Fluid Mechanics and Its Applications

F. Moukalled
L. Mangani
M. Darwish

The Finite Volume Method in Computational Fluid Dynamics

An Advanced Introduction with OpenFOAM® and Matlab®

EXTRAS ONLINE

Springer

Fluid Mechanics and Its Applications

Volume 113

Aims and Scope of the Series

The purpose of this series is to focus on subjects in which fluid mechanics plays a fundamental role.

As well as the more traditional applications of aeronautics, hydraulics, heat and mass transfer etc., books will be published dealing with topics which are currently in a state of rapid development, such as turbulence, suspensions and multiphase fluids, super and hypersonic flows and numerical modeling techniques.

It is a widely held view that it is the interdisciplinary subjects that will receive intense scientific attention, bringing them to the forefront of technological advancement. Fluids have the ability to transport matter and its properties as well as to transmit force, therefore fluid mechanics is a subject that is particularly open to cross fertilization with other sciences and disciplines of engineering. The subject of fluid mechanics will be highly relevant in domains such as chemical, metallurgical, biological and ecological engineering. This series is particularly open to such new multidisciplinary domains.

The median level of presentation is the first year graduate student. Some texts are monographs defining the current state of a field; others are accessible to final year undergraduates; but essentially the emphasis is on readability and clarity.

More information about this series at http://www.springer.com/series/5980

F. Moukalled · L. Mangani
M. Darwish

The Finite Volume Method in Computational Fluid Dynamics

An Advanced Introduction with OpenFOAM® and Matlab®

Springer

F. Moukalled
Department of Mechanical Engineering
American University of Beirut
Beirut
Lebanon

M. Darwish
Department of Mechanical Engineering
American University of Beirut
Beirut
Lebanon

L. Mangani
Engineering and Architecture
Lucerne University of Applied Science and Arts
Horw
Switzerland

Additional material to this book can be downloaded from http://extras.springer.com.

ISSN 0926-5112 ISSN 2215-0056 (electronic)
Fluid Mechanics and Its Applications
ISBN 978-3-319-16873-9 ISBN 978-3-319-16874-6 (eBook)
DOI 10.1007/978-3-319-16874-6

Library of Congress Control Number: 2015939213

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

The impetus to write this book came about from three sources:

The first source was the bi-yearly computational fluid dynamics (CFD) course, which has been offered over the last 15 years at the American University of Beirut (AUB) by both Drs. Darwish and Moukalled to senior and graduate mechanical engineering students, a course that focuses on the finite volume method (FVM) and CFD applications.

The second source grew over the years to become more significant as it was noticed that graduates have started working on increasingly more focused areas and topics in CFD while becoming less cognizant of the general algorithmic expertise that earlier students developed. It became clear that there is a need not only to cover the basis of the numerics at the core of CFD codes but also to discuss the implementation issues to ensure that all students receive a robust understanding of the techniques they are working on.

Finally, the collaborative work in advanced numerics with Prof. Dr. Mangani from HSLU, Lucerne, Switzerland, which started during the Ph.D. supervision of M. Buchmyer (Ph.D.) from TUGraz, provided all the incentive to clarify and detail much of the numerical basis of the algorithms used in OpenFOAM®.

To this end, it was decided that the book would combine a mix of numerical and implementation details allowing the reader, if she/he desires, to fully understand and implement a robust and versatile CFD code based on the FVM.

This ambitious task was possible only by selecting from the various numerical methods in each of the topics covered in the book a handful set with which the authors are intimately familiar. The result is a book that covers intimately all the topics necessary for the development of a robust CFD code for the simulation of fluid flow at all speeds within the framework of the collocated unstructured finite volume method.

The book was also written with the classroom in mind as reflected by the use of copious illustrations; the provision of many exercises covering numerics, programming, and applications; the availability of an academic code (in MATLAB®) that imbeds much of the numerics presented in the book; and finally the various programs and routines in OpenFOAM®.

The hope is that as you read through this book, you will share with us the excitement and intense interest that we have grown to have for this subject.

Beirut F. Moukalled
Horw L. Mangani
Beirut M. Darwish
January 2015

Acknowledgments

It took nearly two years to complete this book, but much of what went in it was learned over a much longer period from interaction with numerous people in conferences and academic visits, from answering pertinent questions in our CFD courses and from our research work. However the enabler for all that is foremost the patience and kindness of our families.

We also wish to acknowledge the support provided to us by our respective institutions

American University of Beirut
Beirut, Lebanon

Lucerne University of Applied Science and Arts

Contents

Part IV Applications

About the Authors

Fadl Moukalled received his Ph.D. in Mechanical Engineering from Louisiana State University in 1987. During that same year, he joined the Mechanical Engineering Department at the American University of Beirut where currently he serves as a professor. His research interests cover several aspects of the finite volume method and its use in computational fluid dynamics. As a founding member of the CFD Group at AUB, he worked on convection schemes, pressure-based segregated algorithms for incompressible and compressible flows, adaptive grid methods, multigrid methods, transient schemes for free surface flows, multiphase flows, and fully coupled pressure-based solvers for incompressible, compressible, and multiphase flows.

Luca Mangani received his Ph.D. form the University of Florence in 2006, where he worked on the development of a state-of-the-art turbo machinery code in OpenFOAM® for heat transfer and combustion analysis. After three years of postdoc work, he joined the Lucerne University of Applied Sciences and Arts as senior research and chief engineer for CFD simulations. Since 2014, he is serving as an associate professor at the Fluid Mechanics and Hydro-machines Department, where he manages a variety of projects with industrial partners aimed at developing advanced and novel CFD tools. His research interests include pressure- and density-based solvers, segregated and fully coupled algorithms, fluid-structure interaction (FSI), turbulence, and conjugate heat transfer modeling.

Marwan Darwish received his Ph.D. in Materials Processing from BRUNEL University in 1991. He worked at the BICOM institute for one year as a postdoc before joining the Mechanical Engineering Department at the American University of Beirut in 1992, where he currently serves as a professor. His research interest covers a range of topics including solidification, advanced numerics, free surface flow, high-resolution schemes, multiphase flows, coupled algorithms, and algebraic multigrid. He is a founding member of the CFD Group at AUB.

Part I
Foundation

Chapter 1
Introduction

Abstract This chapter presents an overview of the book. It starts with a brief description of Computational Fluid Dynamics (CFD) and its use as a core design tool in a whole class of applications, and of the Finite Volume Method (FVM) and its role in the advancement of CFD. The chapter ends with a discussion of the book philosophy, structure, and content.

1.1 What Is Computational Fluid Dynamics (CFD)

"*We are literally at a significant point in history. A third branch of the scientific method, computer simulation, is emerging as a day-to-day tool. It is taking its place next to experimental development and mathematical theory as a way to new discoveries in science and engineering*". This was part of the speech of John Rollwagen, chairman and CEO of Cray Research, to the opening session of Supercomputing 89.

While it is common to refer to this or similar statements about the importance of simulation tools and techniques to the advancement of science and technology in general, it is now very clear that the use of simulation tools has become crucial to the development of a wide range of everyday technologies. In fact, numerical simulation tools nowadays play the role of technology enablers.

Computational Fluid Dynamics (CFD) is one such tool. Even though the impetus to its development was initially provided by some sections within the aeronautics and aerospace industry, it has grown to become an essential tool in a range of other design intensive industries such as the automotive, power generation, chemical, nuclear, and marine industries, to cite a few. Over the past decade newer industries have joined the ranks of heavy CFD users: for example in the electronics industry CFD is employed to optimize energy systems and heat transfer for the cooling of electronic devices, in the biomedical industry CFD is now a core development and validation tool for medical applications, and in the building industry CFD is used in

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_1

HVAC (heating, ventilating, and air conditioning), in fire simulation, and in air-quality assessment.

This has happened in little over two decades since the statement of John Rollwagen was made and over four decades since the development of the seminal SIMPLE algorithm by the CFD group of Brian Spalding at Imperial College in the early 70s.

Computational Fluid Dynamics is just one of the later Computer Aided Engineering tools that has gone mainstream. It has joined a well-established set of tools, such as the Finite Element Analysis (FEA) for Solid Mechanics and Vibration that has been part of the engineering design cycle since the mid 80s. The reason for this delay is the complexity of the equations that need to be solved. At their center is the Navier-Stokes equation that amazingly enough models accurately a whole set of flow phenomena from turbulent or laminar single phase incompressible flows, to compressible all-speed flows, and all the way to multiphase flows.

Amongst the numerical methods used to implement CFD, the Finite Volume Method has come to play a unique role.

1.2 What Is the Finite Volume Method

The Finite Volume Method (FVM) is a numerical technique that transforms the partial differential equations representing conservation laws over differential volumes into discrete algebraic equations over finite volumes (or elements or cells). In a similar fashion to the finite difference or finite element method, the first step in the solution process is the discretization of the geometric domain, which, in the FVM, is discretized into non-overlapping elements or finite volumes. The partial differential equations are then discretized/transformed into algebraic equations by integrating them over each discrete element. The system of algebraic equations is then solved to compute the values of the dependent variable for each of the elements.

In the finite volume method, some of the terms in the conservation equation are turned into face fluxes and evaluated at the finite volume faces. Because the flux entering a given volume is identical to that leaving the adjacent volume, the FVM is strictly conservative. This inherent conservation property of the FVM makes it the preferred method in CFD. Another important attribute of the FVM is that it can be formulated in the physical space on unstructured polygonal meshes. Finally in the FVM it is quite easy to implement a variety of boundary conditions in a non-invasive manner, since the unknown variables are evaluated at the centroids of the volume elements, not at their boundary faces.

These characteristics have made the Finite Volume Method quite suitable for the numerical simulation of a variety of applications involving fluid flow and heat and mass transfer, and developments in the method have been closely entwined with advances in CFD. From a limited potential at inception confined to solving simple physics and geometry over structured grids, the FVM is now capable of dealing with all kinds of complex physics and applications.

