed using the Strategy pattern and allows for easy addition of other functions. Simply inheriting from `solidBody-`
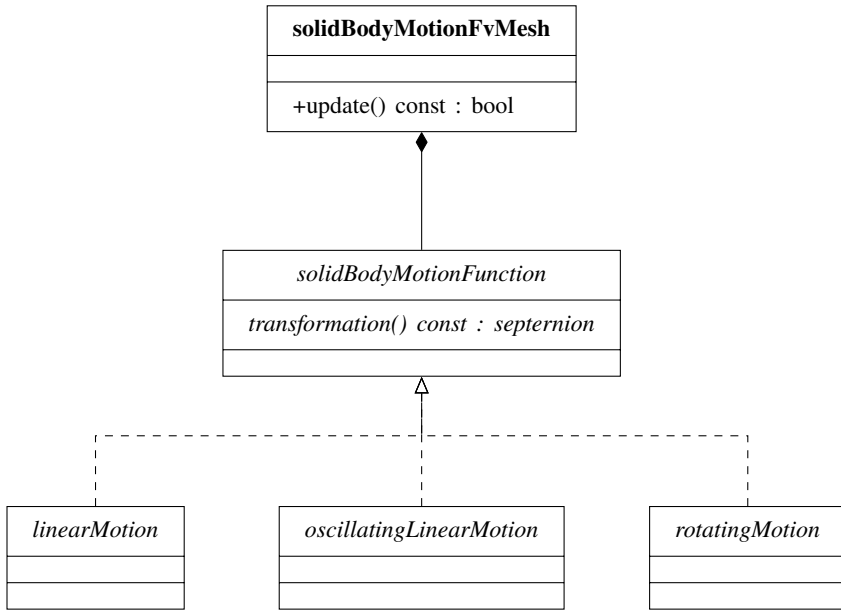
Figure 13.2: Class diagram for the `solidBodyMotionFvMesh` and the `solidBodyMotionFunction`.

> **INFO**
>
> **Strategy pattern:**
> Encapsulating various algorithms in a class hierarchy, composing them in the client (user) class, and making each of them selectable during runtime.

`MotionFunction`, the new functions are made runtime selectable.

If, on the other hand, the `solidBodyMotionFunction` would be implemented as a Template Method pattern in the `solidBodyMotionFvMesh` class, adding another mesh motion function would result in having one more dynamic mesh class. An example of the Template Method pattern application is the `dynamicFvMesh` class and the derived dynamic mesh classes, implementing the `update` algorithm.

The design of the `solidBodyMotionFunction` class hierarchy is a good example of the SRP design principle: even though the motion functions are fairly simple, they are encapsulated as individual classes having a single simple responsibility. The motion function is concerned with com-

> **INFO**
>
> **Template Method pattern:**
> A virtual function is used to implement the algorithm and the (base) class may provide a default algorithm implementation. As a result, the derived classes implement the variety of alternative algorithms, resulting in a per-algorithm branching in the hierarchy.

puting the transformation and nothing else. Additionally that, the motion functions could generally be combined: e.g., an oscillating linear motion with constant rotation. In that case, the Template Method pattern breaks down, as it relies on using multiple inheritances with the dynamic mesh class.

The `solidBodyMotion` dynamic mesh transforms either all points of the mesh, or a specific part (i.e. a `cellZone`), using the same septernion defined by the user selected `solidBodyMotionFunction`. If no `cellZone` is specified, all mesh points are moved using the same septernion; hence a uniform motion is achieved. If a `cellZone` is specified in the dictionary, only the points of the cells in the `cellZone` are moved accordingly. This comes in handy when a body rotates in a rotor-stator configuration, using sliding interfaces (AMI/GGI). In addition to the `solidBodyMotionFvMesh`, there is the `multiSolidBodyMotionFvMesh`, which basically does the same as `solidBodyMotionFvMesh`, but allows for multiple motions to be either superimposed or act upon different `cellZones`. The latter is very important for applications where two rotors rotate, using different `cellZones` and AMI/GGI interfaces. The motion itself is applied to all relevant mesh points directly, using `fvMesh::movePoints`, so no extra equations are solved, which makes this a fairly fast approach.

> **INFO**
>
> The `cellZone`-based selection of mesh points can be used for a rotor-stator mesh motion configuration that involves a sliding interface.

Mesh deformation

Mesh deformation introduces different relative motions between mesh points, which distorts edges, faces, and cells. High distortions caused by mesh deformation may destroy the quality of the mesh.

> INFO
>
> Mesh quality depends on the numerical method chosen for equation discretization. For the FVM, the two most important mesh-related discretization errors are the non-orthogonality and the skewness error ([3], [6]).

