Community Experience Distilled

# Getting Started with OpenFOAM® Technology: RAW

Learn how to program CFD applications using OpenFOAM® with the help of sample projects provided in this practical tutorial

Tomislav Maric Kyle Mooney
Jens Höpken

[PACKT] PUBLISHING open source* community experience distilled

# Getting Started with OpenFOAM® Technology

RAW Book

Learn how to program CFD applications using OpenFOAM® with the help of sample projects provided in this practical tutorial

**Tomislav Maric**
**Jens Hopken**
**Kyle Mooney**

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

# Getting Started with OpenFOAM® Technology

Current RAW Publication: July 2013

RAW Production Reference: 1290713

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-176-9

www.packtpub.com

# Table of Contents

# Preface

Welcome to Learning OpenFOAM® the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

Learning OpenFOAM® is a practical tutorial that will show its readers how to program CFD applications using OpenFOAM® with the help of sample projects. This book is for C++ developers who want to develop CFD programs with the help of OpenFOAM® library. No knowledge of OpenFOAM® or computational fluid dynamics (CFD) is expected, although readers are expected to have some experience in C++ programming.

## What's in This RAW Book

In this RAW book, you will find these chapters:

Preface

The preface contains short description of the OpenFOAM® library/application bundle and what it is used for in general. It also includes a discussion of the advantages of the code being open source versus a commercial black box package.

Chapter 1: Computational Fluid Dynamics in OpenFOAM®

This chapter provides a general overview of the workflow involved with CFD simulations using OpenFOAM®. A basic introduction of the Finite Volume Method (FVM) supported within OpenFOAM® is provided with references pointing the reader to further information sources on this topic. An overview of the toolkit organization is presented as well as the interaction of the organizational elements within the scope of a CFD simulation.

Chapter 2: Geometry Definition, Meshing, and Conversion

This chapter covers domain decomposition and discretization. This includes defining a geometry, mesh generation, and mesh conversion.

## What's Still to Come?

We mentioned before that the book isn't finished, and here is what we currently plan to include in the remainder of the book:

Chapter 3: OpenFOAM® Case Setup

This chapter describes the structure and the setup of a simulation case. This involves setting the initial and boundary conditions, configuring the run control parameters of a simulation, and numerical solver settings.

Chapter 4: Post-Processing, Visualization, and Data

This chapter gives an overview of the utilities used for pre-processing and post-processing calculation as well as instructions on how to visualize computed data of a simulation.

Chapter 5: Design Overview of the OpenFOAM® Library

This chapter provides a more detailed overview of the library than the one presented in chapter 1. In this chapter the reader will learn how to browse the code and where to find the building blocks of the library.

Chapter 6: Productive Programming with OpenFOAM®

This chapter describes how to program with OpenFOAM® in a productive and sustainable way. This chapter will be important for readers interested in programming with OpenFOAM® who may lack a software development background. This chapter covers the development of self-sustained libraries, a way of using the git version control system, debugging and profiling, and so on.

Chapter 7: Turbulence Modeling

This chapter introduces turbulence modeling into a simulation case. This involves setting up a turbulence model and its parameters.

Chapter 8: Writing Pre- and Post- Processing Applications

This is the first chapter that involves programming from the reader's side. Here we show how to develop pre- and post-processing applications using both C++ applications as well as commonly used utilities available for this purpose.

Chapter 9: Solver customization

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

Chapter 10: Boundary conditions

This chapter 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. As a result, the reader will develop a library of boundary conditions which is dynamically linked to the client code (a solver application).

Chapter 11: Transport models

This chapter covers the numerical background, design and implementation of transport models. As an example, an implementation of a temperature dependent viscosity model is provided.

Chapter 12: Function objects

This chapter introduces the use of function objects within OpenFOAM®. The background of function objects in C++ is provided, as well as a list of references for further study. The implementation of function objects in OpenFOAM® is described, in addition to an instructional programming example.

Chapter 13: Dynamic Meshes

This chapter shows how to extend a solver with the functionality of the dynamic mesh in OpenFOAM®. The available dynamic mesh engine in OpenFOAM® is very powerful and enables the readers to build their own dynamic mesh objects by agglomerating exiting ones.

Chapter 14: Outlook

This chapter gives an outlook of further advanced usage and programming topics with OpenFOAM®.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "It is assigned a `jQuery` object containing the results of the `$('.sticky')` query."

A block of code will be set as follows:

```
StickyRotate.init = function() {
  var stickies = $(".sticky");

  // If we don't have enough, stop immediately.
  if (stickies.size() <= 1 || $('#node-form').size() > 0) {
    return;
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
StickyRotate.init = function() {
  var stickies = $(".sticky");

  // If we don't have enough, stop immediately.
  if (stickies.size() <= 1 || $('#node-form').size() > 0) {
    return;
  }
```

Any command-line input and output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

**New terms** and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

## What Is a RAW Book?

Buying a Packt RAW book allows you to access Packt books before they're published. A RAW (Read As we Write) book is an eBook available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

## Is a RAW Book a Proper Book?

Yes, but it's just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final version is published.

## When Do Chapters Become Available?

As soon as a chapter has been written and we are happy for it go into the RAW book, the new chapter will be added into the RAW eBook in your account. You will be notified that another chapter has become available and be invited to download it from your account. eBooks are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

## How Do I Know When New Chapters Are Released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new eBook. Packt will also update the book's page on its website with a list of the available chapters.

## Where Do I Get the Book From?

You download your RAW book much in the same way as any Packt eBook. In the download area of your Packt account, you will have a link to download the RAW book.

## What Happens If I Have Problems with My RAW Book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

## Is There Source Code Available During the RAW Phase?

Any source code for the RAW book can be downloaded from the **Support** page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

## How Do I Post Feedback and Errata for a RAW Title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawfeedback@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

# 1 Computational Fluid Dynamics in OpenFOAM®

*"There cannot be a greater mistake than that of looking superciliously upon practical applications of science. The life and soul of science is its practical application..." – Lord Kelvin*

Before interfacing with the OpenFOAM library itself, we will look at a few aspects of the implemented flavor of finite volume CFD as well as discuss the typical workflow of a CFD engineer. A competent and capable OpenFOAM user should have a good handle on both the numerics of the flow solvers, and the nature of fluid flow physics itself.

What is covered in this chapter:

- Characteristics of non-dimensional numbers
- Some important concerns when approaching a CFD problem
- The CFD workflow
- A review of the finite volume method in OpenFOAM®
- A summary of the contents of the OpenFOAM® library itself

## Understanding The Flow Problem

The first step in any CFD analysis is to gain a fundamental understanding of the flow under consideration. More specifically, we must consider the thermo-physical phenomenon at play, what engineering assumptions we will be making, and how they will complicate or simplify our analysis. Thermo-physical as well as pragmatic questions that might be posed before undertaking a CFD analysis project are outlined below. This list is by no means exhaustive.

- General
  - What is the engineering or scientific data that we intend to take away from this analysis?
  - To what degree of accuracy do we need our results?
  - How will we confirm the validity of our results?
  - How much human engineering time is willing to be dedicated to the project?
- Thermo-physics
  - Is our flow laminar, turbulent, or transitional?
  - Is our flow compressible or incompressible?
  - Does the flow involve multiple phases or fluid species?
  - Do heat transfer and temperature play a role?
  - Do we know enough about the up/downstream or far-field conditions to accurately define boundary conditions?
- Geometry and mesh
  - Can we construct an accurate, discrete representation of the geometries of interest?
  - Will the computational domain be deforming or moving during the simulation?
- Computational Resources
  - How long are we willing to wait for an answer?
  - What kind of distributed computing resources are available?
  - Will one simulation run suffice or is this a multi-run parametric analysis?

The answers to these questions will dictate, from top to bottom, the tools and methods used to complete a proper CFD analysis of any flow problem. Skills in using OpenFOAM® or any commercial fluid simulation package are rendered moot without a proper understanding of fundamental flow physics, numerical methods, and computer hard- and software. This interdisciplinary nature of CFD greatly contributes to its complexity.

The typical method by which a flow is characterized is through the calculation of relevant non-dimensional groups. These groups serve to quantify the relative significance of participating physical phenomena as opposed to looking at them in absolute terms. A few common dimensionless groups are the Reynolds, Weber, Froude, Capillary, or Ohnesorge numbers, the details and definitions of which can be found in many fluid mechanics and heat transfer texts (see Incropera and DeWitt (2001); Wilcox (2007)). An example for this non-dimensionalization is the flow around a $90^{\circ}$ degree pipe bend. Regardless of the fluid being molasses or Nitrogen, the flow will be identical as long as the Reynolds numbers are equal. Most fluids based academic publications communicate results strictly in terms of dimensionless groups to provide results independent of the experimental setup and scale.

While this type of fundamental fluid mechanics analysis is best left to the above mentioned existing texts, its importance in CFD development is not to be discounted. A CFD engineer must be able to determine which physical phenomenon in a flow can be neglected, which need to be modeled, and, if so, to what degree of detail. These decisions will more often then not be made with a careful examination of dimensionless groups while drawing upon past experience and intuition.

## Stages of a CFD Analysis

An analysis based on CFD methods can usually be divided into 5 major components. Some of these steps must be performed multiple times in a loop in order to obtain results of the desired high quality.

## Problem Definition

Before sitting down in front of a computer, one must decide on the physics of the problem that is the focus of the current work. You have to be aware of the flow characteristics that you are planning to simulate and which flow features need to be resolved accurately enough and which can be neglected without sacrificing too much accuracy.

## Mathematical modelling

After having defined the problem properly, it has to be formulated mathematically, with regards to the assumptions previously made. For example: a potential flow is solely governed by Laplace's equation (see Ferziger and Peric (2002)). If more details must be resolved, the mathematical modeling must get more elaborate. This usually leads to more sophisticated mathematical models, such as Reynolds-Averaged Navier-Stokes Equations (RANSE), that account for viscous effects, unsteadiness and turbulence. The latter is treated in a time averaged manner. For more details on particular mathematical fluid models, please confer fluids text books such as Ferziger and Peric (2002); Kundu, Cohen, and Dowling (2011); Versteeg and Malalasekra (1996). More information on turbulence modeling can be also obtained from *Chapter 8*. The mathematical model describes the details of the flow. This means that the numerical simulation, which approximates the solution of the model, cannot produce more information about the flow than described by the model itself.