1.3 This Book

This book is about the Finite Volume Method and Computational Fluid Dynamics. It incorporates the basic know how of the method as acquired by the authors over almost three decades of work in the area. The terminology was carefully chosen, and vector notation was used whenever possible to ensure conciseness and consistency across all topics covered. Derivations are presented in a step by step fashion and illustrations are used extensively in the book to clarify concepts. In addition, a number of solved examples and exercises are also provided. Each chapter ends with a section denoted by "Computational Pointers" that provides pertinent details on implementation issues for two codes. The first, denoted by uFVM, is a Matlab®-based unstructured finite volume CFD educational code developed by the authors; while the second is OpenFOAM®, an open source finite volume code written in C++ capable of solving industrial type problems. Generally the numerics in each chapter are first presented for a one dimensional grid and progress towards two and three dimensional unstructured grids, to ease the introduction of difficult techniques.

The material presented allows the book to be utilized in a variety of ways. It can be used as a textbook for a senior undergraduate course covering the fundamentals of the finite volume discretization. It can also be deployed as a textbook for a graduate course on the application of the finite volume method and its use in computational fluid dynamics. It is also a handy reference book for workers in CFD, numerical heat transfer, and transport phenomena in general.

The content of the book falls into 20 chapters that may be grouped under the following four categories: (i) Foundation (Chaps. 2 through 7), (ii) Numerics (Chaps. 8 through 14), (iii) Algorithms (Chaps. 15 and 16), and (iv) Applications (17 through 19). Chapter 20 presents some closing remarks.

The uFVM Matlab® computer program, the OpenFOAM® developed routines, and the prepared lecture presentations can be downloaded from the book webpage at the following URL: "https://feaweb.aub.edu.lb/research/cfd"

A summary of the material covered in the forthcoming chapters is presented next.

1.3.1 Foundation

This part, covered in Chaps. 2 through 7, provides the necessary background for introducing the FVM.

Chapter 2 presents a short introduction of the elements of linear algebra including vectors, matrices, tensors, and their practices. This is in addition to an examination of the fundamental theorems of vector calculus.

Chapter 3 overviews the conservation principles governing fluid flow and related transport phenomena. It describes the derivations of the continuity, momentum, and

energy equations (collectively known as the Navier-Stokes equations). This is followed by the development of a typical conservation equation for a general scalar, vector, or tensor quantity. This equation forms the cornerstone for the developments presented in the numerics section. The conservation equation is shown to be composed of a transient, convection, diffusion, and source term. The discretization of each of these terms is presented in a separate chapter.

Chapter 4 summarizes the various steps of the discretization process, which include: (i) modeling the geometric domain and the physical phenomena, (ii) the discretization of the modeled geometric domain into a grid system, (iii) the discretization of the partial differential equation into an equivalent system of algebraic equations defined over each of the elements of the computational domain, and (iv) the solution of the system of equations.

Chapter 5 transforms the partial differential equation into a set of semi-discretized equations and presents a broad review of the numerical issues pertaining to the finite volume method. This provides a solid foundation for the chapters that follow.

Chapter 6 is devoted to the finite volume mesh. It starts with mesh discretization that replaces the geometric domain by a set of non overlapping elements. Then it proceeds with the computation of geometric information relevant to the various entities of the computational mesh in addition to the topological information that describes the arrangement and inter-relations of these entities.

Chapter 7 outlines the design decisions that shape the implementation of the two CFD codes, uFVM and OpenFOAM®. First the data structure and memory management schemes of the two codes are presented, then a sample test case is presented. Finally the format of the system of equations generated by each of the two codes are detailed.

1.3.2 Numerics

The material relevant to this part is covered in Chaps. 8 through 14. Each chapter specializes in the discretization of one of the terms in the general conservation equation derived in Chap. 2, with the exception of Chap. 10, which deals with linear solvers of algebraic systems of equations.

Chapter 8 describes the discretization of the diffusion term. The developments start on a structured Cartesian mesh and progress to unstructured non-orthogonal grid, while explaining the adopted treatment of the non-orthogonal cross-diffusion term. The chapter continues with a discussion on the used interpolation profiles and the rules that should be satisfied by the coefficients of the discretized system of algebraic equations. It also details the implementation of boundary conditions in addition to the under-relaxation procedure needed for highly non-linear problems.

Chapter 9 describes several techniques for evaluating gradients on a general mesh topology following either the Green-Gauss or the least square approach. It also presents methods to interpolate the gradient to element faces.

Chapter 10 deals with solvers of systems of algebraic equations. Both direct and iterative solvers are discussed with emphasis on iterative solvers because direct solvers are rarely used in CFD applications. The direct methods presented include the Gauss elimination and LU factorization. The concept of preconditioning is presented and the performance and limitations of some iterative methods are reviewed. This include the Jacobi, Gauss-Siedel, Incomplete LU factorization, and the conjugate gradient methods. The chapter also introduces the algebraic multigrid method, which is generally used in combination with iterative solvers to accelerate their convergence.

Chapter 11 proceeds with the discretization of the convection term assuming a known flow field. The shortcomings of using a symmetrical linear profile for the discretization of the convection term are delineated and a remedy is suggested through the use of an upwind profile. The high diffusion error associated with the upwind scheme is pointed out and upwind-biased higher order schemes are suggested.

Chapter 12 continues the developments of convection schemes and discusses approaches by which the dispersion error (unboundedness of the interpolation profile) affecting High Order (HO) schemes is resolved. This is achieved by enforcing a Convection Boundedness Criterion (CBC) on the HO profiles resulting in the group of High Resolution (HR) schemes. The Normalized Variable Formulation (NVF) and the Total Variation Diminishing (TVD) frameworks for constructing these HR schemes are presented. The commonality between the two approaches is explained through the Normalized Variable Diagram (NVD) and Sweby's diagram used in the NVF and TVD formulation, respectively. Many schemes are presented in the context of both formulations. Techniques for the implementation of HR schemes in structured and unstructured grids are reported.

Chapter 13 focusses on the discretization of the unsteady term that arises in the simulation of transient problems. Several transient schemes are developed following two different approaches. In the first, a finite difference approximation (via Taylor expansion) is used. In the second approach, the finite volume method is used on a temporal element in a similar fashion to what was done to the convection term.

Chapter 14 is devoted to a number of "small" numerical details that may have "big" effects on the convergence behavior. First the linearization of the source term when it is solution dependent is discussed. Then explicit and implicit techniques for under relaxing the algebraic equations are presented. The chapter ends with an examination of convergence indicators.

1.3.3 Algorithms

The previous chapters solved the general conservation equation assuming a given flow field. In general, the flow field is not known and has to be computed. This is the subject of Chaps. 15 and 16.

Chapter 15 is concerned with the prediction of incompressible flows. The difficulties associated with resolving the strong coupling between pressure and velocity, with the absence of an equation for pressure, are overcome by the SIMPLE algorithm with the derivation of a pressure correction equation. The Rhie-Chow interpolation is then introduced to allow realizing solutions to flow problems on collocated grids. Finally the implementation of a number of frequently encountered boundary conditions is detailed.

Chapter 16 extends the SIMPLE algorithm into compressible flows. The dependence of density on pressure and temperature is accounted for in the pressure correction equation through a density correction, giving rise to a convection-like term that transforms the mathematical nature of the equation from elliptic (for incompressible flows) to hyperbolic. Implementation details for a number of boundary conditions are also provided.

1.3.4 Applications

This part describes the implementation and application of the numerical techniques developed in the previous chapters.

Chapter 17 applies these numerical techniques to address some of the challenges faced when solving turbulent flow problems. It introduces several two-equation turbulence models and details the treatment of the near wall region.

Chapter 18 reviews the implementation of boundary conditions in OpenFOAM® and provides the needed information for adding new boundary conditions in the code. The no-slip wall boundary condition is described in some details.

Chapter 19 outlines the solution procedure of a reference test case in which solvers and boundary conditions are applied to solve a turbulent incompressible flow problem.

Finally, Chapter 20 presents some closing remarks.

1.4 Closure

The chapter discussed the growing role played by Computational Fluid Dynamics (CFD) as a core design tool in a whole class of applications and provided a general overview of the Finite Volume Method (FVM). It also clarified the purpose of the book, its intended use, and a summary of its content. The next chapter will give a brief review of the mathematical operations that will be used throughout the book.

Chapter 2
Review of Vector Calculus

Abstract This chapter sets the ground for the derivation of the conservation equations by providing a brief review of the continuum mechanics tools needed for that purpose while establishing some of the mathematical notations and procedures that will be used throughout the book. The review is by no mean comprehensive and assumes a basic knowledge of the fundamentals of continuum mechanics. A short introduction of the elements of linear algebra including vectors, matrices, tensors, and their practices is given. The chapter ends with an examination of the fundamental theorems of vector calculus, which constitute the elementary building blocks needed for manipulating and solving these conservation equations either analytically or numerically using computational fluid dynamics.

2.1 Introduction

The transfer phenomena of interest here can be mathematically represented by equations involving physical variables that fall under three categories: scalars, vectors, and tensors [1–3]. Throughout this book scalars are designated by *lightface* italic, vectors by lower boldface **Roman**, and tensors by boldface **Greek** letters. In addition, matrices are identified by upper boldface **Roman** letters.

A scalar represents a quantity that has magnitude such as volume V, pressure p, temperature T, time t, mass m, and density ρ. A vector represents a quantity of a given magnitude and direction such as velocity $\mathbf{v}$, momentum $\mathbf{L} = m\mathbf{v}$, and force $\mathbf{F}$. A matrix is a rectangular array of quantities ordered along rows and columns. A tensor is a mathematical object analogous to but more general than a vector, represented by an array of components, such as the shear stress tensor. Moreover, the conservation equations are composed of terms that represent the product of two or more variables. The multiplication involved may be of various types to be detailed later and the variables could be a combination of the three types described above. Whenever the multiplication results in a scalar, the product will be enclosed

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_2

by parentheses "(product)", if it results in a vector it will be enclosed by square brackets "[product]", and if it results in a tensor it will be enclosed by curly brackets "{product}".