To ensure a high mesh quality, often only a small sub-region of the mesh is deformed strongly. In contrast, the deformation for the rest of the mesh is kept as low as possible, for example, in the vicinity of the mesh boundary. Additionally, the mesh deformation can be regarded as an optimization problem, where the quality of the mesh represents a domain-global optimized scalar function. If the strongest displacement is applied on the specific part of the mesh boundary, the question remains how to propagate the displacement to the rest of the mesh while keeping the distortion minimal in the regions of no interest for mesh motion. The two most prominent approaches for propagating mesh motion throughout the solution domain are the algebraic displacement interpolation and the solution of a Laplace equation for the displacement.

The diffusion of displacements from the mesh boundary into the mesh is modeled using the Laplace equation

$$\nabla \cdot (\gamma \nabla \mathbf{d}) = 0, \tag{13.1}$$

where γ is the displacement diffusion coefficient, and $\mathbf{d}$ is the point displacement field. The diffusion coefficient γ varies spatially as a function of the distance between the point $\mathbf{x}$ and the mesh boundary, i.e.

$$\gamma = \gamma(r). \tag{13.2}$$

In equation (13.2), r is the distance between the mesh point and the mesh boundary, and the coefficient function is prescribed in a way that makes the coefficient decrease with the distance from the boundary. Equation (13.1) is approximated using the FVM in OpenFOAM. In that case, the fields solved are cell-centered, with the boundary fields stored at face centers. In order to calculate the displacements in the mesh points (cell corner points) using the FVM, an interpolation from cell-centered values to the mesh points needs to be performed. Alternatively, in foam-extend, solution to the equation 13.1, can be approximated using the Finite Element Method (FEM).
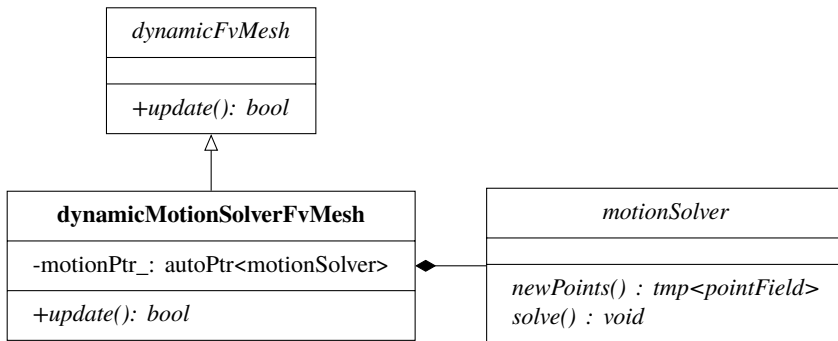
```
┌─────────────────────────────┐
│       dynamicFvMesh          │
├─────────────────────────────┤
│                              │
├─────────────────────────────┤
│     +update(): bool          │
└─────────────────────────────┘
```

```
┌──────────────────────────────────┐        ┌──────────────────────────────────────┐
│  dynamicMotionSolverFvMesh        │        │            motionSolver                │
├──────────────────────────────────┤        ├──────────────────────────────────────┤
│ -motionPtr_: autoPtr<motionSolver>│◆──────│                                        │
├──────────────────────────────────┤        ├──────────────────────────────────────┤
│  +update(): bool                  │        │ newPoints() : tmp<pointField>          │
│                                   │        │ solve() : void                         │
└──────────────────────────────────┘        └──────────────────────────────────────┘
```

Figure 13.3: Mesh motion implemented by `dynamicMotionSolver-`
`FvMesh` class.

The mesh deformation which involves a solution of the Laplace equation
is implemented by `dynamicMotionSolverFvMesh`, shown in figure 13.3.
The motion of the moving body is described by its boundary, which
is done by assigning a motion boundary condition to the corresponding
part of the body boundary mesh. The motion boundary condition can
be given as an explicit function of velocity or displacement, or it can
be calculated based upon external data. For example, a data-driven mesh
motion boundary condition may use data computed to solve solid body
motion driven by fluid forces integrated on the body boundary.

When a body is moving only slightly relative to a large solution domain,
the displacement of outer patches is usually set to a fixed zero value
displacement boundary condition that basically fixes all the points on
these domain boundaries in space.