After having decided on a mathematical model, you can choose your OpenFOAM® solver accordingly. This text covers an overview of the structure, basic usage and the introduction to programming with OpenFOAM®, so the reader is directed to OpenFOAM® User Guide (2012) for additional information on various solvers.

## Pre-processing and mesh generation

The fields used in the simulation need to be initially prescribed. These values set prior to simulation start are typically referred to as initial conditions. If the field values are spatially varying, different utilities may be used to compute and pre-set the fields. There are utilities that are distributed along with OpenFOAM® (e.g. the setFields utility), as well as utilities which are a part of a supporting project (e.g. funkySetFields utility of the swak4foam project). The use of some of the available pre-processing utilities is explained in *Chapter 9*.

[ More information on the swak4Foam project can be found on http://openfoamwiki.net/index.php/Contrib/swak4Foam. ]

Except for a few specific applications, it is generally impossible to solve the governing equations of the mathematical model in an analytical manner. The flow domain must hence be discretized. This spatial discretization consists of separating the flow domain volume into a computational mesh consisting of volumes (cells) of different shapes. Based on the decisions on the model, the mesh (or grid) must be tailored for this particular purpose.

Usually, the mesh must be refined in areas of interest: e.g. where large gradients of flow variables (velocity, pressure, density, etc.) occur. Further on, the accuracy and a proper choice of the mathematical model has to be kept in mind, as resolving flow features in a spatial manner does not compensate for a model that does not account for these features in the first place.

The mesh is one of the most likely components of the simulation work flow that need to be changed if the numerical simulation fails to converge. Failing simulations very frequently are caused by a mesh of insufficient quality.

OpenFOAM® comes with two different mesh generators: blockMesh and snappyHexMesh. The usage of both is covered in *Chapter 3*. 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.

## Solving

The solution step is commonly the most time consuming part of the entire CFD analysis process, although no user interaction is required. Based on the choice of model, an appropriate solver needs to be selected or created and subsequently executed. The chosen mathematical models are then computed according to user-selected solution methods and residual error tolerances.

## Post-processing

After the simulation completes, the user often ends up with a large amount of data that must be analyzed and discussed. The data must be visualized appropriately in order to inspect the details of the flow. Data such as velocity fields, which is a three-dimensional vector field, is impossible to visualize using simple two-dimensional graphs. By using dedicated scientific visualization tools such as Paraview, such data can be discussed and interpreted fairly easily.

[ Paraview may be downloaded for free from www.paraview.org ]

Post-processing and the next step, discussion and verification, typically go had in hand. More details on various post-processing tools and methods are provided in *Chapter 9*.

## Discussion and verification

This is the point where you will have to determine whether to trust your results or not. The code solely does what the user instructs and cannot do any magic. You have to keep in mind that if a mistake was made one of the previous steps, you will most likely and hopefully discover it during discussion.

Though it is advisable to have some data to compare your results to, such as experiments, this is not very likely to be the case in industrial applications. Therefore you have to build up a certain level of trust and confidence in the work you did to obtain them. If you are not satisfied with the results, you must revisit the previous steps until you are.

# The Finite Volume Method in OpenFOAM®

This section provides a very brief overview of the Finite Volume Method (FVM) in OpenFOAM® used for CFD. The reader is directed to Ferziger and Peric (2002); Jasak, Jemcov, and Tukovi c (2007); Versteeg and Malalasekra (1996); Weller, Tabor, H. Jasak, and Fureby (1998) for further details regarding this topic, as it is beyond the scope of this book to cover it with sufficient depth.

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 are dependent variables in a mathematical model: a formal mathematical description of the fluid flow. A mathematical model that describes a fluid flow in three dimensions is defined as a system of partial differential equations. Seemingly different physical processes are sometimes described using the same mathematical description, e.g. the conduction of heat as well as diffusion of sugar concentration in water are modeled as diffusive processes. The scalar transport equation (see Ferziger and Peric 2002) holds the terms often used in mathematical models, and is used exemplary to describe the FVM:

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

Equation 1.1

With φ being the scalar property, U the velocity vector 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 (cf. Ferziger and Peric 2002).

Depending on the nature of the process, some of the terms may be neglected: e.g. for the inviscid fluid flow we neglect the diffusive term (transport) of the momentum, so this term is removed from the momentum equation. In addition, the coefficients that appear in some of the terms may be constant values, or spatially and/or temporally varying fields themselves, depending on the physics of the flow. An example of such a coefficient is a temperature dependent conductivity coefficient for conductive heat transfer: $\cdot (k\ T)$, which makes k a spatially (and possibly temporally) varying field that would depend on the system solution (the temperature field).

The purpose of any numerical method is to obtain an approximation of a solution of the mathematical model. A numerical approximation of the solution of a complex physical process is necessary, since the exact solution can only be obtained for a few very specific cases that often lack relevance in technical applications. An example for this is a Couette flow model (a flow between two infinite plates where the top plate translates with constant velocity), for which an analytical solution exists.

Usually, the solution of the mathematical model is obtained by solving a discrete approximation of the governing system of equations. The approximation process of a numerical method involves a substitution of the system of partial differential equations (the mathematical model) with a corresponding system of algebraic equations that can be solved. These algebraic equations are evaluated at discrete points in space and not continuously. The generation of an algebraic system of equations within the framework of the FVM is made up of two major steps: domain discretization and equation discretization.

## Domain discretization

As previously stated, the mathematical model uses continuous fields that describe the flow field at any point in space. In order to solve the equations of the mathematical model, this continuous space must be discretized into a finite number of volumes (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 $\Omega$, to a discretized domain $\Omega D$ is shown on Figure 2.1. Continuous fields that are defined in each point of the space filled by the fluid on Figure 2.1a are replaced by discrete fields which stored in the centres of the finite volumes C, as shown on Figure 1.1b. Each finite volume stores an averaged value of the physical property (e.g. temperature) in its cell centre C.

Taking all of the cell centres together, the discrete field for the particular physical property can be assembled (e.g. discrete cell-centred temperature field).

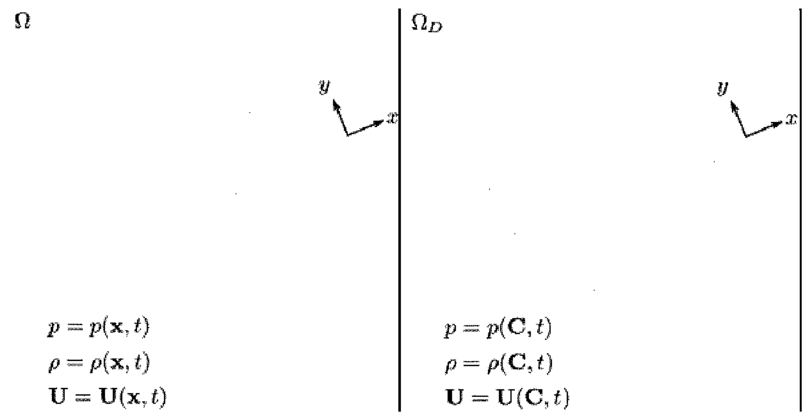
$\Omega$
$y$
$x$
$p = p(\mathbf{x}, t)$
$\rho = \rho(\mathbf{x}, t)$
$\mathbf{U} = \mathbf{U}(\mathbf{x}, t)$
$\Omega_D$
$y$
$x$
$p = p(\mathbf{C}, t)$
$\rho = \rho(\mathbf{C}, t)$
$\mathbf{U} = \mathbf{U}(\mathbf{C}, t)$

Figure 1.1: Continuous and discretized flow domain and corresponding flow fields. Left: Continuous. Right: Discretized.

There are different ways of discretizing the domain, which result in topologically different finite volume meshes. Topologically different domain discretizations determine how the components of the mesh (cells, faces, points) are connected with each other. More information on mesh generation can be found in *Chapter 3*.

We distinguish between three major types of meshes: structured, block-structured and unstructured meshes. Please note that all three have different requirements for domain discretization, equation discretization, and the way the source code must be formulated. The mesh topology and related access optimizations of the mesh may have a direct impact on the accuracy and efficiency of numerical operations performed by the numerical library as well as how the algorithms are parallelized, which is the case in OpenFOAM®.

Structured meshes support direct addressing of an arbitrary cell neighbour as well as direct cell traversal: the cells are labeled with the indices increasing in the directions of the coordinate axis (see Figure 1.2a). Unstructured meshes on the other hand have no apparent direction (see Figure 1.3a) in the way the cells are addressed, and their topology will be a result of a geometrical algorithm used to discretize the computational domain, which is un-ordered by nature.

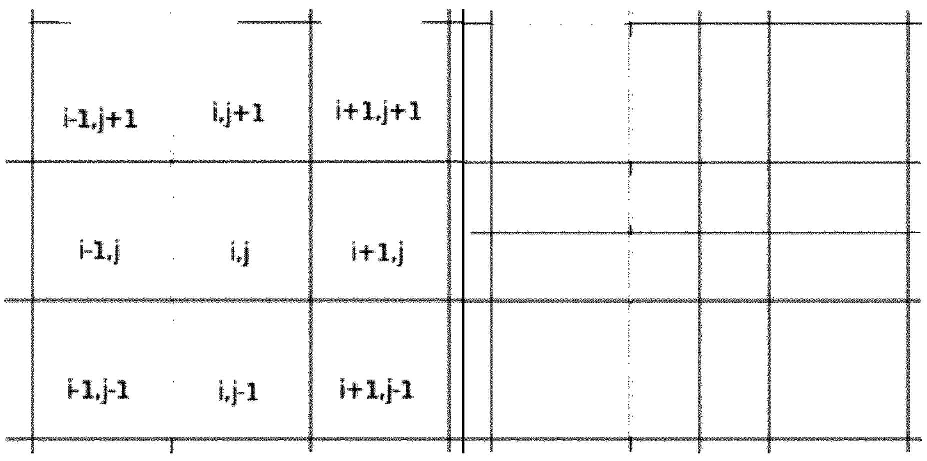

Figure 1.2: A structured equilateral mesh

Structured mesh topology 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 gradients occur and not to waste cells in flow regions where no such sharp gradients occur, while generating the mesh in the shortest possible time. This is impossible with structured meshes, as 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 schematic sketch for this is given in Figure 1.2b.