2.2 Vectors and Vector Operations

The most frequently used vector in fluid dynamics is the velocity vector that will be designated by $\mathbf{v}$. The components of the velocity vector in a three-dimensional Cartesian coordinate system will be denoted by u, v, and w in the x, y, and z direction, respectively (Fig. 2.1). In Cartesian coordinates, $\mathbf{v}$ is written as

$$\mathbf{v} = u\mathbf{i} + v\mathbf{j} + w\mathbf{k} \tag{2.1}$$

where $\mathbf{i}, \mathbf{j}$, and $\mathbf{k}$ are unit vectors in the x, y, and z direction, respectively. A vector is usually presented in a column format with its transpose, denoted with a superscript T, in a row format as

$$\mathbf{v} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad \mathbf{v}^{\mathrm{T}} = \begin{bmatrix} u & v & w \end{bmatrix} \tag{2.2}$$

The magnitude of a vector is given by

$$\|\mathbf{v}\| = \sqrt{u^2 + v^2 + w^2} \tag{2.3}$$

The sum of two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is the sum of their components, i.e.,

$$\left.\begin{aligned} \mathbf{v}_1 &= u_1\mathbf{i} + v_1\mathbf{j} + w_1\mathbf{k} \\ \mathbf{v}_2 &= u_2\mathbf{i} + v_2\mathbf{j} + w_2\mathbf{k} \end{aligned}\right\} \Rightarrow \mathbf{v}_1 + \mathbf{v}_2 = (u_1 + u_2)\mathbf{i} + (v_1 + v_2)\mathbf{j} + (w_1 + w_2)\mathbf{k} \tag{2.4}$$

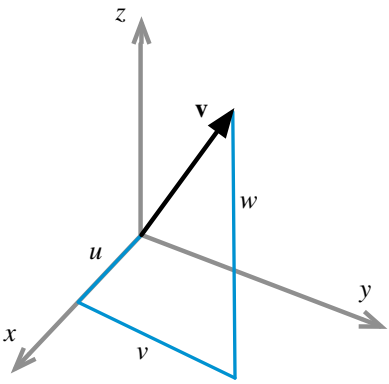


Fig. 2.1 The components of a vector $\mathbf{v}$ in a three-dimensional Cartesian coordinate system

or

$$\mathbf{v}_1 = \begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix} \quad \mathbf{v}_2 = \begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} \Rightarrow \mathbf{v}_1 + \mathbf{v}_2 = \begin{bmatrix} u_1 + u_2 \\ v_1 + v_2 \\ w_1 + w_2 \end{bmatrix} \tag{2.5}$$

The multiplication of a vector $\mathbf{v}$ by a scalar s results in the vector $s\mathbf{v}$ such that

$$\begin{aligned} s\mathbf{v} &= s(u\mathbf{i} + v\mathbf{j} + w\mathbf{k}) \\ &= su\mathbf{i} + sv\mathbf{j} + sw\mathbf{k} = \begin{bmatrix} su \\ sv \\ sw \end{bmatrix} \end{aligned} \tag{2.6}$$

The product of two vectors is not as straightforward. When multiplying a vector $\mathbf{v}_1$ by another vector $\mathbf{v}_2$ two types of multiplications arise [4–6]. The first is denoted by the scalar or dot product, $(\mathbf{v}_1 \cdot \mathbf{v}_2)$, and the second by vector or cross product $[\mathbf{v}_1 \times \mathbf{v}_2]$.

2.2.1 The Dot Product of Two Vectors

By definition, the dot product of two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is a scalar quantity given by

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\|\|\mathbf{v}_2\|\cos(\mathbf{v}_1, \mathbf{v}_2) \tag{2.7}$$

where $\cos(\mathbf{v}_1, \mathbf{v}_2)$ denotes the cosine of the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$. From the definition of the vector dot product, it follows that

$$\begin{aligned} &\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = \mathbf{k} \cdot \mathbf{k} = 1 \\ &\mathbf{i} \cdot \mathbf{j} = \mathbf{i} \cdot \mathbf{k} = \mathbf{j} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{k} = \mathbf{k} \cdot \mathbf{i} = \mathbf{k} \cdot \mathbf{j} = 0 \end{aligned} \tag{2.8}$$

In terms of orthonormal Cartesian components, the dot product of the two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ can be calculated as

$$\begin{aligned} \mathbf{v}_1 \cdot \mathbf{v}_2 &= (u_1\mathbf{i} + v_1\mathbf{j} + w_1\mathbf{k}) \cdot (u_2\mathbf{i} + v_2\mathbf{j} + w_2\mathbf{k}) \\ &= u_1u_2 + v_1v_2 + w_1w_2 \end{aligned} \tag{2.9}$$

2.2.2 Vector Magnitude

From Eq. (2.9) it follows that the magnitude of a vector $\mathbf{v}$ can be obtained as

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{u^2 + v^2 + w^2} \tag{2.10}$$

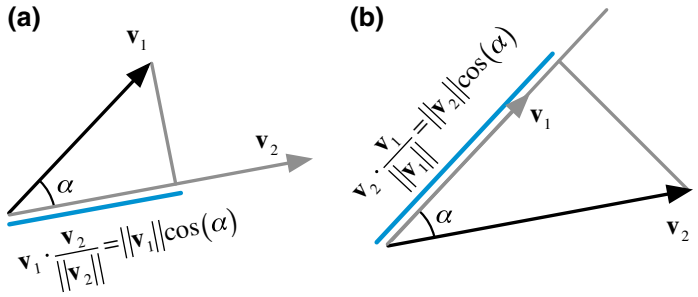


Fig. 2.2 **a** Projection of vector $\mathbf{v}_1$ onto the unit direction of vector $\mathbf{v}_2$; **b** Projection of vector $\mathbf{v}_2$ onto the unit direction of vector $\mathbf{v}_1$

2.2.3 The Unit Direction Vector

A unit vector $\mathbf{e}_v$ in the direction of $\mathbf{v}$ can be derived from the definition of the dot product as

$$\left.\begin{aligned} \mathbf{v}\cdot\mathbf{v} &= \|\mathbf{v}\|\|\mathbf{v}\|\overbrace{\cos(\mathbf{v},\mathbf{v})}^{=1} = \|\mathbf{v}\|^2 \Rightarrow \mathbf{v}\cdot\frac{\mathbf{v}}{\|\mathbf{v}\|} = \|\mathbf{v}\| \\ \mathbf{v}\cdot\mathbf{e}_v &= \|\mathbf{v}\|\underbrace{\|\mathbf{e}_v\|}_{=1}\underbrace{\cos(\mathbf{v},\mathbf{e}_v)}_{=1} = \|\mathbf{v}\| \Rightarrow \mathbf{v}\cdot\mathbf{e}_v = \|\mathbf{v}\| \end{aligned}\right\} \Rightarrow \mathbf{e}_v = \frac{\mathbf{v}}{\|\mathbf{v}\|} \tag{2.11}$$

Therefore the component of a vector in the direction of another vector (i.e., magnitude of the projected length) can be viewed as the dot product of the vector to be projected with the unit direction of the other vector as shown in Fig. 2.2a, b.

2.2.4 The Cross Product of Two Vectors

Whereas the dot product of two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is a scalar quantity, their cross or vector product is a vector $\mathbf{v}_3$ normal to the plane formed by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, of magnitude calculated as

$$\|\mathbf{v}_3\| = \|\mathbf{v}_1 \times \mathbf{v}_2\| = \|\mathbf{v}_1\|\|\mathbf{v}_2\||\sin(\mathbf{v}_1,\mathbf{v}_2)|, \tag{2.12}$$

and of direction given by the right hand rule. As shown in Fig. 2.3, the magnitude of the cross product of two vectors represents the area of the parallelogram spanned by the two vectors. Since, in addition, the resulting vector is normal to the plane

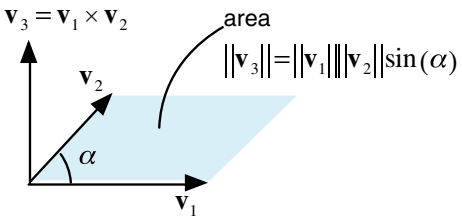


Fig. 2.3 The cross product of two vectors

formed by the vectors, the cross product of two vectors represents their surface vector.

It is then clear that the cross product of two collinear vectors is zero as they define no area, and that the cross product of two orthogonal unit vectors is a unit vector perpendicular to the two unit vectors. Adopting the right hand rule to define the direction of the resulting vector, the following cross product operations hold:

$$\begin{aligned} \mathbf{i} \times \mathbf{i} = \mathbf{j} \times \mathbf{j} = \mathbf{k} \times \mathbf{k} = 0 \quad & \mathbf{i} \times \mathbf{j} = \mathbf{k} = -\mathbf{j} \times \mathbf{i} \\ \mathbf{j} \times \mathbf{k} = \mathbf{i} = -\mathbf{k} \times \mathbf{j} \quad & \mathbf{k} \times \mathbf{i} = \mathbf{j} = -\mathbf{i} \times \mathbf{k} \end{aligned} \tag{2.13}$$

Using the above relations, the cross product of two vectors in terms of their Cartesian components is given by

$$\begin{aligned} \mathbf{v}_1 \times \mathbf{v}_2 &= (u_1\mathbf{i} + v_1\mathbf{j} + w_1\mathbf{k}) \times (u_2\mathbf{i} + v_2\mathbf{j} + w_2\mathbf{k}) \\ &= u_1u_2\mathbf{i} \times \mathbf{i} + u_1v_2\mathbf{i} \times \mathbf{j} + u_1w_2\mathbf{i} \times \mathbf{k} \\ &\quad + v_1u_2\mathbf{j} \times \mathbf{i} + v_1v_2\mathbf{j} \times \mathbf{j} + v_1w_2\mathbf{j} \times \mathbf{k} \\ &\quad + w_1u_2\mathbf{k} \times \mathbf{i} + w_1v_2\mathbf{k} \times \mathbf{j} + w_1w_2\mathbf{k} \times \mathbf{k} \\ &= u_1u_20 + u_1v_2\mathbf{k} + u_1w_2(-\mathbf{j}) \\ &\quad + v_1u_2(-\mathbf{k}) + v_1v_20 + v_1w_2\mathbf{i} \\ &\quad + w_1u_2\mathbf{j} + w_1v_2(-\mathbf{i}) + w_1w_20 \\ &= (v_1w_2 - v_2w_1)\mathbf{i} - (u_1w_2 - u_2w_1)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k} \end{aligned} \tag{2.14}$$

which can be written using determinant notation as

$$\mathbf{v}_1 \times \mathbf{v}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \end{vmatrix} = \begin{bmatrix} v_1w_2 - v_2w_1 \\ u_2w_1 - u_1w_2 \\ u_1v_2 - u_2v_1 \end{bmatrix} \tag{2.15}$$

Example 1
Compute the area of the triangle formed by points (Fig. 2.4): $P_1(0, 0, 0), P_2(1, 0, 0)$ *and* $P_3(0.5, 1, 0)$.