Once the boundary motion is prescribed with a boundary condition for
the displacement (velocity) field, the motion solver takes over the respon-
sibility of solving for the motion field inside the solution domain. The
`motionSolver` is composited within the `dynamicMotionSolverFvMesh`
and is implemented as an abstract base class to allow different approaches
to solving for the motion field (e.g., finite volume equation solution, or
algebraic interpolation of displacement). The concept of the motion solver
is required to generate new mesh points, and nothing else - the motion of
the mesh points is then further delegated to the parent `fvMeshClass`:

```
bool Foam::dynamicMotionSolverFvMesh::update()
{
    fvMesh::movePoints(motionPtr_->newPoints());
```

```
if (foundObject<volVectorField>("U"))
{
    volVectorField& U =
        const_cast<volVectorField&>(lookupObject<volVectorField>("U"));
    U.correctBoundaryConditions();
}

return true;
}
```

and the motion solver (`motionPtr_`) is only required to generate new mesh points with the `newPoints` member function:

```
Foam::tmp<Foam::pointField> Foam::motionSolver::newPoints()
{
    solve();
    return curPoints();
}
```

Calling `newPoints` causes the approximation of the mesh deformation which modifies mesh points. The solution process implemented by `solve` will be different depending on using the interpolation or the Laplace equation to propagate displacements. Different motion solvers are available, but describing all of them is out of scope.

A Finite Volume solver for the displacement field is chosen as an example, namely the `displacementLaplacianFvMotionSolver`. The two most important classes that interact with the Laplacian finite volume mesh motion solver are shown in figure 13.4. The `displacementMotionSolver` encapsulates the displacement fields necessary for the mesh deformation and provides field-access to other displacement solvers. The `motionDiffusivity` is a Strategy (i.e. an OpenFOAM model) used to compute a variable diffusivity coefficient, given by equation 13.2. This spatially variable field can then be used to scale the displacement which has been propagated to the internal parts of the mesh by interpolation or as a coefficient in the diffusion equation 13.1. A new function for the mesh-motion diffusivity can be developed easily: a class that inherits from `motionDiffusivity` and registers itself to its RTS table automatically becomes available in the mesh motion framework.

> **INFO**
>
> Although we have omitted some details of mesh motion in OpenFOAM in this section, the available information should be sufficient for understanding and extending mesh motion.
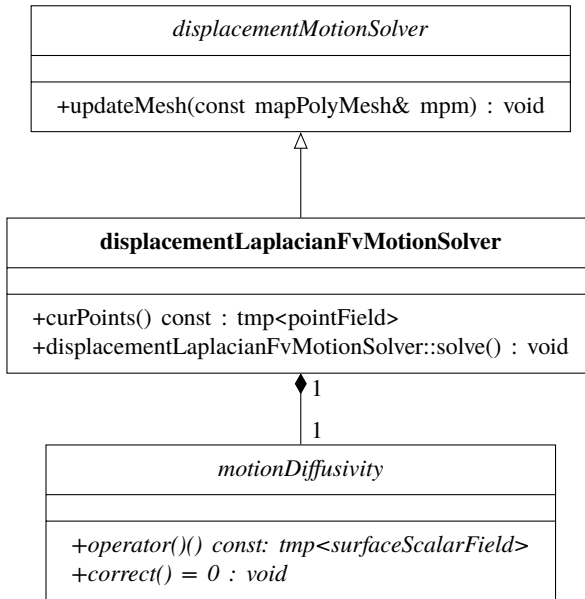
Figure 13.4: Laplacian finite volume based motion solver class diagram.

The Laplacian finite volume mesh motion solver implemented by `dis-`
`placementLaplacianFvMotionSolver`, shown in figure 13.4, solves the
diffusion equation for the cell centered displacement field in the `solve`
member function:

```
diffusivityPtr_->correct();
pointDisplacement_.boundaryField().updateCoeffs();

Foam::solve
(
   fvm::laplacian
   (
      diffusivityPtr_->operator()(),
      cellDisplacement_,
      "laplacian(diffusivity,cellDisplacement)"
   )
);
```

and the point displacements are then interpolated using Inverse Distance
Weighted (IDW) interpolation from the cell centers, to the cell corner
points (mesh points) within the `curPoints` member function:

```
volPointInterpolation::New(fvMesh_).interpolate
(
   cellDisplacement_,
```

```
        pointDisplacement_
    );
```

INFO

A new IDW interpolation object is allocated each time when the `cur-Points` is executed - when the mesh is deformed using the FVM based motion solver.

Creating a new IDW interpolation object each time step is necessary as the inversed distance interpolation weights change when a relative displacement exists between mesh points.