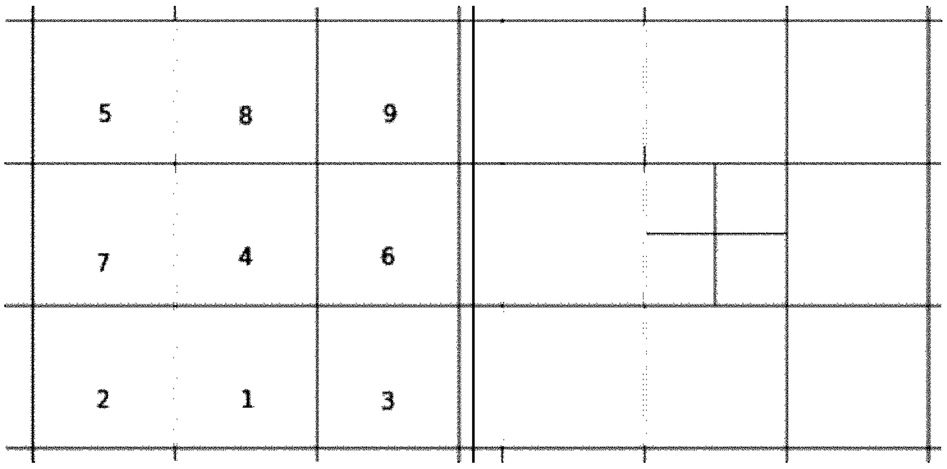

Figure 1.3: An unstructured 2D Catesian mesh.

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

There is one problem, however: What happens if we want to refine the mesh locally in a region of extreme changes of the physical property, when the order of interpolation still is not sufficient to capture the large jump of a physical property? Such large jumps in the values of physical properties are present e.g. in 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 from Figure 1.4. A good example for such a flow regime is a water-air two-phase system with a ratio of densities being approximately 1000 (see Ubbink 1997, Rusche 2002).

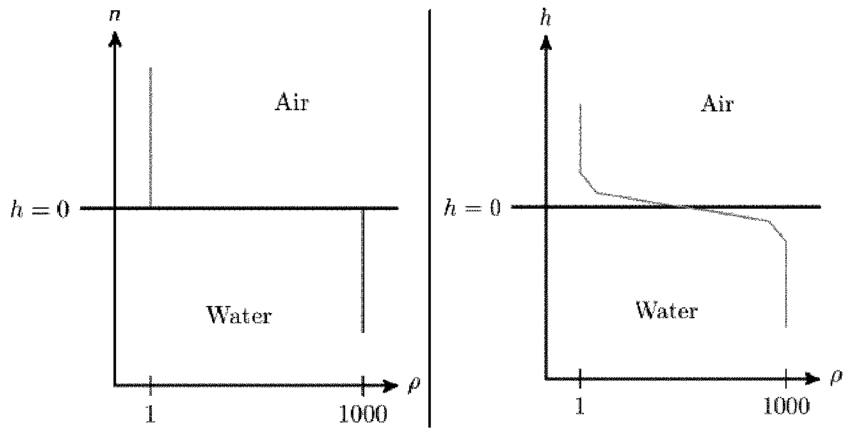

Figure 1.4 Qualitative distribution of the density rho with respect to the height h over a free surface. Left: Continuous space with sudden density jump. Right: Discretized space with gradual yet steep jump in density.

To resolve such steep gradients in the fields, local mesh refinement is often applied. This refinement can either be done during pre-processing or applied adaptively during runtime. As mentioned previously, refining a structured mesh cannot be done locally: the topology of a structured mesh forces us to refine in the complete direction, as shown on the Figure 1.2b, where the refinement of a single cell in two directions generates refinement throughout the mesh. Structured meshes that conform to curved geometries are especially difficult to generate, since a mathematical parametrization of curved domain boundaries (coordinatization) is necessary in order to maintain structured mesh topology.

In order to increase the accuracy locally (meaning in a only a subsection of the domain), block refinement may be done, 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. The numerical method must either be able to deal with non-conforming block patches (hanging nodes), or the block refinement needs to be carefully crafted, so that the points on adjoining blocks of different densities match perfectly (patch-conforming block-meshes). Building block structured meshes is a complex problem even for simple flow domains, which makes block-structured meshes a poor choice for many technical applications involving complex geometries of the flow domain. Refining block-structured meshes results in refinement regions spreading through the blocks, and with standard solvers that rely on patch-conforming block-meshes, the refinement complicates the mesh generation even more.

Dynamic adaptive local refinement of structured meshes is preformed 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 data structure is used to split the cells of the structured Cartesian mesh into octants. Information carried 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 cell-cut approach, where 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 initial domain to be box-shaped. More information about local adaptive mesh refinement procedure can be found in the book Adaptive Mesh Refinement Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3-5, 2003 (Lecture Notes in Computational Science and Engineering) 2005.

OpenFOAM® implements a FVM of second order of convergence with support for arbitrary unstructured meshes. Arbitrary unstructured means that in addition to the unstructured mesh topology, the mesh cells can assume an arbitrary shape. This allows the user to discretize flow domains of very high geometrical complexity. The unstructured mesh allows for a very fast, sometimes automatic mesh generation procedure, which is very important for industrial applications where the time needed to obtain results is of great importance. Hence, unstructured meshes are still a main choice of domain discretization for numerical simulations of industrial interest where the flow domains are geometrically complex.

Figure 1.3a shows a two-dimensional schematic of a quadratic unstructured mesh. Since the mesh addressing is not structured, the cells have been labeled solely for the purpose of explaining the mesh topology. The un-ordered cells complicate the possibility to perform operations in a specific direction without executing costly additional searches and re-creating the structure of the mesh locally with respect to the given direction. Another advantage of the unstructured mesh is the ability for a cell to be refined locally and directly, which is shown on the Figure 1.3b, which results in a locally refined mesh of geometrically complex flow domains. The local refinement is more efficient in terms of increase of the overall mesh density, since it only increases the mesh density where it is required.

How then does the numerical method find its way around a cell in order to operate on the values of neighboring cells when assembling the system of algebraic equations, where neighboring values must be accessed from each cell?

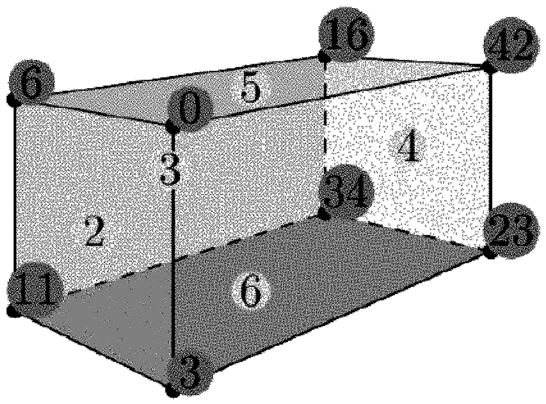

Figure 1.5 Example of a hexahedral cell. Red and green labels indicate point and face indices respectively. Face 1 is ommitted for the sake of visibility.

## Element Addressing

The way mesh elements are addressed by the algorithms of the numerical method is determined by the mesh topology. OpenFOAM® defines its topological mesh structure using: indirect addressing, owner-neighbour addressing and boundary mesh addressing.

## Indirect addressing

Indirect addressing defines both the cells and faces of the mesh as sets of indirected indexes to mesh points. The face is defined as an ordered set of indices that map to the list of mesh points. This means that a face is defined by the indices of its points rather then by its points directly. A cell is built accordingly and consists of an unordered set of indices that map to the list of mesh faces. Indirect addressing 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 topological operations.

In order to clarify this we consider face 2 in Figure 1.5. It consists of points 0, 3, 6, and 11 and doesn't know anything about the locations of these points as it just referred to the points by their indices. The same goes for the hexahedral cell that consists of faces 1 to 6 and does not store any point related data directly. Each face of the cell can be accessed using the index to the particular face stored in the cell, that relates to the list of mesh faces.

## Owner-neighbour addressing

Owner-neighbor addressing is an optimization which defines the way the indices in the mesh faces are ordered, by setting the direction of the face area normal vector Sf shown as an arrow on Figure 1.6. Two global lists are introduced into the mesh with owner-neighbour addressing optimization: the face-owner and the face-neighbour list. For each face of the mesh, there may be only two adjacent cells defined with one cell being the face-owner cell (marked with P on Figure 1.6) and the other a face-neighbour (marked with N on Figure 1.6). The owner cell of a face will be the cell with a lower index in the list of mesh cells. This information determines the ordering of the face indices: the face area normal vector is directed always from the owner into the neighbour cell. Switching the orientation of the face area normal is an efficiency optimization which is done to reduce redundant computations in the equation discretization step described in the following subsection.

## Boundary addressing

Boundary addressing is an optimization driven by face access efficiency and object oriented design, that isolates the boundary faces and stores them at the end of the list of mesh faces. This allows an efficient definition of the boundary conditions as slices (patches) of the list of mesh faces, which, in turn, results with separated operations for the internal mesh faces and the boundary faces. The boundary mesh is defined as a set of patches (sets of boundary faces), which can be interpreted as different physical boundary conditions, or even processor boundaries in parallel simulations. Such a definition of the boundary mesh results enables the automatic parallelization of all the top-level code in OpenFOAM® that relies on the face-based interpolation practice. All the faces of the boundary mesh are directed outwards from the flow domain which means that they have only a cell owner and no neighbour.

A schematic sketch of how cell-centred values of the unstructured mesh are addressed by the face owner-neighbour addressing mechanism is shown in Figure 1.6. As the considered cell with the index 1 has three faces that are taken into account in this example, it has 3 neighbouring cells. This cell has the lowest index (1) of all its neighbours (2,3,4) and is thus listed in the face-owner list. Each of its neighbouring cells need to know that the face to the cell with index 1 is still used, but not owned by each particular cell. Therefore these cells are listed in the face-neighbour list. This results in the face area normal vectors being oriented outward from the cell 1 to all of its neighbours in this example.