Solution
The surface defined by the triangle (P_1, P_2, P_3) can be computed using the cross product of two sides as

$$\mathbf{S}_{123} = 0.5\ \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}$$

$$\overrightarrow{P_1P_2} = (x_2 - x_1)\mathbf{i} + (y_2 - y_1)\mathbf{j} + (z_2 - z_1)\mathbf{k} = \mathbf{i}$$
$$\overrightarrow{P_1P_3} = (x_3 - x_1)\mathbf{i} + (y_3 - y_1)\mathbf{j} + (z_3 - z_1)\mathbf{k} = 0.5\mathbf{i} + \mathbf{j}$$
$$\mathbf{S}_{123} = 0.5\mathbf{i} \times (0.5\mathbf{i} + \mathbf{j}) = 0.5\mathbf{k} \Rightarrow \|\mathbf{S}_{123}\| = 0.5$$

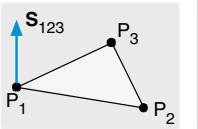


Fig. 2.4 Example 1

2.2.5 The Scalar Triple Product

In addition, combined products of three vectors $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ may arise such as $(\mathbf{v}_1 \cdot [\mathbf{v}_2 \times \mathbf{v}_3])$, which can be calculated using the following determinant (to be explained later):

$$(\mathbf{v}_1 \cdot [\mathbf{v}_2 \times \mathbf{v}_3]) = \begin{vmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{vmatrix} \tag{2.16}$$

As shown in Fig. 2.5, the absolute value of the scalar triple product represents the volume of the parallelepiped formed by the vectors $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$.

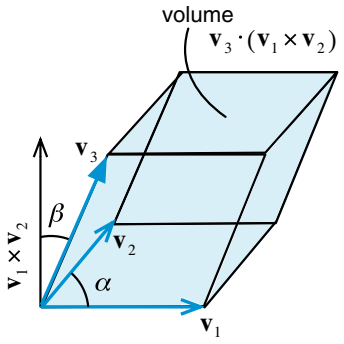


Fig. 2.5 Geometric representation of scalar triple product

Example 2
Compute the volume of the pyramid defined by the points: $P_1(0, 0, 0), P_2(1, 0, 0), P_3(0.5, 1, 0)$, *and* $P_4(0.5, 0.5, 1)$ *shown in* Fig. 2.6.

Solution
The volume of the pyramid can be computed using the scalar triple product as

$$\begin{aligned} V &= 0.25\,\overrightarrow{P_1P_4}\cdot\left(\overrightarrow{P_1P_2}\times\overrightarrow{P_1P_3}\right) \\ &= 0.25(0.5\mathbf{i}+0.5\mathbf{j}+\mathbf{k})\cdot\mathbf{k} \\ &= 0.25 \end{aligned}$$

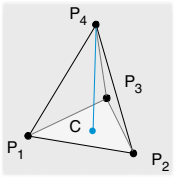


Fig. 2.6 Example 2

2.2.6 Gradient of a Scalar and Directional Derivatives

An important vector operator, which arises frequently in fluid dynamics, is the "del" (or "nabla") operator defined as

$$\nabla = \frac{\partial}{\partial x}\mathbf{i} + \frac{\partial}{\partial y}\mathbf{j} + \frac{\partial}{\partial z}\mathbf{k} \tag{2.17}$$

When the "del" operator is applied on a scalar variable s it results in the gradient of s [7, 8] given by

$$\nabla s = \frac{\partial s}{\partial x}\mathbf{i} + \frac{\partial s}{\partial y}\mathbf{j} + \frac{\partial s}{\partial z}\mathbf{k} \tag{2.18}$$

Thus the gradient of a scalar field is a vector field indicating that the value of s changes with position in both magnitude and direction.

The projection of ∇s in a certain direction of unit vector $\mathbf{e}_l$ is given by

$$\frac{ds}{dl} = \nabla s \cdot \mathbf{e}_l = \|\nabla s\| \cos(\nabla s, \mathbf{e}_l) \tag{2.19}$$

and is called the directional derivative of s along the direction of the unit vector $\mathbf{e}_l$, as schematically depicted in Fig. 2.7. The maximum value of the directional derivative is $\|\nabla s\|$ and is obtained when $\cos(\nabla s, \mathbf{e}_l) = 1$, that is in the direction of ∇s. Therefore, it can be stated that the gradient of a scalar field s indicates the direction and magnitude of the largest change in s at every point in space. Moreover, ∇s is normal to the constant s surface that passes through that point.

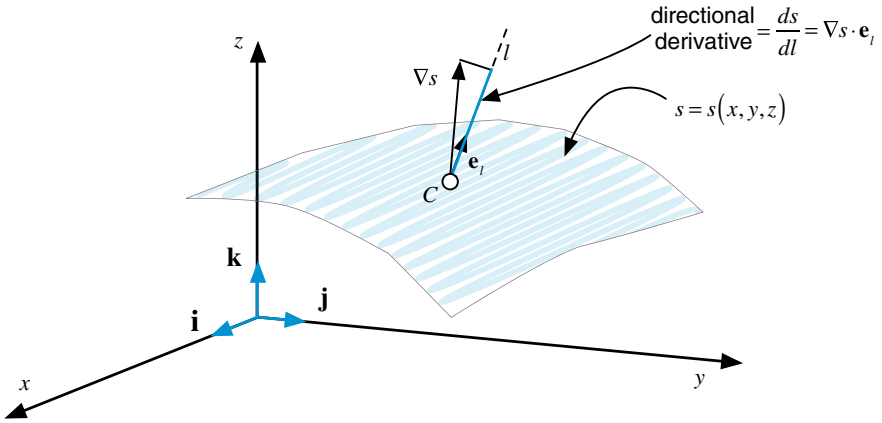


Fig. 2.7 The rate of change of $s(x, y, z)$ in the direction of vector $\mathbf{e}_l$

Example 3

Let $f(x, y, z) = x^2y + y^2z + z^2x$

(a) *find* ∇f *at point* $(3, 2, 0)$.

(b) *find the derivative at point* $(3, 2, 0)$ *along the direction* $(1, 2, 2)$.

Solution

(a) $\dfrac{\partial f}{\partial x} = 2xy + z^2 \quad \dfrac{\partial f}{\partial y} = x^2 + 2yz \quad \dfrac{\partial f}{\partial z} = y^2 + 2xz$

$$\nabla f = (2xy + z^2)\mathbf{i} + (x^2 + 2yz)\mathbf{j} + (y^2 + 2xz)\mathbf{k}$$

Thus

$$\nabla f|_{(3,2,0)} = 12\mathbf{i} + 9\mathbf{j} + 4\mathbf{k}$$

(b) The unit vector along direction $(1, 2, 2)$ is

$$\mathbf{e}_l = \frac{1\mathbf{i} + 2\mathbf{j} + 2\mathbf{k}}{\sqrt{1^2 + 2^2 + 2^2}} = \frac{1\mathbf{i} + 2\mathbf{j} + 2\mathbf{k}}{3}$$

The derivative along the direction $(1, 2, 2)$ is

$$\left.\frac{df}{dl}\right|_{(3,2,0)} = \nabla f|_{(3,2,0)} \cdot \mathbf{e}_l$$
$$= (12\mathbf{i} + 9\mathbf{j} + 4\mathbf{k}) \cdot \frac{1\mathbf{i} + 2\mathbf{j} + 2\mathbf{k}}{3}$$
$$= (12 + 18 + 8)/3 = 38/3$$

2.2.7 Operations on the Nabla Operator

The dot product of the del operator with a vector $\mathbf{v}$ of components u, v, and w in the x, y, and z direction, respectively, results in the divergence of the vector [7, 8], which is a scalar quantity written as

$$\nabla \cdot \mathbf{v} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{2.20}$$

Physically the divergence of a vector field over a region is a measure of how much the vector field points into or out of the region.

The divergence of the gradient of a scalar variable s is denoted by the Laplacian of s and is a scalar given by

$$\nabla \cdot (\nabla s) = \nabla^2 s = \frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} + \frac{\partial^2 s}{\partial z^2} \tag{2.21}$$

The Laplacian of a vector follows from the above definition of the Laplacian operator and is a vector computed as

$$\nabla^2 \mathbf{v} = \left(\nabla^2 u\right)\mathbf{i} + \left(\nabla^2 v\right)\mathbf{j} + \left(\nabla^2 w\right)\mathbf{k} \tag{2.22}$$

Example 4
Find the divergence of $\mathbf{v} = (u, v, w) = (3x, 2xy, 4z)$

Solution
Then divergence of $\mathbf{v}$ is obtained as

$$\nabla \cdot \mathbf{v} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$
$$= 3 + 2x + 4$$
$$= 7 + 2x$$

Another quantity of interest is the curl of a vector field [7, 8] formed between the "del" operator and the vector **v**, resulting in the following vector:

$$\begin{aligned}\nabla \times \mathbf{v} &= \left(\frac{\partial}{\partial x}\mathbf{i} + \frac{\partial}{\partial y}\mathbf{j} + \frac{\partial}{\partial z}\mathbf{k}\right) \times (u\mathbf{i} + v\mathbf{j} + w\mathbf{k}) \\ &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \dfrac{\partial}{\partial x} & \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial z} \\ u & v & w \end{vmatrix} = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}\right)\mathbf{i} + \left(\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}\right)\mathbf{j} + \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)\mathbf{k}\end{aligned} \tag{2.23}$$

Examples of the divergence and curl of a vector field are schematically displayed in Fig. 2.8. The radial vector field shown in Fig. 2.8a has only divergence with zero curl. In fluid mechanics this vector field represents the velocity field of a sink/source flow. On the other hand Fig. 2.8b depicts a rotational vector field which has only curl with zero divergence (i.e., a divergence free vector field). Such a field corresponds to the velocity field of a vortex flow.