13.1.2 Topological changes

Changing the mesh topology involves modifying its topological information: adding and deleting mesh elements (cells, faces, edges, points) and updating all the data structures that describe the mutual connectivity between mesh elements (e.g., the edge-cells connectivity list). Applying topological changes to the mesh is usually motivated either by increasing the accuracy in the domain regions where large gradients are present or by modeling dynamical systems whose simulation domains experience extreme changes in shapes and sizes. Mesh deformation may cause severe mesh quality degradation as the angles between cell faces and ratios of neighboring cell sizes can be severely modified. Severe mesh deformation causes the loss in accuracy of the unstructured FVM because cells become distorted in a way that introduces interpolation, non-orthogonality, and skewness errors. The mesh deformation may be coupled with topological changes of the mesh to maintain accuracy, resulting in a more accurate and efficient dynamic mesh engine.

The topological changes in OpenFOAM are implemented as individual operations, and they can be agglomerated when designing dynamic mesh classes. Dynamic meshes that deal with changing the mesh topology usually are more specialized than the mesh motion classes because the topological changes are generally more complex. The complexity lies in adding and removing mesh elements, changing their connectivity, and updating the field values to account for the topological change. The `dynamicFvMesh` library, located at `$FOAM_SRC/dynamicFvMesh` contains general purpose topological changers, such as `dynamicRefineFvMesh`.

This dynamic mesh implements dynamic and local refinement of a hexahedral mesh based on a value interval for the refinement criterion (e.g., the pressure gradient). The user defines the refinement criterion value interval, and the cells that contain field values that lie inside of a user prescribed interval get dynamically refined, and those that store values outside that interval get unrefined. Using different fields as refinement criteria is possible as well. For example, the refinement criterion may use a more complicated refinement model, calculated during the simulation in a user-developed function object.

More specialized topological changers are available in

`$FOAM_SRC/topoChangerFvMesh`

and the `movingConeTopoFvMesh` serves as an example: it models the dynamic simulation of a moving piston within a cylinder. The mesh moves a patch (a part of the mesh boundary) in the direction of the x-axis. It adds and removes cell layers in the region where the cells get highly deformed to maintain high mesh quality. The ability to develop such a specialized dynamic mesh in OpenFOAM is a result of the separation of abstraction levels. Single topological operations are on the lower abstraction level, and their agglomeration results in an operation that adds cell layers to the mesh, on the higher abstraction level.

Figure 13.5 shows important class relationships of the `dynamicRefine-FvMesh` class. As other dynamic mesh classes, it implements the `update` member function to conform to the `dynamicFvMesh` interface. The mesh refinement operation involves refinement (splitting) and un-refinement (merging) of cells, depending on the above-described refinement criterion. Both operations are implemented by two private member functions `refine` and `unrefine`, respectively. Each time either of the two operations are executed the `dynamicRefineFvMesh` class collaborates with the parent `fvMesh` class in order to update the fields for topological operations.

The algorithm for mesh refinement/un-refinement is shown, with a reduced level of detail, in algorithm 2. Mes refinement and un-refinement generate a *topological map*: data that relates the modified mesh elements with the original mesh. Splitting mesh faces of a cubic cell results in polyhedral cells in OpenFOAM: these cells are still cubical but have more than six faces. As long as two adjacent cells share the same number of sub-faces generated by splitting the original face, the owner-neighbor
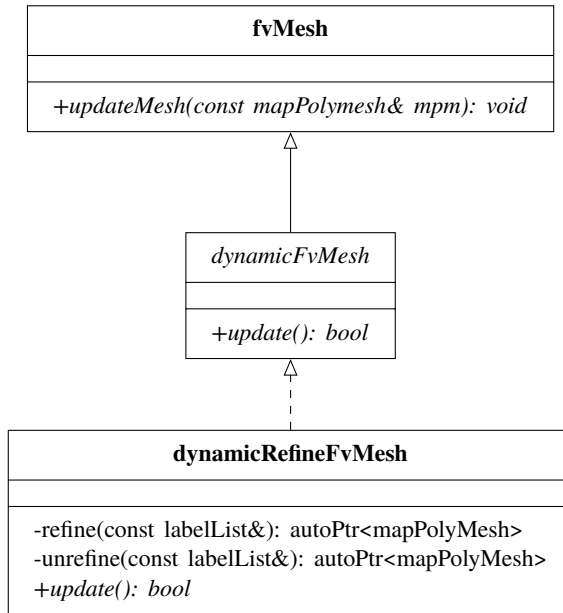
Figure 13.5: Mesh refinement implemented by the `dynamicRefine-FvMesh` class.

INFO

The collaboration between the `dynamicRefineFvMesh` and `fvMesh` is achieved by calling the `fvMesh::updateMesh` member function. The mapping of fields may be a complex algorithm to implement. When a new class for topological changes is implemented, it may be easier to re-use this algorithm by adhering to the class relationship shown in figure 13.5

indirect addressing of the mesh (chapter 1) will work without modification. However, it is important to note that such splitting introduces non-orthogonality, aspect ratio, and skewness errors. Thus, it is common to provide an *additional layer of refined cells*: the boundary of the refined cell layer should lie in the region with small gradients, where high accuracy is not as important as within the refined cell layer.