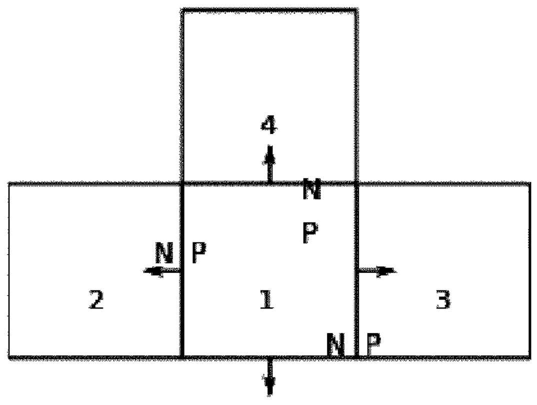

Figure 1.6 Owner neighbour addressing for an example cell (1).

Additional addressing, such as cell-cells and point-cells, is also stored in the unstructured mesh in OpenFOAM®. Both are used to provide easy access to all face-adjacent cells of a target cell and the cells whose edges meet at the specified point, respectively. The actual indexing of all the connectivity is a direct result of the mesh generation algorithm.

$$\int_t^{t+\Delta t} \int_{V_P} \nabla \cdot (\mathbf{U}\phi)\, dx\, dt = \int_t^{t+\Delta t} \int_{\partial V} \phi \mathbf{n}\, do\, dt \approx \sum_f \phi_f \mathbf{U}_f \mathbf{S},$$

Equation 1.5

## Equation discretization

Once the domain is discretized into finite volumes, approximations are applied on the terms of the mathematical model which transfer the differential terms into discrete differential operators. In contrast to the domain discretization, this is done during runtime in each solver in OpenFOAM®. An exception are solvers that employ dynamic meshes, which may perform the domain discretization repeatedly. Detailed descriptions of the equation discretization in OpenFOAM® are provided by Jasak (1996); Jureti c (2004); Rusche (2002); Ubbink (1997). Here we describe only the discretization of a simple advection equation for a scalar property φ with the velocity U without source terms:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi) = 0$$

Equation 1.2

Equation 1.3 has two terms: the temporal term and the advective term. Both terms need to be discretized in order to obtain the algebraic equation, since the equation cannot be solved in the existing form analytically. The numerical method must be consistent (see Ferziger and Peric 2002): as the size of the cells we generated in the domain discretization step is reduced, the discrete (algebraic) mathematical model must approach the exact mathematical model. Or in other words, as described by Ferziger and Peric (2002), refining the computational domain infinitely and solving the discretized model on this spatial discretization leads to the solution of the mathematical model consisting of partial differential equations. To obtain the discrete model, Equation 1.3 is integrated in time and space:

$$\int_t^{t+\Delta t} \int_{V_P} (\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi))\, dx\, dt = 0.$$

Equation 1.3

Integration of the temporal term of Equation 2.3 can be approximated to:

$$\int_t^{t+\Delta t} \int_{V_P} \frac{\partial \phi}{\partial t} \, dx \, dt \approx V_P \frac{\phi^n - \phi^o}{\Delta t},$$

Equation 1.4

where VP is the cell volume, n and o mark the time-step of new and the old the simulation respectively, and Δt denotes the time-step value. The time-step is introduced since time is to be discretized as well into a sequential finite intervals (time-steps). The advective term is discretized by integrating and applying the Gauss divergence theorem:

Where ∂V marks the continuous boundary of a finite volume VP (a surface bounding the volume VP) with the area differential do, f marks a face of a cell, and S marks the outward-pointing face area normal vector. The face area normal vector is a vector normal to the cell face with the magnitude of the face area. Face centered values must be computed from adjoining cell centres in order to proceed with the computation of the right hand side of Equation 1.5. This is done using interpolation (we will explain that at the end of this chapter) and the interpolated values are marked by the index f , e.g. φf . When we consider two discrete algebraic terms, Equation 1.3 takes the following discrete form:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f \mathbf{U}_f \mathbf{S} = 0.$$

Equation 1.6

It is easy to see that in the limit where both Δt and VP tend to zero, the discrete Equation 1.6 corresponds to the continuous mathematical model shown in Equation 1.3.

As you might have observed, no indices marking the new (n) or the old (o) time step are present in Equation 1.5. This is a result of neglecting variations of the face interpolated values in time. Depending on the choice of the new or the old time-step for the final term of Equation 1.5, the resulting algebraic equation will be solved explicitly or implicitly. The differences are presented in the following:

## Explicit temporal discretization

If we evaluate the spatial terms in the old time-step, the only value from the new time-step is the value stored in the centre of the volume for which the algebraic equation is assembled:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f^o \mathbf{U}_f^o \mathbf{S} = 0,$$

Equation 1.7

and in this case, we can simply put all the old values on the right hand side (r.h.s.) of the equation and compute the new cell value φn by explicitly evaluating the r.h.s. of the equation.

## Implicit temporal discretization

In case we choose to evaluate the r.h.s. in the new time-step:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f^n \mathbf{U}_f^n \mathbf{S} = 0,$$

Equation 1.8

the algebraic equation assembled for the cell in question will carry dependent variables from the surrounding cells in the new time-step, which means that we need to assemble such an equation for each cell of the mesh to construct the system of algebraic equations and solve that system to get the entire cell-centred field in the new time-step. This kind of solution is called an implicit solution.

We can see that the shape of the equation will be determined by the following factors:

- the way cell centred values are interpolated to the faces (Uf and φf) using different interpolation methods,
- the geometrical shape of the cell (especially the number of cell faces) as well as the number and geometry of adjacent cells, since the cell shapes determine the position of the cell centre, and thus have an impact on interpolation,
- the size of the cell: the smaller the VP, the closer the algebraic equation will be to the exact equation (numerical consistency), increasing the accuracy of the solution,
- what terms are present in the equation: we may add a diffusive term and/or a source term, whose discretization will change the values of the coefficients in the final algebraic equation,
- the size of the time-step we use: the smaller time-step results in increased time accuracy for transient problems.

The owner-neighbour addressing is applied when the sum term of Equation 1.6 is evaluated. Should this term be evaluated naively for each cell using the outward directed normal S, the computation would be doubled for each cell face f once the loop reaches the adjacent cell (see Jasak 1996). However, the owner-neighbour addressing allows the FVM method in OpenFOAM to interpolate the face values only once, and then simply apply the same contribution for both adjacent cells:

$$\sum_f \phi_f \mathbf{U}_f \mathbf{S} = \sum_f^{owner} \phi_f \mathbf{U}_f \mathbf{S}_f - \sum_f^{neighbour} \varphi_f \mathbf{U}_f \mathbf{S}_f,$$

Equation 1.9

where the sum is split into the owner sum and the neighbour sum. All the numerical calculations in OpenFOAM® that involve face interpolation are based on looping over mesh faces. Cell values are accessed using owner-neighbour addressing of the cells, that has been described previously. This way the values φf and Uf in Equation 1.9 are interpolated once to the face centres. Their contribution to the discretized term of two adjacent cells is also computed only once in a loop that will add the same contribution to the face-owner cell, and deduct it from the face-neighbour cell. This way a significant amount of computational time is saved.

## Face interpolation

We have mentioned before that the volumes store discrete field values in their centers, and the discrete Equation 1.6 makes use of the face values as well when it interpolates the face values. Evaluating face values is done by using interpolation schemes, which are one of the main building blocks of the OpenFOAM® library. Interpolation schemes use values stored in cell centres C to interpolate the values in the face centres Cf . Using different interpolation schemes for the face centered value, φf will define the form of the discrete equation defined for each cell of the mesh. There are various interpolation schemes available, however explaining all of them is not within the scope of this book. To explain the basic working principle of an interpolation scheme, we have chosen linear interpolation, which is also known as the central differencing scheme (CDS):

$$\phi_f = f_x \phi_P + (1 - f_x)\phi_N,$$

Equation 1.10

where fx is the linear coefficient which is computed from the mesh geometry:

$$f_x = \frac{||\vec{fN}||}{||\vec{PN}||}.$$

Equation 1.11

Equation 1.11 clearly shows what role does the mesh geometry play in the final algebraic equation assembled for a finite volume: large differences in cell size may lead to large errors in the face interpolation, that in turn reflect on the entire system of algebraic equations. A rigorous and detailed derivation of the interpolation errors of the arbitrary unstructured FVM is provided by Jureti c 2004.

The positions of the points P , f and N will be determined with the shape of the cells, and this plays a crucial role for the accuracy and stability of the numerical method. Equation 1.10 introduces the neighbouring cell centre values into the algebraic equation of a cell. To better illustrate how the algebraic equation for a cell is assembled, consider the example of a 2D finite volume with labeled surrounding cells of the unstructured mesh shown on the Figure 1.6. For this volume, the discrete Equation 1.6 takes on the following form:

$$a_1\phi_1^n + a_4\phi_4^n + a_3\phi_3^n = 0.$$

Equation 1.12

In this example, dependent variables are: φ1 , φ4 , and φ3 for the cell 1 when an implicit temporal discretization scheme is applied. The number of the dependent variables in the algebraic equation is determined by the cell shape, since it determines the number of adjacent cells which take part in the assembly of the discrete advective term.

## Boundary conditions

There is one thing missing in this description of the FVM: if the adjacent cells introduce dependent variables into the algebraic equation of a cell, what happens when the cell is adjacent to the domain boundary? Such cell faces are highlighted in red and labeled boundaryField in Figure 1.8. In that case, the variable cannot be made a dependent variable of the system, it needs to be prescribed. This is the reason why we have to set boundary conditions for our simulations, which can be further explained when observing the expanded discrete Equation 1.8.

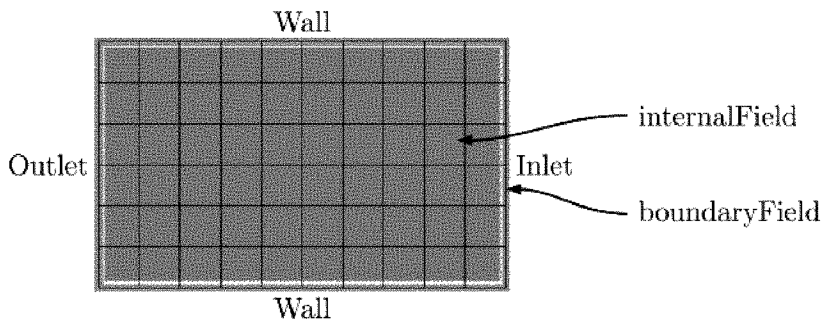

Figure 1.8 Example of a simple 2D channel flow with the inlet on the right side and the outlet on t e left hand side. The other remaining two boundaries are assumed to be walls.