The divergence of a vector **v** with its gradient also arises in the equations of interest in this book and is computed as

$$\begin{aligned}[(\mathbf{v}.\nabla)\mathbf{v}] &= (u\mathbf{i} + v\mathbf{j} + w\mathbf{k}) \cdot \left(\frac{\partial}{\partial x}\mathbf{i} + \frac{\partial}{\partial y}\mathbf{j} + \frac{\partial}{\partial z}\mathbf{k}\right)(u\mathbf{i} + v\mathbf{j} + w\mathbf{k}) \\ &= \left(u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z}\right)(u\mathbf{i} + v\mathbf{j} + w\mathbf{k}) \\ &= \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\right)\mathbf{i} + \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}\right)\mathbf{j} + \left(u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}\right)\mathbf{k}\end{aligned} \tag{2.24}$$

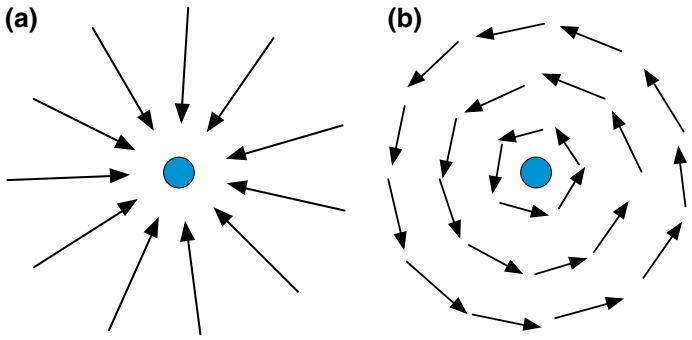


Fig. 2.8 **a** A radial vector field, **b** a solenoidal vector field

Example 5

Determine for the flow fields shown in Fig. 2.9a, b, c *which is divergence free (i.e., neither expanding nor compressing) and which is irrotational (i.e., does not undergo a rotation)*

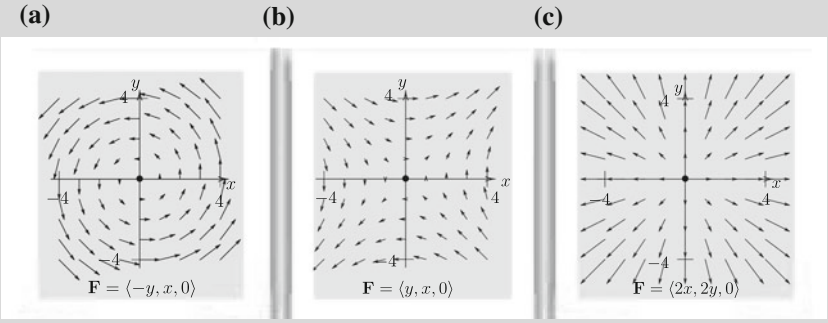


Fig. 2.9 Example 5

a $\nabla \cdot \mathbf{F} = 0$ $\quad \nabla \times \mathbf{F} = 0\mathbf{i} + 0\mathbf{j} + 2\mathbf{k}$
b $\nabla \cdot \mathbf{F} = 0$ $\quad \nabla \times \mathbf{F} = 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$
c $\nabla \cdot \mathbf{F} = 2 + 2 = 4$ $\quad \nabla \times \mathbf{F} = 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$

2.2.8 Additional Vector Operations

If s is a scalar function, and $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are vector fields, then the following relations, which are listed without proof, apply:

$$\nabla \cdot (\nabla \times \mathbf{v}) = 0 \tag{2.25}$$

$$\nabla \times (\nabla s) = 0 \tag{2.26}$$

$$\nabla \cdot (s\mathbf{v}) = s\nabla \cdot \mathbf{v} + \mathbf{v} \cdot \nabla s \tag{2.27}$$

$$\nabla \times (s\mathbf{v}) = s\nabla \times \mathbf{v} + \nabla s \times \mathbf{v} \tag{2.28}$$

$$\nabla(\mathbf{v}_1 \cdot \mathbf{v}_2) = \mathbf{v}_1 \times (\nabla \times \mathbf{v}_2) + \mathbf{v}_2 \times (\nabla \times \mathbf{v}_1) + (\mathbf{v}_1 \cdot \nabla)\mathbf{v}_2 + (\mathbf{v}_2 \cdot \nabla)\mathbf{v}_1 \tag{2.29}$$

$$\nabla \cdot (\mathbf{v}_1 \times \mathbf{v}_2) = \mathbf{v}_2 \cdot (\nabla \times \mathbf{v}_1) - \mathbf{v}_1 \cdot (\nabla \times \mathbf{v}_2) \tag{2.30}$$

$$\nabla \times (\mathbf{v}_1 \times \mathbf{v}_2) = \mathbf{v}_1(\nabla \cdot \mathbf{v}_2) - \mathbf{v}_2(\nabla \cdot \mathbf{v}_1) + (\mathbf{v}_2 \cdot \nabla)\mathbf{v}_1 - (\mathbf{v}_1 \cdot \nabla)\mathbf{v}_2 \tag{2.31}$$

$$\nabla \times (\nabla \times \mathbf{v}) = \nabla(\nabla \cdot \mathbf{v}) - \nabla^2\mathbf{v} \tag{2.32}$$

$$(\nabla \times \mathbf{v}) \times \mathbf{v} = \mathbf{v} \cdot (\nabla\mathbf{v}) - \nabla(\mathbf{v} \cdot \mathbf{v}) \tag{2.33}$$

2.3 Matrices and Matrix Operations

A matrix $\mathbf{A}$ of order $M \times N$ is a rectangular array of quantities (numbers or expressions) arranged in M rows and N columns [9–11]. An element of $\mathbf{A}$ located on the ith row and jth column is denoted by a_{ij}. For example, element a_{32} of the 4×3 matrix shown in Fig. 2.10 is 12.

Based on this definition it follows that a column vector $\mathbf{v}$ of dimensionality N is a matrix of order $N \times 1$ and a scalar s is a matrix of order 1×1.

The transpose of a matrix $\mathbf{A}$ of order $M \times N$ is another matrix denoted by $\mathbf{A}^{\mathrm{T}}$ of order $N \times M$ for which the rows of $\mathbf{A}$ are the columns of $\mathbf{A}^{\mathrm{T}}$ and the columns of $\mathbf{A}$ are the rows of $\mathbf{A}^{\mathrm{T}}$. Mathematically, this can be written as

$$\mathbf{A} = \left[a_{ij}\right] \Rightarrow \mathbf{A}^{\mathrm{T}} = \left[a_{ji}\right] \tag{2.34}$$

Two matrices of the same order are equal if their corresponding elements are equal. Two matrices of the same order can be added or subtracted element by element. For example, if $\mathbf{A}$ and $\mathbf{B}$ are given by

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & -1 & 7 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -2 & 1 & 4 \\ -3 & 1 & 6 \end{bmatrix}$$

then $\mathbf{A} + \mathbf{B}$ and $\mathbf{A} - \mathbf{B}$ are found to be

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} -1 & 3 & 8 \\ 0 & 0 & 13 \end{bmatrix} \quad \mathbf{A} - \mathbf{B} = \begin{bmatrix} 3 & 1 & 0 \\ 6 & -2 & 1 \end{bmatrix}$$

If a matrix is multiplied by a scalar s than all its elements are multiplied by s. Mathematically this is written as

$$\mathbf{A} = \left[a_{ij}\right] \Rightarrow s\mathbf{A} = \left[sa_{ij}\right] \tag{2.35}$$

To multiply two matrices $\mathbf{A}$ and $\mathbf{B}$, the number of columns of $\mathbf{A}$ must be equal to the number of rows of $\mathbf{B}$. Therefore, if $\mathbf{A}$ is of size $M \times X$ for the product $\mathbf{P} = \mathbf{AB}$

$$\begin{array}{cc} i & j \rightarrow \\ \downarrow & \\ 1 & \\ 2 & \\ 3 & \\ 4 & \end{array} \begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -4 \\ 5 & 4 & 7 \\ 0 & 12 & -2 \\ 3 & 6 & 3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} = \left[a_{ij}\right]$$

Fig. 2.10 Example of a 4×3 matrix

to be possible, **B** must be of size $X \times N$. The size of **P** will be $M \times N$ with its element p_{ij} obtained as

$$p_{ij} = \sum_{k=1}^{X} a_{ik} b_{kj} \tag{2.36}$$

If **A** is a 3×2 matrix and **B** a 2×4 matrix given by

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \\ 2 & -5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 2 & -1 & 0 & 4 \\ -3 & 0 & 3 & 2 \end{bmatrix}$$

then $\mathbf{P} = \mathbf{AB}$ will be a 3×4 matrix computed as

$$\mathbf{P} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \\ 2 & -5 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 4 \\ -3 & 0 & 3 & 2 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix}$$

$$\left.\begin{array}{l} p_{11} = 1 * 2 + 2 * (-3) = -4 \\ p_{12} = 1 * (-1) + 2 * 0 = -1 \\ p_{13} = \ldots \\ \vdots \end{array}\right\} \Rightarrow \mathbf{P} = \begin{bmatrix} -3 & -1 & 6 & 8 \\ -11 & 1 & 9 & 2 \\ 19 & -2 & -15 & -2 \end{bmatrix}$$

2.3.1 Square Matrices

If the number of columns N of matrix **A** is equal to its number of rows, then **A** is a square matrix of order N. The elements a_{ii} of a square matrix **A** form its main diagonal which stretches from top left to bottom right. The diagonal composed of elements a_{ij} for which $i + j = N + 1$ is called the cross diagonal and it extends from the bottom left to top right.