Algorithm 2 Mesh refinement and unrefinement algorithm

Read the control dictionary for the `dynamicRefineFvMesh`.
if not first time step **then**
    Look up the refinement criterion field.
    if number of mesh cells < maxCells **then**
        Select cells to be refined (`refineCells`).
        Add cells from the refinement layer and protected cells to `refine-`
`Cells`.
        if cells to be refined > 0 **then**
            refinementMap = refine(cellsToBeRefined)
            update refineCell (refinementMap)
            updateMesh (refinementMap)
            mark the mesh as changed
        end if
    end if
    select unrefinement points
    if number of unrefinement points > 0 **then**
        refinementMap = unrefine (unrefinement points)
        updateMesh (refinementMap)
        mark the mesh as changed
    end if
end if

INFO

Adaptive refinement of hexahedral unstructured meshes in OpenFOAM *is not* based on an octree data structure. The mesh topology is changed directly and stored in the new mesh, enabling the application of existing discretization operators and schemes on the topologically modified mesh.

13.2 Using dynamic meshes

Any solver that contains `DyMFoam` in its name can use any dynamic mesh available in OpenFOAM. The prerequisite is the `dynamicMeshDict`, which must be present in the `constant` folder in the simulation case, and be configured properly.

Two examples are provided in this section and each relates to the mesh motion types described in the previous section: global mesh motion using `solidBodyMotionFvMesh` and mesh deformation based on `dynamicMotionSolverFvMesh`. Both examples utilize the same base mesh, which
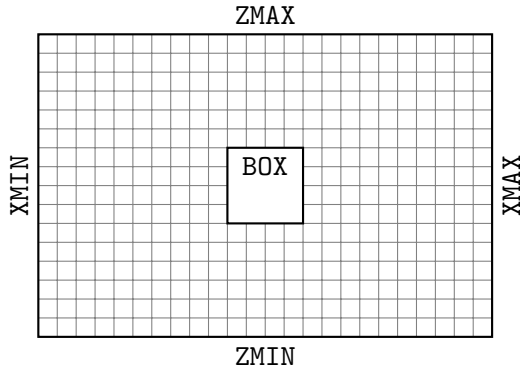
Figure 13.6: Sketch of the geometry in the $x - z$ plane, used in the unitCubeBase cases.

is a unit-cube immersed in a larger cubical domain. In both cases, the inner cube performs the same motion, but each example computes the mesh motion differently. The two examples prescribe a linear translational motion and use the solidBodyMotionFvMesh and the dynamicMotion-SolverFvMesh dynamic mesh class, respectively. The example base cases are located in the example case repository under chapter13/unitCube-Base_globalMotion and chapter13/unitCubeBase_patchMotion.

Setting up OpenFOAM cases that use the dynamic mesh can be tedious work, especially if the flow solver takes up a lot of computational time during a time step. The tool moveDynamicMesh avoids waiting for a long time to check if the case is configured correctly: it performs all the steps the solvers make when using dynamic meshes without the expensive flow solution step. Hence, only mesh. update() is executed inside the time loop, which triggers the mesh modifications performed by the runtime selected dynamic mesh class. This approach works well, as long as the motion is not dependent on any data that results from the flow simulation.

> INFO
>
> Extending a simulation case with dynamic mesh operations is faster with the moveDynamicMesh application because it requires shorter execution times than the flow solver.

13.2.1 Global mesh motion

As outlined in the beginning of this section, global mesh motion can be achieved by using the `solidBodyMotionFvMesh` in conjunction with a `solidBodyMotionFunction` of any choice. To achieve an linear translational motion, `linearMotion` is used in this example. Similar to all other tutorial cases described in this book, the first step is to copy the tutorial to a location where it can be edited safely. To do so, change into the example case repository and locate the `chapter13` directory:

```
ofprimer > cp -r cases/chapter13/unitCubeBase_globalMotion $FOAM_RUN
ofprimer > run
```

The `dynamicMeshDict` is the only configuration file that needs to be adjusted, if the mesh motion needs to be changed. As can be seen from the following excerpt, it employs the `solidBody` motion solver to define the mesh motion class and the `linearMotion` defines the motion function.

```
dynamicFvMesh      dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver          solidBody;

solidBodyMotionFunction linearMotion;

velocity (1 0 0);
```

The above dictionary instructs `solidBody` to obtain the transformation from `linearMotion` and apply it to all mesh points, rather than a zone. Before being able to test the motion, the mesh needs to be generated. Therefore execute the following two steps, but be aware that the second step will generate time step directories in the case folder:

```
?> blockMesh
?> moveDynamicMesh
```

The second call executes `moveDynamicMesh` which in turn takes care of the mesh motion and provides several lines of output to the screen per time step. Depending on the configuration of the `controlDict`, the time step directories are generated at various frequencies. With the current case configuration in the example case repository, the data is written at every 0.05 seconds and can be inspected using paraView. Chapter 4 provides description on how to use paraView for visualizing OpenFOAM results, the whole mesh simply moves in the direction of the x-axis.