While expanding the sum term, we will come across a cell face which is the boundary face. If we mark this face with b, we will need to compute something like φb Ub Sb and add it to the rest of the sum. Since there is only one cell next to a boundary face, this cell will always be the owner of the boundary face, and hence the normal area vector of a boundary face will always be directed out of the flow domain (out of the owner cell). The values of the physical properties (in this case, the property φ and the velocity U) will either be fixed (e.g. fixed value boundary condition) or they will be computed from the internal cell values (e.g. zero gradient boundary condition). For the fixed value boundary condition, the procedure is simple: e.g. we as a user prescribe the φb value as well as the boundary velocity Ub. The simplest form of the boundary condition that depends on the cell centre values is the so-called Neumann, or "natural" boundary condition, which prescribes a zero gradient of the property at the domain boundary:

$$\nabla\phi(\mathbf{x}_b) = 0,$$

Equation 1.13

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:

$$\phi_P \approx \phi_b + \nabla\phi(\mathbf{x}_b)\delta x \approx \phi_b,$$

Equation 1.14

wnich means nothing more than that the boundary value takes on the value from the single owner cell. The boundary contribution to the algebraic equation 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: a1 in Equation 1.12.

There is a multitude of various boundary conditions implemented in OpenFOAM®: all of them either prescribe the boundary value or boundary gradient in a way. Regardless of how the particular property gradient or value is obtained, applying the boundary condition is basically either defining a value or a gradient on a certain boundary face.

## Solving the system of algebraic equations

Equation 1.6 presents an example of how the partial differential equation like the Equation 1.3 can be converted using equation discretization and interpolation schemes on top of the existing mesh (domain discretization) into an algebraic equation. Implicit temporal discretization will result in an equation being assembled per each cell, that needs to be solved for dependent variables stored in the the cell and its surrounding cells.

A system of algebraic equations generated this way will be very large: its size is directly proportional to the number of cells. One look at the fully expanded algebraic equation for our example cell 1 shows how the cell indexing resulting from the mesh generation process determines the structure of the coefficient matrix of the algebraic system of equations. If we chose a different ordering of cells than the one shown in Figure 1.3a, the coefficients and the dependent variables for the volume 1 would have different indices.

In order to solve an algebraic system of equations, we need a quadratic matrix of coefficients, and no rows or columns may be defined as linear combinations of each other. This means that the true length of the fully expanded example Equation 1.12 will be equal to the number of mesh cells, which results with a large number of equations for the entire mesh, that are assembled and solved in a matrix:

$$\underline{\mathbf{A}} \cdot \mathbf{x} = \mathbf{c}$$

Equation 1.15

where A is the coefficient matrix, x represents unknowns of the system and c denotes the source vector of the system.

Each row represents the connection of the particular cell to the other cells. As one cell does not possess a lot of direct connections to the remaining cells, only few columns in that row do actually have a value. The remaining columns are filled with zeros. Applied to our example, this means that the rest of Equation 1.12 is filled with zeros for all the cells of the mesh that are not related to the example cell 1 by the face-based connectivity that is implemented in OpenFOAM®. This is why the final coefficient matrix A assembled for an unstructured mesh is a sparse matrix: a matrix filled mostly with zeros.

Now that the system is assembled properly, it has to be solved as quickly as possible. In principle we distinguish between direct and iterative methods.

## Direct methods

A popular example for the direct methods is Gauss elimination which obtains the solution of Equation 1.15 in a direct manner, by rearranging the matrix. Unfortunately this number of those rearrangements is proportional to n3, with n being the size of the matrix (see Ferziger and Peric 2002). This renders them unfeasible for large matrices, which usually occur in today's applications. Especially because the common matrix is sparse and after applying the upper triangulation of the Gauss elimination, the matrix is not sparse anymore, which accounts for the slowness of the method.

## Iterative methods

These methods are essential for non-linear problems and opposed to the direct methods, the accuracy is not as good. But this is absolutely fine as the accuracy of the matrix solver must solely be higher than the one of the discretizations. The solving process is started at a guessed solution and refined iteratively. Ferziger and Peric (2002) formulate the system of equations for an iterative solving process like the following:

$$\underline{\mathbf{A}} \cdot \mathbf{x}^n = \mathbf{c} - \rho^n$$

Equation 1.16

The intermediate solution xn after n iterations does not suffice Equation 1.15 and hence a residual ρn has to be introduced. It is always the aim to drive the error towards zero. Since iterative solvers don't solve the system of equations in a absolutely accurate manner, the grade of accuracy must be defined somehow. This is where the residual comes into play, defining the difference between the exact solution and the current iteration.

Finding the most efficient iterative solver for the particular application is necessary if one desires a fast convergence. OpenFOAM® provides a large number of solvers, ranging from preconditioned conjugate gradient (PCG) to more sophisticated ones, such as generalized geometric-algebraic multi-grid (GAMG). To describe each of them in detail would be beyond the scope of this book and the reader is referred to Ferziger and Peric (2002); Saad (2003) for details on this topic.

## Improvement of convergence

As real world applications tend to include various physical phenomenons and are usually of unsteady nature, achieving proper convergence can turn out to be challenging. One common method to improve the convergence of the simulation is to employ under-relaxation.

Under-relaxation means that the user can define a blending factor α between the old and the new solution. An α = 1 means that no effectively under-relaxation is applied and the new solution is completely assembled by the result for this time-step. Setting α = 0 practically disables the development of the solution, as it is entirely composed by the previous solution, so choosing this value is of no merit. Values within the interval (0, 1) define the blending between both solutions used to set the new solution of the system.

## Overview of the organization of the OpenFOAM® toolkit

The OpenFOAM® toolkit consists of many different libraries, stand-alone solvers, and utility programs. In order to establish some orientation around this massive and often intimidating code base, we'll start with simply having a look at the contents of the root OpenFOAM® directory.

## Contents of the /OpenFOAM directory:

- /applications
    - Houses source code for solvers, utilities, and auxiliary 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
    - Houses bash (not C++ binaries) scripts of with a broad array of functions from checking the installation (foamInstallationTest) to executing a parallel run in debug (mpirunDebug) to generating an empty source code template (foamNew) or case (foamNewCase).
- /doc
    - Contains the User's Guide, the Programmer's Guide, and the Doxygen generation files. These are all excellent resources for new users or engineers trying their hand in code development.
- /etc
    - Contains many compilation and runtime selectable configuration controls for the 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
    - Separates and stores compiled binaries 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 vs. Icc)
- 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 primitive 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 from a more in-depth software engineering perspective in *Chapter 5*. There we describe how object oriented C++ programming is used to make OpenFOAM® such a flexible and powerful CFD platform. In the following chapter we will introduce mesh generation and conversion and some associated utilities. Here we will start our first example project which we will develop throughout the rest of the book.

## Summary

This chapter reviewed some important aspects of computational fluid dynamics which form an important foundation for both inexperienced CFD engineers and new OpenFOAM® users. An important idea to take away is that CFD is immensely complicated and no software package can perform magic. The accuracy of your simulation will only be as good as your ability to make proper assumptions and design an intelligent simulation. With that said, we will begin interacting with OpenFOAM® in the next chapter as we learn to generate meshes and explore different options when discretizing a flow domain.

## Bibliography

Adaptive Mesh Refinement - Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3-5, 2003 (Lecture Notes in Computational Science and Engineering) (2005). 1st ed. Lecture notes in computational science and engineering, 41. Springer. isbn: 3540211470.

Ferziger, Joel H. and Milovan Peric (2002). Computational Methods for Fluid Dynamics. 3rd rev. ed. Berlin: Springer. isbn: 3-540-42074-6. doi: 10.1007/978-3-642-56026-2.

Incropera, Frank P. and David P. DeWitt (2001). Introduction to Heat Transfer. 4Th rev. ed. Wiley. isbn: 0471386499.

Jasak, Hrvoje (1996). "Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science.

Jasak, Hrvoje, Aleksandar Jemcov, and Zeljkoˇ Tukovi c (2007). "OpenFOAM: A C++ Library for Complex Physics Simulations".

Jureti c, Franjo (2004). "Error Analysis in Finite Volume CFD". PhD thesis. Imperial College of Science.

Kundu, Pijush K., Ira M. Cohen, and David R Dowling (2011). Fluid Mechanics with Multimedia DVD, Fifth Edition. 5th ed. Academic Press. isbn: 0123821002.

OpenFOAM User Guide (2012). OpenCFD limited.

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

Saad, Yousef (Apr. 2003). Iterative Methods for Sparse Linear Systems, Second Edition. 2nd ed. Society for Industrial and Applied Mathematics. isbn: 9780898715347url: http://www-users.cs.umn.edu/~saad/PS/all_pdf.zip.

Ubbink, Onno (1997). "Numerical prediction of two fluid system with sharp interfaces". PhD thesis. Imperial College of Science.

Versteeg, H. K. and W. Malalasekra (1996). An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach. Prentice Hall.

Weller, H. G. et al. (1998). "A tensorial approach to computational continuum mechanics using object-oriented techniques". In: Computers in Physics 12.6, pp. 620–631. doi: 0.1063/1.168744.

Wilcox, David C. (2007). Basic Fluid Mechanics. 3rd rev. ed. DCW Industries Inc. isbn: 1928729312.

# 2 Geometry Definition, Meshing, Conversion, and Utilities

*"There is geometry in the humming of the strings, there is music in the spacing of the spheres."* — *Pythagoras*

A geometry is essentially a three dimensional representation of the flow region. If you consider 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, solely the details on the outside of the car's body are relevant and need to be resolved sufficiently by spatial discretization. In this chapter we will outline how to create a mesh from scratch, how to convert meshes from third party packages, and illustrate various utilities for manipulating a mesh after creation.

What is covered in this chapter:

- Defining a geometry
- Meshing in **blockMesh**
- Generating meshes with **snappyHexMesh**
- Converting meshes from external formats
- Generating axisymmetric meshes.
- Using mesh modification utilities.