Square matrices possess properties that are not applicable to other types of matrices such as symmetry and antisymmetry. In addition, many operations such as taking determinants and calculating eigenvalues are only defined for square matrices.

The result of multiplying a square matrix of order N by itself is a square matrix of order N. Therefore a square matrix can be multiplied by itself as many times as needed and the notation $\mathbf{A}^k$ designates **A** multiplied by itself k times, i.e.,

$$\mathbf{A}^k = \underbrace{\mathbf{A} \times \mathbf{A} \times \mathbf{A} \ldots \times \mathbf{A}}_{k \text{ times}} \tag{2.37}$$

A square matrix $\mathbf{A}$ is symmetric if $a_{ij} = a_{ji}\left(\text{i.e.}, \mathbf{A}^{\mathrm{T}} = \mathbf{A}\right)$, and antisymmetric if $a_{ij} = -a_{ji}$. An example of a symmetric square matrix of order 3 is

$$\begin{bmatrix} 5 & 3 & -2 \\ 3 & 2 & 7 \\ -2 & 7 & -1 \end{bmatrix}$$

and of an antisymmetric square matrix of order 4 is

$$\begin{bmatrix} 0 & 3 & -2 & 4 \\ -3 & 0 & 1 & -3 \\ 2 & -1 & 0 & -2 \\ -4 & 3 & 2 & 0 \end{bmatrix}$$

A diagonal square matrix $\mathbf{D}$ is one for which all elements off the main diagonal are zero while elements on the main diagonal are arbitrary. An example of a square diagonal matrix of order 3 is

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

A diagonal matrix of order N for which all elements on the main diagonal are 1 (i.e., $a_{ii} = 1$) is called an identity matrix of order N and is designated by $\mathbf{I}$. An identity matrix of order 4 is given by

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The inverse of a square matrix $\mathbf{A}$ of order N is the square matrix $\mathbf{A}^{-1}$ of order N satisfying

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I} \tag{2.38}$$

An upper triangular matrix $\mathbf{U}$ is a square matrix in which all elements below the main diagonal are zero. Mathematically this can be expressed as

$$\mathbf{U} = \begin{cases} u_{ij} & i \leq j \\ 0 & i > j \end{cases} \tag{2.39}$$

A lower triangular matrix $\mathbf{L}$ is a square matrix in which all elements above the main diagonal are zero. Using mathematical notation, this is written as

$$\mathbf{L} = \begin{cases} \ell_{ij} & i \geq j \\ 0 & i < j \end{cases} \tag{2.40}$$

Examples of upper and lower triangular square matrices of order 3 are

$$\mathbf{U} = \begin{bmatrix} 1 & 2 & 6 \\ 0 & 4 & 5 \\ 0 & 0 & -7 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 3 & 0 & 0 \\ -1 & 2 & 0 \\ -9 & -2 & 4 \end{bmatrix}$$

2.3.2 Using Matrices to Describe Systems of Equations

Matrices can be used to compactly describe systems of equations [12]. A system of N equations in N unknowns can be written as

$$\begin{array}{l} a_{11}\phi_1 + a_{12}\phi_2 + a_{13}\phi_3 + \ldots + a_{1N}\phi_N = b_1 \\ a_{21}\phi_1 + a_{22}\phi_2 + a_{23}\phi_3 + \ldots + a_{2N}\phi_N = b_2 \\ a_{31}\phi_1 + a_{32}\phi_2 + a_{33}\phi_3 + \ldots + a_{3N}\phi_N = b_3 \\ \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{N1}\phi_1 + a_{N2}\phi_2 + a_{N3}\phi_3 + \ldots + a_{NN}\phi_N = b_N \end{array} \tag{2.41}$$

In matrix notation, this system of equations is equivalent to

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} \tag{2.42}$$

or in compact form as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \tag{2.43}$$

2.3.3 The Determinant of a Square Matrix

A determinant is a value associated with a square matrix $\mathbf{A}$ that can be computed from the elements of the matrix through a mathematical procedure and is denoted by $\det(\mathbf{A})$ or $|\mathbf{A}|$ (which should not be confused with the absolute value notation) [13].

The calculation of the determinant of a matrix of order 2 is straightforward and is the product of the elements in the main diagonal minus the product of the elements in the cross diagonal. If $\mathbf{A}$ is a square matrix of order 2 then,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \Rightarrow \det(\mathbf{A}) = a_{11}a_{22} - a_{21}a_{12} \tag{2.44}$$

For higher order matrices the procedure is more involved and is based on the notion of minors and cofactors.

A minor $(mi)_{ij}$ for an element a_{ij} is the determinant that results when the ith row and jth column are deleted. The cofactor $(co)_{ij}$ of an element a_{ij} is the value of the minor multiplied by either a positive or a negative sign depending on whether $(i+j)$ is even or odd, respectively. The mathematical relation between cofactors and minors can be written as

$$(co)_{ij} = (-1)^{i+j}(mi)_{ij} \tag{2.45}$$

The determinant of a square matrix $\mathbf{A}$ of order N is computed by finding the cofactors of one of its rows or its columns, multiplying each cofactor by the corresponding element, and adding the results. Mathematically this is given by

$$\det(\mathbf{A}) = \begin{cases} \sum_{i=1}^{N} a_{ij}(co)_{ij} & \textit{for any } j \\ \textit{or} & \\ \sum_{j=1}^{N} a_{ij}(co)_{ij} & \textit{for any } i \end{cases} \tag{2.46}$$

It should be clarified that the calculation of the cofactors may require further decomposition of the minor determinants. This decomposition may give rise to further decompositions until a determinant with a size of 2 is reached. Moreover, based on the above discussion it is easily demonstrated that the determinant of an upper, a lower, or a diagonal matrix $\mathbf{A}$ of order N is the product of the elements along its main diagonal, i.e., $\det(\mathbf{A}) = \prod_{i=1}^{N} a_{ii}$.

Example 6

Calculate the determinant of matrix $\mathbf{A}$ *of order 4 given by*

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 2 & 0 & 5 \\ 2 & 3 & -2 & 0 \\ 4 & 1 & -5 & 3 \end{bmatrix}$$

Solution

As mentioned above, the determinant can be calculated based on the cofactors of any selected row or column. A smart choice would be a row or a column with the largest number of zeros. Therefore computations will be reduced by selecting either the first row or the last column. The determinant will be calculated using both to further show that the end results will be the same.

The signs of cofactors are

$$\begin{bmatrix} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{bmatrix}$$

The determinant using cofactors of row 1 is computed as

$$\det(\mathbf{A}) = 1 * (co)_{11} + 1 * (co)_{13} = \begin{vmatrix} 2 & 0 & 5 \\ 3 & -2 & 0 \\ 1 & -5 & 3 \end{vmatrix} + \begin{vmatrix} 1 & 2 & 5 \\ 2 & 3 & 0 \\ 4 & 1 & 3 \end{vmatrix}$$

The first new determinant is calculated using the cofactors of row 1 while the second determinant is calculated using cofactors of column 3 as

$$\begin{aligned} \det(\mathbf{A}) &= 2\begin{vmatrix} -2 & 0 \\ -5 & 3 \end{vmatrix} + 5\begin{vmatrix} 3 & -2 \\ 1 & -5 \end{vmatrix} + 5\begin{vmatrix} 2 & 3 \\ 4 & 1 \end{vmatrix} + 3\begin{vmatrix} 1 & 2 \\ 2 & 3 \end{vmatrix} \\ &= 2(-6-0) + 5(-15+2) + 5(2-12) + 3(3-4) \\ &= -12 - 65 - 50 - 3 \\ \det(\mathbf{A}) &= -130 \end{aligned}$$

The determinant using cofactors of column 4 is calculated as

$$\det(\mathbf{A}) = 5 * (co)_{24} + 3 * (co)_{44} = 5\begin{vmatrix} 1 & 0 & 1 \\ 2 & 3 & -2 \\ 4 & 1 & -5 \end{vmatrix} + 3\begin{vmatrix} 1 & 0 & 1 \\ 1 & 2 & 0 \\ 2 & 3 & -2 \end{vmatrix}$$

The first and second new determinants are calculated using the cofactors of row 1 as

$$\begin{aligned} \det(\mathbf{A}) &= 5\begin{vmatrix} 3 & -2 \\ 1 & -5 \end{vmatrix} + 5\begin{vmatrix} 2 & 3 \\ 4 & 1 \end{vmatrix} + 3\begin{vmatrix} 2 & 0 \\ 3 & -2 \end{vmatrix} + 3\begin{vmatrix} 1 & 2 \\ 2 & 3 \end{vmatrix} \\ &= 5(-15+2) + 5(2-12) + 3(-4-0) + 3(3-4) \\ &= -65 - 50 - 12 - 3 \\ \det(\mathbf{A}) &= -130 \end{aligned}$$

As expected, the same value is obtained.

2.3.4 Eigenvectors and Eigenvalues

Consider a square matrix $\mathbf{A}$ and a vector $\mathbf{v}$. The vector $\mathbf{v}$ is an eigenvector of $\mathbf{A}$ if the product $\mathbf{Av}$ results in a vector that has the same direction as $\mathbf{v}$ [14–19]. Therefore an eigenvector of a matrix is a nonzero vector that does not rotate when is applied to it. As shown in Fig. 2.11, the only effects may be to change its length and/or reverse its direction. Thus, there exist a scalar λ such that $\mathbf{Av} = \lambda\mathbf{v}$. The value of λ is an eigenvalue of $\mathbf{A}$. It is clear that for any constant α the vector $\alpha\mathbf{v}$ is also an eigenvector of $\mathbf{A}$ because $\mathbf{A}(\alpha\mathbf{v}) = \alpha\mathbf{Av} = \alpha\lambda\mathbf{v} = \lambda(\alpha\mathbf{v})$. Thus, a scaled eigenvector is also an eigenvector.