13.2.2 Mesh deformation

In comparison to the global mesh motion example presented in the previous section, mesh morphing is a little more demanding, in terms of configuration efforts that have to be spent. A prepared versionof this example case can be found at `chapter13/unitCubeBase_patchMotion`. The added `constant/dynamicMeshDict` is responsible for the mesh motion, a new field is added to the `0` directory and an additional solver entry needs to be inserted in the `system/fvSolution`. The last two entries depend on the *type* of the mesh motion solver, and they have been briefly addressed in section 13.1. For this example, a displacement based mesh motion solver is selected. Therefore, the added new field is named `pointDisplacement`, which is a `pointVectorField` and the respective boundary conditions need to be set for this field:

```
dimensions      [0 1 0 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
   "(XMIN|XMAX|YMIN|YMAX|ZMIN|ZMAX)"
   {
      type           fixedValue;
      value          uniform (0 0 0);
   }
   HULL
   {
      type           solidBodyMotionDisplacement;

      solidBodyMotionFunction linearMotion;
      linearMotionCoeffs
      {
         velocity (1 0 0);
      }
   }
}
```

Rather than defining a motion only for the box itself, a velocity is assigned to the outer boundaries as well, in order to illustrate the capabilities of the `dynamicMotionSolverFvMesh`. The box has the same velocity as for the previous tutorial, but it is reduced to 0.75 for the remaining patches. Of course, this setup has a practically limited time of execution, as the box will compress the cells too much with its motion. Still, this setup serves as an example of how various motions can be assigned to different boundaries.

The `dynamicMeshDict` of the previous example needs to be changed as well and should look like the following:

```
dynamicFvMesh      dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver             displacementLaplacian;

displacementLaplacianCoeffs
{
    diffusivity       inverseDistance (HULL);
}
```

As already mentioned earlier, a new solver entry in `system/fvSolution` is required. A `GAMG` type solver with a `GaussSeidel` smoother is selected for this purpose. The following entry must be added to the `solvers` sub-dictionary of `fvSolution`:

```
cellDisplacement
{
    solver        GAMG;
    tolerance     1e-5;
    relTol        0;
    smoother      GaussSeidel;
    cacheAgglomeration true;
    nCellsInCoarsestLevel 10;
    agglomerator   faceAreaPair;
    mergeLevels    1;
}
```

With the addition of the `0/pointDisplacement` boundary file, the above entry into `fvSolution` and the described changes to the `dynamicMesh-Dict`, the `unitCubeBase` example can be executed again.

```
?> rm -rf 0.* [1-9]*
?> moveDynamicMesh
```
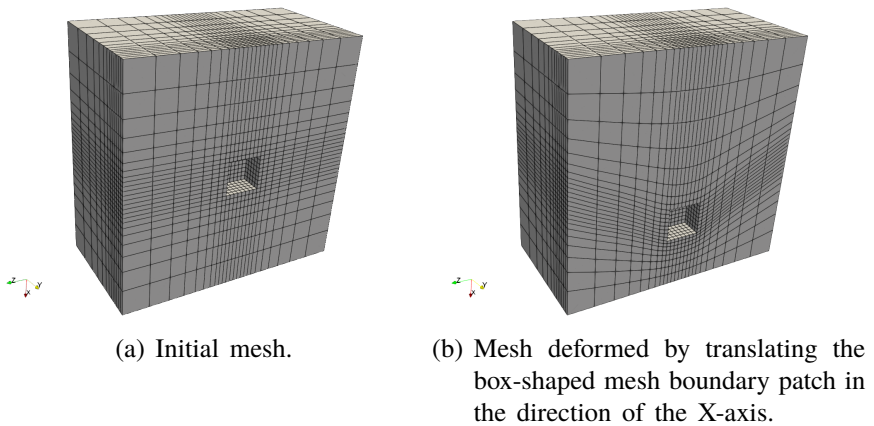
The mesh deformation results are shown in figure 13.7.

(a) Initial mesh.

(b) Mesh deformed by translating the box-shaped mesh boundary patch in the direction of the X-axis.