## Geometry Definition

We distinguish between the actual mesh geometry and the geometry that comes out of a Computer Aided Design (CAD) program. Though some words on the general mesh connectivity have been spent in the previous chapter, we would like to give an overview how the actual mesh is stored in the file system. If you don't know of what components an OpenFOAM® case is composed of, please read Chapter 3 before proceeding.

As long as we are dealing with static meshes, the computational grid is always stored in **constant/polyMesh**. It lives in this directory because it is supposed to be constant, hence the constant folder. From a programming point of view it is described as a polyMesh, which is a general description of an OpenFOAM® mesh, with all it's features and restrictions. The **pitzDaily** tutorial of the **potentialFoam** solver serves as an example in the following, which can be found by typing

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

The polyMesh directory must contain the following files that must be filled with data correctly, in order to provide a valid mesh (see **Listing 1** below):

- **points** defines all points of the mesh in a vectorField, with their position in space being specified in meters. These points are not the cell centres **C**, but the corners of the cells. If you would like to translate the mesh by say 1 m into positive *x* direction of the global coordinate system, you would solely have to move each point accordingly. Touching any other structure in the polyMesh sub-directory for this purpose is not required, but we come to that later in Section 2.4.

```
?> cat constant/polyMesh/points
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)
...
)
```

Listing 2: Excerpt of the points file

- One can see from **Listing 2** that it is a list of 25012 points. This list may not be ordered in any way, though it can be. In addition all elements of the list are unique, meaning that no point coordinates can occur multiple times. Accessing those points is performed by defining their position in the vectorField, starting with index 0.

- **faces** composes the faces from the points by their position in the points vectorField and stores them in a list. This position in the vectorField is referred to as it's label. 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 connect by a line to its neighbours (OpenFOAM User Guide 2013). From the points that define the face, the surface area vector $\mathbf{S}_f$ is calculated and the direction is determined by the right-hand-rule. The example shown in **Listing 3** is taken from the **potentialFoam** tutorial **pitzDaily**. It consists of 49180 faces of which only a subset is shown.

  ```
  ?> cat constant/polyMesh/faces
  49180  // Number of faces
  (
    4(1 20 172 153) // Face 0 with it's four point labels
    4(19 171 172 20)
    4(2 21 173 154)
    4(20 172 173 21)
    4(3 22 174 155)
    ...
  )
  ```

  Listing 3: Excerpt of an example faces file

- **owner** is a list (**labelList**) with the same dimension as the list storing the faces. It tells the code that the first face (index *0*) is owned by the cell with the label that is stated in the owner list at index *0*. For our example shown in **Listing 4** this defines faces *0* and *1* as being owned by cell *0* and faces *2* and *3* owned by cell *1*. The ordering of this list is the result of the owner-neighbour-addressing that was presented in the previous chapter.

  ```
  ?> cat constant/polyMesh/owner
  49180
  (
      0
      0
      1
  ```

```
        1
        2
        ...
    )
```

Listing 4: Excerpt of an example owner file

- **neighbour** has to be regarded in conjunction with the owner list. It does not define the owning cell of certain face, but it's neighbour. Again, the working principle of the owner-neighbour-addressing is explained in the previous chapter.
- **boundary** contains all the information on the boundaries (or patches) in shape of a list with nested subdictionaries. An example for our unit cube is shown in **Listing 5**. This information includes the patch name, type, number of faces and the label of the first face of this patch. All faces that are boundary faces must be covered by the boundary description.

From a user's perspective, the last two are not to be touched as this will most certainly destroy the mesh. The first two however, may need to be altered, depending on your workflow. Changing a patch name or type can be done easily in this file, rather than running the respective mesh generator again, which is likely to be a time consuming task.

```
?> cat constant/polyMesh/boundary
   6  // Number of patches
   (
     XMIN  // Name of first patch
      {
       type  patch;        // Type of first patch
       nFaces  2500;       // Number of faces in patch
       startFace  367500;  // Start face label of patch
     }
   ...
```

Listing 5: Excerpt of an example boundary file.

There are several patch types that can be assigned to a boundary. Some of them will be used on a day to day basis, whereas some others won't. We have to distinguish between a patch and the boundary conditions applied on the patches. A patch is an outer boundary of the computational domain and it is specified in the boundary file, hence being a topological property. Each face on a boundary patch does not have a neighbouring cell. In contrast to the patch, boundary conditions are applied on the patches for each field, respectively. If three fields need boundary conditions, a boundary condition must be applied on each patch for each field individually. 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 their derivatives 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 apply wall functions to that patch (see Chapter 7). Preventing a flow through the patch of type wall must still be done in 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 but the symmetryPlane, which has to be done for all fields.
- **empty** In case of a two-dimensional simulation, this has to be applied to the patches that are "in-plane". Similar to the symmetryPlane type, the boundary conditions of those patches have to be set to empty as well. No other boundary conditions will for those patches. It is essential that all cell edges between both empty patches are parallel. Otherwise no two-dimensional simulation is possible.
- **cyclic** If a geometry consists of multiple components that are identically (e.g. a propeller blade or a turbine blade), only one needs to be discretized and treated as if it is located in between similar 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, they act as being coupled physically.
- **wedge** Similar to a cyclic patch only specifically designed for cyclic patches which form a 5 degree wedge

It does not matter how the above mentioned structure of the polyMesh is obtained. This can either be done by importing a mesh from an alternative software, using the mesh generators that come with OpenFOAM® or even by hand.

## 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 done using **snappyHexMesh** but the usage of this mesh generator will be explained later on. The only important thing is that only stereolithography (**STL**) files can be imported. This 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 are using the ASCII one.

Such an example for a surface with only one triangle is given in Listing 6. In this example only one solid is defined, that is named CUBE. A STL file may contain multiple solids that which are defined one after the other. Each of the triangles that compose the surface has a normal vector and three points.

```
solid CUBE
    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 CUBE
```

Listing 6: Example STL file, with only one triangle

No other formats can be imported directly, but need to be converted into STL. The drawback of using ASCII STL files is that their file size tends to grow rapidly with increasing resolution of the surface. Edges are not included explicitly because only triangles are stored in the file. Therefore, extracting feature edges from an STL can be a challenging task.

An advantage of using STL as a file format is that one obtains a triangulated surface mesh, which by definition always has planar surface components (triangles). This in turn simplifies the usage later on in the code (e.g. **snappyHexMesh**).

## Mesh generation

OpenFOAM® comes with two mesh generators: **blockMesh** and **snappyHexMesh**. Both will be addressed briefly in this section and we try to explain how they work and how we can use them for our purposes. The purpose of the mesh generators is to help you to generate the polyMesh files described in the previous section without having to define them by hand. Both mesh generators read in a dictionary file and write the final mesh to **constant/polyMesh**.

This section is subdivided into two major parts. At first we introduce **blockMesh** and **snappyHexMesh** as the major OpenFOAM® mesh generators and explain their working principles based on a minimal example.

## blockMesh

**blockMesh** is started by calling its executable named **blockMesh**. When calling the executable **blockMesh** the **blockMeshDict** is read in automatically from the **constant/ polyMesh** directory, where it must be present in. If it is not found there, **blockMesh** will complain and throw an error.

**blockMesh** generates block-structured hexahedral meshes that are converted into the arbitrary unstructured format of OpenFOAM® . If you recollect the limitations of the mesh generation in an block-structured fashion, you can already guess that it is possible to generate high-quality grids with **blockMesh**, even for fairly complex geometries. But the effort that the user has to spend generating the **blockMeshDict** increases tremendously for complex geometries. For the ordinary user the limitation in handling the **blockMeshDict** is reached quite quickly. All of this makes **blockMesh** a great tool to generate meshes that either consist of a fairly simple geometry, that can be decomposed into blocks, or act as background meshes for **snappyHexMesh**.

An example of a **blockMesh** block is shown in Figure 2.1. Each block consists of 8 corners that are called vertices. The hexahedral block is built from these corners. Edges, as indicated in Figure 2.1, connect the particular 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. Their length and number of nodes on the particular edges has to match. Boundary conditions for the actual simulation will be applied later on those patches.

Though it is possible to generate blocks with less than 8 vertices as well as non-matching nodes on patches (see OpenFOAM User Guide (2013)), 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, for example, an arc does affect the shape of the block edge, but the connection between the final mesh points on that edge remain straight lines.

**Coordinate Systems**

The final mesh is constructed in the global (right-handed) coordinate system, which is Cartesian and aligned with the major coordinate axis: $x$, $y$, and $z$. This leads to a problem when we would like to position and align blocks arbitrarily in space, maybe even twist them. To circumvent this issue, each block gets its own right-handed coordinate system, which by definition does not require the three axis to be orthogonal. The three axis are labeled $x_1$ , $x_2$, $x_3$ (see OpenFOAM User Guide (2013) and **Figure 2.1**). Defining that local coordinate system is done based on the notation shown in **Figure 2.1**: Vertex 0 defines the origin, the vector between vertices *0* and *1* represents $x_1$, $x_2$ and $x_3$ are composed of the vectors between vertices *0* and *2* and *0* and *4*, respectively.

**Node Distribution**

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

$$n_{cells} = n_{nodes} + 1$$

It is up to the user to define in **blockMeshDict** how many cells will appear on a certain edge. The cell distribution on an edge can be uniform or defined by two types of grading. An expansion ratio describes the grading on an edge, which in turn is the size ratio of the last to the first cell on that particular edge. The grading definition on an edge is defined in the OpenFOAM User Guide 2013 as:

$e_r = \frac{\delta_e}{\delta_s}$ 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 is scaled by the following relation:

$r = e_r^{\frac{1}{1-n}}$ where $n$ represents the number of nodes on that particular edge. By combining the two equations, we can calculate the relative position of the $i$-th node on an edge.

$\lambda(r, i) = \frac{1 - r^i}{1 - r^n}$ 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 other cases, simple trial and error usually suffices.

**Defining the dictionary for a minimal example**

As a small example on how the **blockMeshDict** is set up, we are discretizing a cube of 1 $m^3$ in volume. The dictionary itself consists of one keyword and four sub-dictionaries. The first keyword is **convertToMeters** which is usually 1. All point locations are multiplied 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 zeros, which is a tedious task. By setting **convertToMeters** accordingly, we can save some typing. The first line of the **blockMeshDict** should then look like this:

```
convertToMeters 1;
```

Secondly the vertices must be defined. Remember that the vertices in **blockMesh** are different from the points of the created polyMesh, though their definition is fairly similar. For our unit cube example the vertices may look somewhat like this:

```
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)
);
```

We can tell just from having a first glance at it, that the syntax is a list similar to the points in 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 z = 0 plane and the following do the same for the z = 1 plane. Similar to the points in polyMesh, each element is accessed by its position in the list and not by the coordinates. Note that each vertex must be unique and only occur once in the list.