If $\mathbf{A}$ is symmetric of order N, then it can be shown that $\mathbf{A}$ has a set of linearly independent eigenvectors denoted $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_N$. As proved above this set is not unique. However the corresponding set of their eigenvalues, denoted $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_N$, which may or may not be equal to each other, is unique. The eigenvalues of the identity matrix are all ones, and every nonzero vector is an eigenvector of $\mathbf{I}$.

In general the eigenvalues of a square matrix $\mathbf{A}$ of order N are obtained from solving the following equation:

$$\mathbf{Av} = \lambda\mathbf{v} \Rightarrow \mathbf{Av} = \lambda\mathbf{Iv} \Rightarrow (\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0 \tag{2.47}$$

Since, by definition, eigenvectors are nonzero, then

$$\mathbf{A} - \lambda\mathbf{I} = 0 \Rightarrow \det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{2.48}$$

The expanded form of Eq. (2.48) is given by

$$\det\begin{bmatrix} a_{11}-\lambda & a_{12} & a_{13} & \cdots & \cdots & a_{1N} \\ a_{21} & a_{22}-\lambda & a_{23} & \cdots & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33}-\lambda & \cdots & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & \cdots & a_{NN}-\lambda \end{bmatrix} = 0 \tag{2.49}$$

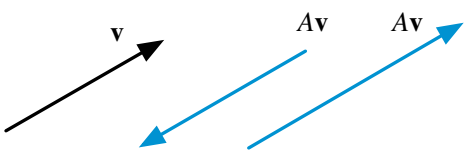


Fig. 2.11 Effects of multiplying a matrix $\mathbf{A}$ by one of its Eigenvectors $\mathbf{v}$

As an example, the eigenvalues of the following square matrix of order 2 are found as:

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 8 & 1 \end{bmatrix} \Rightarrow \begin{vmatrix} \lambda - 3 & 1 \\ 8 & \lambda - 1 \end{vmatrix} = 0 \Rightarrow (\lambda - 3)(\lambda - 1) - 8 = 0$$
$$\therefore \lambda^2 - 4\lambda - 5 = 0 \Rightarrow (\lambda + 1)(\lambda - 5) = 0 \Rightarrow \lambda_1 = -1 \text{ or } \lambda_2 = 5$$

2.3.5 A Symmetric Positive-Definite Matrix

A symmetric matrix $\mathbf{A} = [a_{ij}]$ of order N is positive-definite if for all column vectors $\mathbf{p}$ in R^N the following inequality holds:

$$\mathbf{p}^{\mathrm{T}}\mathbf{A}\mathbf{p} > 0 \tag{2.50}$$

For example, if $\mathbf{A}$ is an order 3 symmetric matrix given by

$$\mathbf{A} = \begin{bmatrix} 5 & 3 & 1 \\ 3 & 7 & 4 \\ 1 & 4 & 8 \end{bmatrix}$$

then Eq. (2.50) for any column vector $\mathbf{p}$ of order 3 gives

$$\mathbf{p}^{\mathrm{T}}\mathbf{A}\mathbf{p} = [a \;\; b \;\; c] \begin{bmatrix} 5 & 3 & 1 \\ 3 & 7 & 4 \\ 1 & 4 & 8 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$
$$= 3(a+b)^2 + (a+c)^2 + 4(b+c)^2 + a^2 \; + \; 4b^2 + 3c^2 > 0$$

which is positive-definite.

If $\mathbf{A}$ is a symmetric positive-definite matrix given by

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & \cdots & a_{NN} \end{bmatrix} \tag{2.51}$$

then, among others, the following properties apply:

1. Any sub-matrix **P** of **A** of order $M(1 \leq M \leq N)$ of the form

$$\mathbf{P} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MM} \end{bmatrix} \tag{2.52}$$

   is also positive-definite.
2. The N eigenvalues of **A**, λ_1, λ_2, λ_3,..., λ_N are positive.
3. If all the eigenvalues of a matrix **A** are positive, then **A** is positive-definite.
4. **A** has a unique decomposition of the form $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$, where **L** is a lower triangular matrix. This decomposition is known as the Cholesky decomposition.

2.3.6 Additional Matrix Operations

If s_1 and s_2 are scalar functions, **I** an identity matrix, and **A**, **B**, and **C** are matrices, then the various matrix operations, addition, subtraction, scalar multiplication, and matrix multiplication, have the following properties listed without proof:

$$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C} \tag{2.53}$$

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \tag{2.54}$$

$$s_1(\mathbf{A} + \mathbf{B}) = s_1\mathbf{A} + s_1\mathbf{B} \tag{2.55}$$

$$(s_1 + s_2)\mathbf{A} = s_1\mathbf{A} + s_2\mathbf{A} \tag{2.56}$$

$$\mathbf{A}(\mathbf{B}\mathbf{C}) = (\mathbf{A}\mathbf{B})\mathbf{C} \tag{2.57}$$

$$\mathbf{A}\mathbf{I} = \mathbf{I}\mathbf{A} = \mathbf{A} \tag{2.58}$$

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{A}\mathbf{B} + \mathbf{A}\mathbf{C} \tag{2.59}$$

$$(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{A}\mathbf{C} + \mathbf{B}\mathbf{C} \tag{2.60}$$

$$(\mathbf{A} + \mathbf{B})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}} + \mathbf{B}^{\mathrm{T}} \tag{2.61}$$

$$(s_1\mathbf{A})^{\mathrm{T}} = s_1\mathbf{A}^{\mathrm{T}} \tag{2.62}$$

$$(\mathbf{A}\mathbf{B})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} \tag{2.63}$$

$$(\mathbf{A}\mathbf{B})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1} \tag{2.64}$$

2.4 Tensors and Tensor Operations

Tensors can be thought of as extensions to the ideas already used when defining quantities like scalars and vectors [2, 20, 21]. A scalar is a tensor of rank zero, and a vector is a tensor of rank one. Tensors of higher rank (2, 3, etc.) can be developed and their main use is to manipulate and transform sets of equations. Since within the scope of this book only tensors of rank two are needed, they will be referred to simply as tensors.

Similar to the flow velocity vector $\mathbf{v}$, the deviatoric stress tensor $\boldsymbol{\tau}$ (Fig. 2.12) will be referred to frequently in this book and is used here to illustrate tensor operations.

Let x, y, and z represent the directions in an orthonormal Cartesian coordinate system, then the stress tensor $\boldsymbol{\tau}$ and its transpose designated with superscript T($\boldsymbol{\tau}^{\mathrm{T}}$) are represented in terms of their components as

$$\boldsymbol{\tau} = \begin{bmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{bmatrix} \quad \boldsymbol{\tau}^{\mathrm{T}} = \begin{bmatrix} \tau_{xx} & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \tau_{yy} & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \tau_{zz} \end{bmatrix} \tag{2.65}$$

Similar to writing a vector in terms of its components, defining the unit vectors $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ in the x, y, and z direction, respectively, the tensor $\boldsymbol{\tau}$ given by Eq. (2.65) can be written in terms of its components as

$$\boldsymbol{\tau} = \mathbf{ii}\tau_{xx} + \mathbf{ij}\tau_{xy} + \mathbf{ik}\tau_{xz} + \mathbf{ji}\tau_{yx} + \mathbf{jj}\tau_{yy} + \mathbf{jk}\tau_{yz} + \mathbf{ki}\tau_{zx} + \mathbf{kj}\tau_{zy} + \mathbf{kk}\tau_{zz} \tag{2.66}$$

Equation (2.66) allows defining a third type of vector product for multiplying two vectors, known as the dyadic product, and resulting in a tensor with its components formed by ordered pairs of the two vectors. In specific, the dyadic product

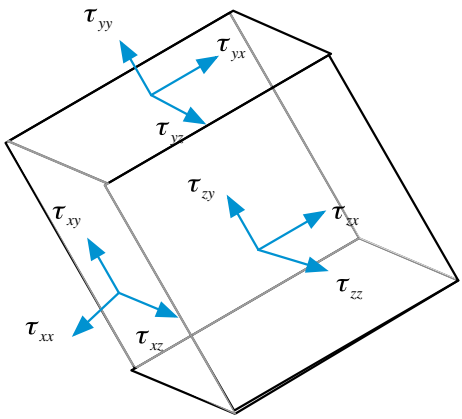


Fig. 2.12 Schematic of a stress tensor field

of a vector $\mathbf{v}$ by itself, arising in the formulation of the momentum equation of fluid flow, gives

$$\left.\begin{aligned}\{\mathbf{vv}\} &= (u\mathbf{i}+v\mathbf{j}+w\mathbf{k})(u\mathbf{i}+v\mathbf{j}+w\mathbf{k})\\ &= \mathbf{ii}uu+\mathbf{ij}uv+\mathbf{ik}uw+\\ &\quad \mathbf{ji}vu+\mathbf{jj}vv+\mathbf{jk}vw+\\ &\quad \mathbf{ki}wu+\mathbf{kj}wv+\mathbf{kk}ww\end{aligned}\right\} \Rightarrow \{\mathbf{vv}\} = \begin{bmatrix} uu & uv & uw\\ vu & vv & vw\\ wu & wv & ww\end{bmatrix} \tag{2.67}$$

The gradient of a vector $\mathbf{v}$ is a tensor given by

$$\left.\begin{aligned}\{\nabla\mathbf{v}\} &= \left(\frac{\partial}{\partial x}\mathbf{i}+\frac{\partial}{\partial y}\mathbf{j}+\frac{\partial}{\partial z}\mathbf{k}\right)(u\mathbf{i}+v\mathbf{j}+w\mathbf{k})\\ &= \mathbf{ii}\frac{\partial u}{\partial x}+\mathbf{ij}\frac{\partial v}{\partial x}+\mathbf{ik}\frac{\partial w}{\partial x}+\\ &\quad \mathbf{ji}\frac{\partial u}{\partial y}+\mathbf{jj}\frac{\partial v}{\partial y}+\mathbf{jk}\frac{\partial w}{\partial y}+\\ &\quad \mathbf{ki}\frac{\partial u}{\partial z}+\mathbf{kj}\frac{\partial v}{\partial z}+\mathbf{kk}\frac{\partial w}{\partial z}\end{aligned}\right\} \Rightarrow \{\nabla\mathbf{v}\} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} & \frac{\partial w}{\partial x}\\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} & \frac{\partial w}{\partial y}\\ \frac{\partial u}{\partial z} & \frac{\partial v}{\partial z} & \frac{\partial w}{\partial z}\end{bmatrix} \tag{2.68}$$