Figure 13.7: Mesh deformation with a prescribed motion applied to the mesh boundary.

13.3 Developing with dynamic meshes

This section covers the extension of OpenFOAM solvers with dynamic mesh capability and the development of new dynamic mesh classes.

13.3.1 Adding a dynamic mesh to a solver

Extending an OpenFOAM solver with dynamic mesh capability is a fairly straightforward process, and the `scalarTransportFoam` serves as an example. Since the extension for dynamic mesh handling requires modifying particular solver application files, files that require modification must be copied.

> INFO
>
> When extending OpenFOAM solvers, and in general when programming, reduce the amount of copied source code as much as possible.

Copying source code increases the time required for maintenance. If a bug is present in the original file, it must also be resolved in the copy. As OpenFOAM evolves, unnecessary source code copies make it difficult to follow upstream changes. Additionally, complex OpenFOAM solvers contains files used for initializing global variables and for the solution

algorithms for coupled PDEs. Suppose the entire solver folder is copied, and the modification to the solver is minor and placed in a single file. In that case, it becomes difficult to find out the difference between the new and the original solver. Therefore, the "common" practice of blindly copying the entire solver folder when writing a new OpenFOAM solver is, in the long run, inefficient.

The first step when programming a new OpenFOAM solver is, therefore, to create the solver folder and copy only the necessary files to the solver folder

```
?> mkdir scalarTransportDyMFoam && cd scalarTransportDyMFoam
?> cp $FOAM_SOLVERS/basic/scalarTransportFoam/scalarTransportFoam.C .
?> cp -r $FOAM_SOLVERS/basic/scalarTransportFoam/Make .
```

Even for the relatively simple `scalarTransportFoam` solver, it is possible to re-use the `createFields.H` file. With more complex "real-world" solvers, many more solver files can be re-used, leading to a cleaner implementation. File re-use is enabled by appending the path to the original solver in the `Make/options` OpenFOAM build configuration file:

```
EXE_INC = \
    -I$(FOAM_SOLVERS)/basic/scalarTransportFoam \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -ldynamicMesh \
    -ldynamicFvMesh \
    -lsampling
```

The first "include" line leads to the folder of the OpenFOAM solver using the `$FOAM_SOLVER` environment variable. If the folder of the original solver contains sub-folders that contain other necessary files, those can be included as well. The other lines listed in the above code snippet are there for including dynamic mesh headers and loading dynamic mesh libraries.

Since this example creates a new solver, the `scalarTransportFoam.C` file should be renamed to `scalarTransportDyMSolver.C`, following the OpenFOAM convention for solvers with dynamic mesh support. Then,
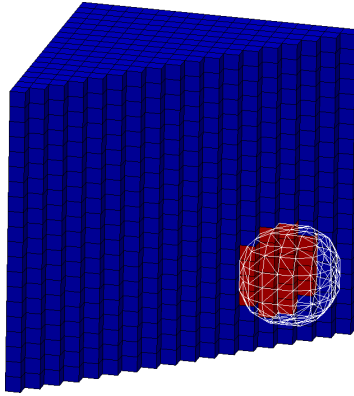
Figure 13.8: Test case for scalar transport with mesh refinement. Red cells are initialized with the value 100, and blue cells with value 0, the wireframe sphere is an iso-surface with the iso-value of 50.

all occurences of `scalarTransportFoam` in `Make/files` should be replaced with `scalarTransportDyMFoam`. It is important to compile the solver executable into `$FOAM_USER_APPBIN`, rather than the standard `$FOAM_APPBIN`, in `Make/files`.

The new `scalarTransportDyMFoam` solver in its current state does nothing new, compared to the original solver. This can be tested using the simulation case `chapter13/scalarTransportAutoRefine`.

Figure 13.8 shows the initial conditions and the domain geometry for the test case used with this example. The test case consists of a cubical domain with an initial field preset as a sphere in the corner of the domain and a constant velocity field used to transport the field in the direction of the spatial diagonal.

Having verified that `scalarTransportDyMSolver` works as expected, the next step is implementing the dynamic mesh capability in the new solver. To do so, `scalarTransportDyMSolver.C` must be opened in a text editor and `dynamicMesh.H` must be included after `simpleControl.H`:

```
#include "fvCFD.H"
#include "fvOptions.H"
#include "simpleControl.H"
#include "dynamicFvMesh.H"
```

In the `main` function `createDynamicFvMesh.H` rather than `createMesh.H` has to be included, in order to enable dynamic meshes:

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
```

A closer look at the sources of `scalarTransportFoam` reveals that there is no call to `mesh.update()`, which is responsible for executing the dynamic mesh functions. This must hence be added at the end of the time loop:

```
    mesh.update();
    runTime.write();
}

Info<< "End\n" << endl;

return 0;
```

The dynamic mesh header files and respective library binary code must be made available during the build process for the new solver to work. To include the headers and link the libraries, the `Make/options` file needs to be modified by adding the following "include" lines:

```
-I$(LIB_SRC)/dynamicMesh/lnInclude \
-I$(LIB_SRC)/dynamicFvMesh/lnInclude \
```

The application is linked with dynamic mesh libraries by adding the following "linking" lines to `Make/options`:

```
-ldynamicMesh \
-ldynamicFvMesh \
```

The `dynamicRefineFvMesh::update` member function corrects the volumetric flux values for the new faces generated by the mesh refinement procedure. This process is named *flux mapping* and can be governed by the user by modifying the *flux mapping table* placed in the `constant/dynamicMeshDict`. Information on mesh refinement is available in section 13.1. However, the mapped volumetric fluxes are a good enough initial guess for the flow solution algorithm, that enforces volume conservative volumetric flux values. For a scalar transport equation in this example, no flow solution algorithm is necessary as the field is advected using a constant prescribed velocity.

Compiling and running the solver at this point leads to a numerically unbounded solution. To solve this problem, we insert the last line of code into the solver application, which mimics the presence of a flow solution algorithm, and computes volume conservative volumetric flux values. The line is inserted just below the call to the `update` member function:

```
mesh.update();
phi = fvc::interpolate(U) & mesh.Sf();
```

The code is now ready to be compiled - by calling `wmake` in the solver folder. The new solver can be tested in the

```
ases/chapter13/scalarTransportAutoRefine
```

in the case folder. Figure 13.9 shows the transported spherical scalar field T shown in figure 13.8 with dynamic adaptive mesh refinement. The advection of `T` is diffusive, and the mesh refinement and de-refinement resolve the diffusive region and follow the transported field.

> **INFO**
>
> By refining the mesh, new cell faces are created, which requires the modification of the volumetric flux field. The volume conservation, numerical boundedness as well as solution stability depend strongly on the volumetric flux values.

> **EXERCISE**
>
> The refinement criterion used for this test case refines the interior of the transported sphere uniformly. An interesting exercise would be to apply a refinement criterion that would follow the jump in values of `T`. This would decrease computational costs and increase accuracy.
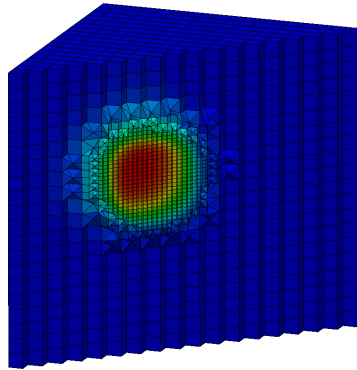
Figure 13.9: Scalar field transported with dynamic mesh refinement.

13.4 Summary

The overview of the design of different dynamic mesh classes from the representative mesh handling categories, and the descriptions of the dynamic mesh usage, coupled with the extension of a solver for dynamic mesh handling in OpenFOAM should provide a good starting point for developing dynamic mesh handling in OpenFOAM. More complex tasks such as developing lower-level topological changes and their agglomeration into entirely new dynamic mesh classes in OpenFOAM are more complex and are out of scope for this book.

Further reading

[1] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[2] R. Goldman. *Rethinking Quaternions: Theory and Computation*. Morgan & Claypool, 2010.

[3] Jasak. "Error Analysis and Estimatino for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science, 1996.

[4] H. Jasak and H. Rusche. "Dynamic mesh handling in openfoam". In: *Proceeding of the 47th Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, Orlando, Florida.* 2009.

[5] H. Jasak and Z. Tukovic. "Automatic mesh motion for the unstructured finite volume method". In: *Transactions of FAMENA* 30.2 (2006), pp. 1–20.

[6] F. Juretić. "Error Analysis in Finite Volume CFD". PhD thesis. Imperial College of Science, 2004.

14

Conclusions

Although many core parts of OpenFOAM are covered in this book, there are still many interesting topics left open, such as the implementation of new FVM schemes and operators, parallel programming with Open MPI in OpenFOAM, developing new Euler-Lagrange models and methods for particle-laden and spray flows, development of new dynamic mesh classes, developing new constitutive laws for compressible flows, and much more. Nevertheless, the software design of those OpenFOAM elements resembles the content covered in this book and is based on the repetition of a handful of software design patterns in the C++ programming language outlined throughout the text. A thorough understanding of the material presented here and the basic understanding of software design patterns in C++ - and not avoiding the hands-on work on the examples and exercises - presents a solid basis for further developing OpenFOAM.