As a next step, the blocks must be defined. An example block definition for the unit cube might look like this:

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

Again this is a list that contains blocks and not a dictionary, due to the round brackets. The definition might look a little 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 $x_3 = 0$ plane

It is possible to obtain a valid block definition by messing up the order of the vertex list in the particular block definition, but the resulting block will either look twisted or uses incorrect global coordinate orientation. In any case, you will notice that as soon as you run **blockMesh**, **checkMesh** and analyze the mesh in a post-processor (e.g. **paraview**).

The second set of round brackets tells **blockMesh** how many cells each direction of the local coordinate system should get. In this case, we settled for a block that contains only one cell. If we would decide to change that to the *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** 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 **simpleGrading** defines the grading for four edges. Sometimes this is not versatile enough, though. And that is where the **edgeGrading** comes into play, which is essentially the same as **simpleGrading**, but you can specify the grading for each of the 12 edge on a hexahedron explicitly. Therefore the last set of brackets would not list 3 numbers, but 3 times 4. Hence each edge can be set individually.

If we would save the **blockMeshDict** right now and execute **blockMesh** afterwards, we would obtain a valid mesh that looks similar to what we have specified. But **blockMesh** would warn us about not having defined any patches, that are put into the **defaultFaces** patch by default. How can we define the patches as we wish? That is done by defining them inside the list called patches and for the example patch *0*, this look like this:

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

This tells **blockMesh** to generate a patch of type patch named **XMIN**, out of the face that is constructed from the vertices *4, 7, 3* and *0*. How the vertices are ordered is not arbitrary. 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 2.3**, with highlighted **XMIN, YMIN** and **ZMAX** patches.

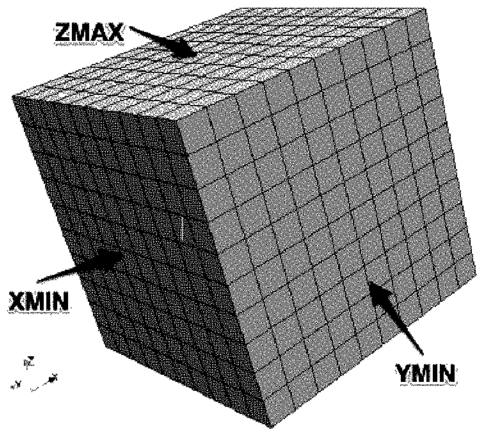

Illustration of a 10x10x10 cubic mesh generated with blockMesh.

As stated earlier, the edges of a block are lines by default and thus the list containing the edge definitions is optional. Quite similar to the above defined blocks and patches, connecting two vertices by e.g. 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, that 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.

An example how inserting the above listed code alters the shape of the unit cube shown in its original shape in **Figure 2.3** is presented in **Figure 2.4.**

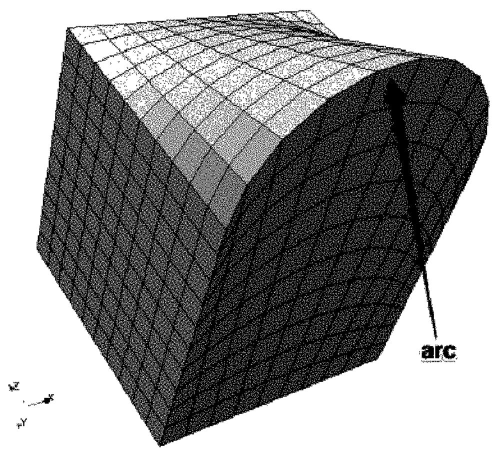

Illustration of a blockMesh block with one edge as an arc.

Now execute **blockMesh** and make yourself comfortable with editing the **blockMeshDict**:

```
?> blockMesh
```

To proceed with the **snappyHexMesh** section, you should end up having a unit cube consisting of 50 cells in each direction.

## snappyHexMesh

Compared to **blockMesh**, **snappyHexMesh** does not require as much tedious work (like adding and connecting blocks). With **snappyHexMesh** hexa-dominant meshes can be generated easily, needing only two things: A hexahedral background mesh and secondly one or multiple geometries as STL files. **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.

With **snappyHexMesh** being a complex program and requiring lots of parameters, describing all them extensively is beyond the scope of this book. Please read the OpenFOAM User Guide (2013) in conjunction with this book. A run of **snappyHexMesh** can be split into three major steps, that 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:

- **castellatedMesh** This is the first step and does essentially two main things. Adding the geometry to the grid and removing all of the cells that are not inside the flow domain. Secondly the existing cells are split according to the user specifications. The result is a mesh that still only consists of hexahedrons, that more or less resembles the geometry. But the majority of mesh points that are supposed to be placed on the geometry's surface are not. A screenshot of a later example at this stage of the meshing process is shown in Figure 2.5.
- **snap** By performing the snapping step, the mesh points in the vicinity of the surface are moved onto this surface. This can be seen in Figure 2.6. During this process, the topology of those cells may get changed from hexahedrons to polyhedrons. Some may get deleted or merged together.
- **addLayers** At last additional cells are introduced on the geometry surface, that are usually used to refine the near wall flow (see Figure 2.7). The already existing cells are moved away from the geometry, in order to create space for the additional cells. Those cells are most likely to be prisms.

All the above mentioned settings and many more are defined in **system/snappyHexMeshDict** that contains all of the parameters required by **snappyHexMesh**. A lot of 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, that 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, though). Each of the three steps will be addressed individually in the following.

Illustration of the three stages of snappyHexMesh applied to a spherical surface: castellated mesh generation, snap, and layer addition.

### 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 2.7). If a cell gets refined by one level, each of the edges gets sliced into half, giving 8 instead of one cell. This way of refining is based on octrees and thus only works 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 get refined - by definition - in all three spatial directions uniformly.

### Defining the geometry

Before we can start the meshing process, the geometry has to 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 anyway and serves as background mesh. Usually for external flow simulations, one does not have small geometrical features of the outer boundaries, that must get resolved. For such cases the dimensions of the outer boundaries defined by the background mesh don't have to get touched and should resemble the desired shape. 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.

As a minimal example, we reuse the unit cube example that we prepared in the previous section and insert a sphere 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 your case and adding the following lines to the geometry subdictionary in your **snappyHexMeshDict**:

| **Shape** | **Name** | **Parameters** |
|---|---|---|
| Box | searchableBox | Min, max |
| Cylinder | searchableCylinder | Point1, point2, radius |
| Plane | searchablePlane | Point, normal |
| Plate | searchablePlate | Origin, span |
| Sphere | searchableSphere | Centre, radius |
| Collection | searchableSurfaceCollection | geometries |

Table 2.1: List of cell selection shapes.

```
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; Other geometry objects can be constructed without the need to open any CAD program, right inside **snappyHexMesh**. A list of these geometrical shapes is compiled in **Table 2.1**.

Any of the mentioned shapes can be constructed in the geometry subdictionary, by simply appending to the existing subdictionary. As an example, we are adding a box to the geometry subdictionary, which is constructed from a minimum and maximum point. This makes it 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** comes into play. By using this on geometry components that already exist, they can be combined into one and even 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 like this:

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

**Setting up the castellatedMesh**

After the geometries have been defined properly, **snappyHexMesh** needs to know what to do with them. How often must surface faces and surface adjacent cells get refined, where are volumetric refinements planned to be placed? Any of those refinements are executed during the first step (**castellatedMesh**) and must hence be defined in the **castellatedMesh** subdictionary. We have to distinguish between refinements that are defined by geometry surfaces and volumetric refinement. With a surface refinement, only the directly adjacent cells get refined to the defined surface level. Applying such a surface refinement to our **SPHERE** would lead to:

```
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 get refined higher than the lesser curved ones.

Refinements in **snappyHexMesh** are not limited to get defined by surfaces. Any geometry defined in the geometry subdictionary can serve as defining shape for a volumetric refinement. These volumetric refinements are called **refinementRegions** and get defined in a subdictionary with the same name, in the **castellatedMesh** controls. Refining anything inside the smallerBox to level 1 can be done by adding the following lines:

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

Each **refinementRegion** must get a mode and a list of levels. The mode can either be inside, outside or distance, which are fairly self-explanatory. Defining the list of refinement levels is a bit trickier, though: Each level must be defined in a pair with a distance, which is 1E15 in the example above. With increasing position in the list, the levels must decrease and the distances must increase.

Without specifying a point located inside the volume of the final mesh, it is impossible for **snappyHexMesh** to decide which part 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 our unit cube example, this point is defined as:

```
locationInMesh (0.989999 0.989999 0.989999);
```

The next step is to tweak the parameters of the snap subdictionary in the **snappyHexMeshDict**.

**Setting up addLayers**

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 we need to define how many cell layers are to be extruded, by specifying it in the layers subdictionary.

```
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. In the above example, we use regular expressions to match any patch names that start with SPHERE , which basically is only the sphere itself. A cross-section of the final mesh is shown in Figure 2.7.

Various parameters of **snappyHexMesh**, related to the layer extrusion, need to get tweaked in order to obtain a mesh that suffices 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 our minimal example, we used the following settings:

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

Finally we just need to execute **snappyHexMesh** in the case to start the meshing process, using **snappyHexMesh**. Each step generates a new time step directory, that contain the mesh of the particular stage. Remember to delete those before restarting **snappyHexMesh**.

## 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.

## Conversion from External Meshing Packages

Many more advanced external meshing utilities offer the user additional levels of control during mesh generation such as selectable element types, fitted boundary layer meshes, and length scale control to name a few. As of OpenFOAM® version 2.1.1, there is support for conversion from many popular meshing programs. In addition, some meshers can export directly to a functional OpenFOAM® mesh format. Listed below is a list of the mesh formats supported for conversion in OpenFOAM® 2.1.1:

- Ansys
- CFX
- Fluent
- GMSH
- Netgen
- Plot3D
- Star-CD
- tetgen
- KIVA

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. The source code for all of the above mentioned conversion utilities are found here: **$FOAM_APP/utilities/mesh/conversion/**. Users also have the option of converting foam meshes into Fluent or Star-CD mesh formats using the **foamMeshToFluent** and **foamToStarMesh** 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 using the **fluentMeshToFoam** conversion utility. To begin the process, start with a new case directory or copy a tutorial case to the directory of your choice. Here we will start with an existing mesh conversion tutorial for the **icoFoam** solver. Copy the case to the directory of your choice, rename, and move into the directory.

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

Now, converting the mesh is as simple as running the conversion utility and passing the mesh file as the argument. During conversion the utility will output patch names and mesh statistics to the console. The polyMesh files will be updated accordingly.

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

Note that when importing a mesh, the case will need to be updated to reflect the new patch names in the initial and boundary condition files. For this tutorial the **U** and **p** fields were pre-configured for this particular mesh. For an arbitrary mesh import, these files (and any other flow variables) will require a manual update to match the patches list in **./ constant/polyMesh/boundary**.

Should you need to scale the mesh during the conversion processes, it is as simple as adding the option and scaling factor to the command. In this case we're reducing the mesh size 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...o patches. While the conversion processes will attempt to match certain boundary condition formats to a corresponding OpenFOAM® format, the user should not assume that the conversion correctly parsed any flow information what so ever, whether it be an internal initial condition, or a boundary condition.

## Converting from 2D to Axisymmetric Meshes

We will show how to create an axisymmetric mesh by starting with a simple example. Here we will convert **icoFoam**'s cavity tutorial case into a wedge. In OpenFOAM® an axisymmetric mesh has the following properties: The mesh is one cell thick (similar to 2D meshes) and is rotated about an axi-symmetry axis to form a 5 degree wedge shape. The two angled faces of the wedge are considered two separate patches of type wedge.

[ Download **makeAxialMesh** here: http://openfoamwiki.net/index.php/Contrib_MakeAxialMesh ]

Make a copy of the case folder to a working directory of your choice, rename the directory to avoid any future confusion, cd into it, and create the 2D base mesh.

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

Now we will run **makeAxialMesh**. We will be converting the movingWall patch into a symmetry axis. In addition, the single frontAndBack patch will be split and act as the two faces of the wedge (frontAndBack_neg frontAndBack_pos). The flags entered into the command line will reflect this.

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

The utility will create a new time file (**./0.005/polyMesh**) to store the transformed mesh which is written to the case directory. The case directory should now contain the folder shown below.

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

Update the main **./constant/polyMesh** mesh with the newly created polyMesh and remove the **./0.005** directory.

```
?> cp -r ./0.005/polyMesh ./constant/
?> rm -r ./0.005/
```

At this point the mesh has been warped into a 5 degree wedge shape (as shown in **Figure 2.8**), however, the faces from the movingWall patch are still present. **makeAxialMesh** transforms the point positions but does not alter the mesh connectivity. Because of this, the symmetry patch faces are now of zero size and must be removed and converted to edges. To do this we will use the **collapseEdges** tool. **collapseEdges** takes two mandatory command line arguments: edge length, and merge angle, as shown here.

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

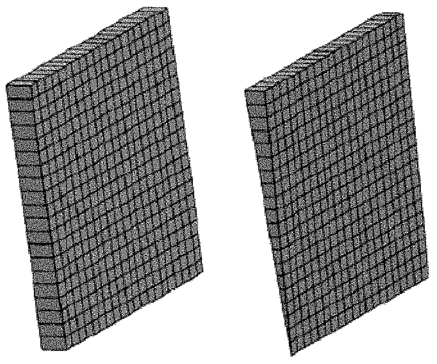

Illustration of a standard 2D mesh before and after wedge transformation.

For many applications an edge length of 1e-8 meters and merge angle of 179 degrees 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 inadvertent removal of valid edges. Run **collapseEdges** with these execution parameters as shown. Update and clean the case as before.

```
?> collapseEdges 1e-8 179
?> cp -r ./0.005/polyMesh ./constant/
?> rm -r ./0.005/
```

For some final housekeeping we will remove the now empty patches from the boundary list. To do this, open the boundary list file contained in **./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 like the example below.

```
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 we could split the fixedWalls patch into 3 separate patches using the **autoPatch** utility. This will look at a contiguous patch and try to identify appropriate places to split it based on a given feature angle. In this case, we will inform the utility that any patch edges that form an angle greater than 30 degrees can be split for form a new patch. This way we will have more freedom when assigning boundary conditions to this case going forward.

```
?> autoPatch -overwrite 30
```

The patches will be renamed after the split. The -overwrite flag will write the split mesh into the **./constant/polyMesh** directory instead of creating a separate polyMesh under a new time folder.

## Mesh utilities in OpenFOAM

The utility applications (utilities) that deal with mesh operations can be found in the directory **$FOAM_APP/utilities/mesh**. The mesh utilities are grouped in the following categories: generation, manipulation, advanced and conversion. Generating the mesh and converting it from different formats into the OpenFOAM® format has been described in Section 2.2 and Section 2.3. In this section we will concentrate on manipulating the mesh as well as advanced operations like mesh refinement.

To start, copy the **damBreak** tutorial to the working directory of your choice, generate the mesh and initialize the $\alpha_l$ field.

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

At this point we have a mesh generated with the **blockMesh** utilities and the α1 field is set using the **setFields** pre-processing utility. The **setFields** utility described together with the pre-processing utilities are in Section. You can use the basic calculator utility **foamCalc** to compute and store the gradient of the $\alpha_1$ field.

```
?> foamCalc magGrad alpha1
```

This will store the cell-centred scalar field of the gradient magnitude in the initial time directory **0** named magGradalpha1. To refine the mesh based on the gradient magnitude using the **refineHexMesh** application we need to copy the configuration dictionary file for this utility 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
```

If you open the dictionary file **system/refineMeshDict**, you will notice a line which specifies a name of the set of cells **cellSet** used for the mesh refinement:

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

This **cellSet**, when created, will be stored in the **constant/polyMesh** and the **refineMesh** application will try to find it when executed in order to figure out which cells are refined. There are two utilities available for creating **cellSets**: **topoSet** and **setSet**. The **topoSet** is a utility that requires a dictionary file to be configured, and is executed on the command line, resulting with the generated **cellSet** stored in the **constant/polyMesh** directory. To create our cell set with **topoSet**, begin by copying an example dictionary into the system folder.

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

Now, replace the example actions subdictionary of **topoSetDict** with the following:

```
        actions
        (
            {
                name    c0;
                type    cellSet;
                action  new;
                source  fieldToCell;
                sourceInfo
                {
                    fieldName magGradalpha1;
                    min 20;
                    max 100;
                }
            }
        );
```

Now we will set the **cellSet** and refine only those cells.

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

The mesh should now had additional resolution in areas of the high alpha gradients. Now we will do the same using the **setSet** utility. The **setSet** utility is interactive, and the user can build and save multiple **cellSets** while working in the command line interface it provides. First, open the **setSet** interface, then enter the **fieldToCell** syntax as shown.

```
?>setSet
?>cellSet c0 new fieldToCell magGradalpha1 20 100
```

You can now refine the mesh using **refineHexMesh** c0 as before.

## 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 remaining stored mesh information is purely connectivity based as discussed previously. With that said, the mesh size, scale, and position can be altered by transforming the point locations alone. To do this we will use the **transformPoints** mesh utility. Because this utility is relatively straight forward, we will not walk though an example but execution syntax is still shown. The most often used options when transforming a mesh are the **-rotate**, **-translate**, and **-scale** options.

The **-scale** option can scale the points in your mesh in any or all cardinal directions by a specified scalar amount. **-scale '(1.0 1.0 1.0)'** will leave your mesh unchanged, while **-scale '(2.0 2.0 2.0)'** will double the size of your mesh in all directions. Any non-unform scaling will stretch or compress your mesh in your given direction(s).

The **-translate** option will move your mesh by the given vector, effectively adding this vector to every point position vector in the mesh.

The **-rotate** option will, you guessed it, rotate your mesh. Define the rotation by inputting two 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 during execution. 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) )'
```

## mirrorMesh

There is a simple way to mirror and join meshes along a planar patch. For this example we will be converting a 1/4 mesh into a full domain. First, copy the following solid analysis case into the directory of your choice and rename it. We must also copy the **mirrorMeshDict** 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 simply define the plane we will be mirroring the mesh about. Define the normal in **mirrorMeshDict** as shown below and run **mirrorMesh**. Patches about which the reflection is taking place are automatically removed.

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

    ?> mirrorMesh
```

Define a new plane and mirror the mesh again.

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

```
    }
?> mirrorMesh
```

We now have a full domain mesh instead of a symmetric fraction as shown in Figure.2.9.

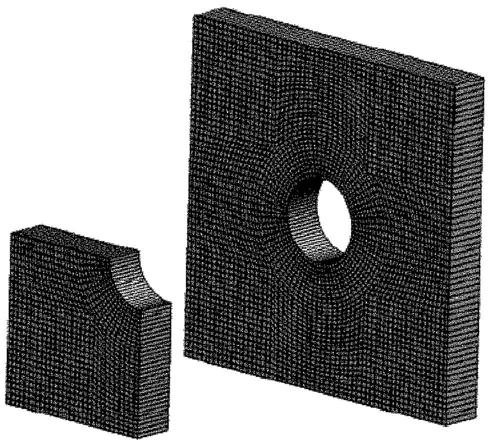

Illustration of a ¼ mesh and the resulting full scale mesh after 2 reflections with mirrorMesh.

## Summary

In this chapter we began interacting with the OpenFOAM® library at a logical first stage: mesh generation. It should be obvious at this point that while there is no "fluid" aspects of mesh generation, it can still be a very cumbersome and complicated process. It is not uncommon to spend considerable time setting up a CFD simulation, only to have an inadequate mesh result in immediate numerical instabilities. It is our hope that between **blockMesh**, **snappyHexMesh**, mesh conversion options, and mesh manipulation utilities, you the user can produce the discretized domain necessary for your CFD applications. In the next chapter we will use our new mesh generation skills and proceed to setup a full OpenFOAM® case and perform our first full CFD calculations.