The sum of two tensors $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ is a tensor $\boldsymbol{\Sigma}$ whose components are the sum of the corresponding components of the two tensors, i.e.,

$$\boldsymbol{\Sigma} = \boldsymbol{\sigma}+\boldsymbol{\tau} = \begin{bmatrix} \sigma_{xx}+\tau_{xx} & \sigma_{xy}+\tau_{xy} & \sigma_{xz}+\tau_{xz}\\ \sigma_{yx}+\tau_{yx} & \sigma_{yy}+\tau_{yy} & \sigma_{yz}+\tau_{yz}\\ \sigma_{zx}+\tau_{zx} & \sigma_{zy}+\tau_{zy} & \sigma_{zz}+\tau_{zz}\end{bmatrix} \tag{2.69}$$

Multiplying a tensor $\boldsymbol{\tau}$ by a scalar s results in a tensor whose components are multiplied by that scalar, i.e.,

$$\{s\boldsymbol{\tau}\} = \begin{bmatrix} s\tau_{xx} & s\tau_{xy} & s\tau_{xz}\\ s\tau_{yx} & s\tau_{yy} & s\tau_{yz}\\ s\tau_{zx} & s\tau_{zy} & s\tau_{zz}\end{bmatrix} \tag{2.70}$$

The dot product of a tensor $\boldsymbol{\tau}$ by a vector $\mathbf{v}$ results in the following vector:

$$[\boldsymbol{\tau}\cdot\mathbf{v}] = \begin{pmatrix}\mathbf{ii}\tau_{xx}+\mathbf{ij}\tau_{xy}+\mathbf{ik}\tau_{xz}+\mathbf{ji}\tau_{yx}+\\ \mathbf{jj}\tau_{yy}+\mathbf{jk}\tau_{yz}+\mathbf{ki}\tau_{zx}+\mathbf{kj}\tau_{zy}+\mathbf{kk}\tau_{zz}\end{pmatrix}\cdot(u\mathbf{i}+v\mathbf{j}+w\mathbf{k}) \tag{2.71}$$

which upon expanding becomes

$$\begin{aligned}[\boldsymbol{\tau}\cdot\mathbf{v}] = {} & \mathbf{ii}\cdot\mathbf{i}\tau_{xx}u + \mathbf{ii}\cdot\mathbf{j}\tau_{xx}v + \mathbf{ii}\cdot\mathbf{k}\tau_{xx}w + \mathbf{ij}\cdot\mathbf{i}\tau_{xy}u + \mathbf{ij}\cdot\mathbf{j}\tau_{xy}v \\ & + \mathbf{ij}\cdot\mathbf{k}\tau_{xy}w + \mathbf{ik}\cdot\mathbf{i}\tau_{xz}u + \mathbf{ik}\cdot\mathbf{j}\tau_{xz}v + \mathbf{ik}\cdot\mathbf{k}\tau_{xz}w + \mathbf{ji}\cdot\mathbf{i}\tau_{yx}u \\ & + \mathbf{ji}\cdot\mathbf{j}\tau_{yx}v + \mathbf{ji}\cdot\mathbf{k}\tau_{yx}w + \mathbf{jj}\cdot\mathbf{i}\tau_{yy}u + \mathbf{jj}\cdot\mathbf{j}\tau_{yy}v + \mathbf{jj}\cdot\mathbf{k}\tau_{yy}w \\ & + \mathbf{jk}\cdot\mathbf{i}\tau_{yz}u + \mathbf{jk}\cdot\mathbf{j}\tau_{yz}v + \mathbf{jk}\cdot\mathbf{k}\tau_{yz}w + \mathbf{ki}\cdot\mathbf{i}\tau_{zx}u + \mathbf{ki}\cdot\mathbf{j}\tau_{zx}v \\ & + \mathbf{ki}\cdot\mathbf{k}\tau_{zx}w + \mathbf{kj}\cdot\mathbf{i}\tau_{zy}u + \mathbf{kj}\cdot\mathbf{j}\tau_{zy}v + \mathbf{kj}\cdot\mathbf{k}\tau_{zy}w + \mathbf{kk}\cdot\mathbf{i}\tau_{zz}u \\ & + \mathbf{kk}\cdot\mathbf{j}\tau_{zz}v + \mathbf{kk}\cdot\mathbf{k}\tau_{zz}w \end{aligned} \tag{2.72}$$

Using Eq. (2.8), Eq. (2.72) reduces to

$$[\boldsymbol{\tau}\cdot\mathbf{v}] = \left(\tau_{xx}u + \tau_{xy}v + \tau_{xz}w\right)\mathbf{i} + \left(\tau_{yx}u + \tau_{yy}v + \tau_{yz}w\right)\mathbf{j} + \left(\tau_{zx}u + \tau_{zy}v + \tau_{zz}w\right)\mathbf{k} \tag{2.73}$$

The above equation can be derived using matrix multiplication as

$$[\boldsymbol{\tau}\cdot\mathbf{v}] = \begin{bmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} \tau_{xx}u + \tau_{xy}v + \tau_{xz}w \\ \tau_{yx}u + \tau_{yy}v + \tau_{yz}w \\ \tau_{zx}u + \tau_{zy}v + \tau_{zz}w \end{bmatrix} \tag{2.74}$$

In a similar way the divergence of a tensor $\boldsymbol{\tau}$ is found to be a vector given by

$$\begin{aligned}[\nabla\cdot\boldsymbol{\tau}] = {} & \left(\frac{\partial\tau_{xx}}{\partial x} + \frac{\partial\tau_{yx}}{\partial y} + \frac{\partial\tau_{zx}}{\partial z}\right)\mathbf{i} + \left(\frac{\partial\tau_{xy}}{\partial x} + \frac{\partial\tau_{yy}}{\partial y} + \frac{\partial\tau_{zy}}{\partial z}\right)\mathbf{j} \\ & + \left(\frac{\partial\tau_{xz}}{\partial x} + \frac{\partial\tau_{yz}}{\partial y} + \frac{\partial\tau_{zz}}{\partial z}\right)\mathbf{k} \end{aligned} \tag{2.75}$$

The double dot product of two tensors $\boldsymbol{\tau}$ and $\{\nabla\mathbf{v}\}$ is a scalar computed as

$$(\boldsymbol{\tau} : \nabla\mathbf{v}) = \begin{pmatrix} \mathbf{ii}\tau_{xx} + \mathbf{ij}\tau_{xy} + \mathbf{ik}\tau_{xz} + \\ \mathbf{ji}\tau_{yx} + \mathbf{jj}\tau_{yy} + \mathbf{jk}\tau_{yz} + \\ \mathbf{ki}\tau_{zx} + \mathbf{kj}\tau_{zy} + \mathbf{kk}\tau_{zz} \end{pmatrix} : \begin{pmatrix} \mathbf{ii}\dfrac{\partial u}{\partial x} + \mathbf{ij}\dfrac{\partial v}{\partial x} + \mathbf{ik}\dfrac{\partial w}{\partial x} + \\ \mathbf{ji}\dfrac{\partial u}{\partial y} + \mathbf{jj}\dfrac{\partial v}{\partial y} + \mathbf{jk}\dfrac{\partial w}{\partial y} + \\ \mathbf{ki}\dfrac{\partial u}{\partial z} + \mathbf{kj}\dfrac{\partial v}{\partial z} + \mathbf{kk}\dfrac{\partial w}{\partial z} \end{pmatrix} \tag{2.76}$$

The final value is obtained by expanding the above product and performing the double dot product on the various terms. For example,

$$\mathbf{ij}\tau_{xy} : \mathbf{ji}\frac{\partial u}{\partial y} = \mathbf{i}\underbrace{\mathbf{j} : \mathbf{j}}_{=1}\mathbf{i}\tau_{xy}\frac{\partial u}{\partial y} = \underbrace{\mathbf{i}\cdot\mathbf{i}}_{=1}\tau_{xy}\frac{\partial u}{\partial y} = \tau_{xy}\frac{\partial u}{\partial y} \tag{2.77}$$

Performing the same steps on every term in the expanded product, the final form of $(\boldsymbol{\tau} : \nabla \mathbf{v})$ is obtained as

$$(\boldsymbol{\tau} : \nabla \mathbf{v}) = \tau_{xx}\frac{\partial u}{\partial x} + \tau_{xy}\frac{\partial u}{\partial y} + \tau_{xz}\frac{\partial u}{\partial z} + \tau_{yx}\frac{\partial v}{\partial x} + \tau_{yy}\frac{\partial v}{\partial y} + \tau_{yz}\frac{\partial v}{\partial z} + \tau_{zx}\frac{\partial w}{\partial x} + \tau_{zy}\frac{\partial w}{\partial y} + \tau_{zz}\frac{\partial w}{\partial z} \tag{2.78}$$

2.5 Fundamental Theorems of Vector Calculus

All mathematical formulations presented in this book will be performed using vectors. Therefore a good knowledge of the fundamental theorems of vector calculus is helpful. A brief review of some of these theorems is presented next.

2.5.1 Gradient Theorem for Line Integrals

The gradient theorem for line integrals relates a line integral to the values of a function at its endpoints [22]. It states that if C is a smooth curve, as shown in Fig. 2.13, described by the vector $\mathbf{r}(t) = \mathbf{r}[x(t), y(t), z(t)]$ for $a \leq t \leq b$, and s is a scalar function whose gradient, ∇s, is continuous on C, then

$$\int_C \nabla s \cdot d\mathbf{r} = s(r(b)) - s(r(a)) \tag{2.79}$$

where a and b are the endpoints of C. It follows that the value of the integral over a closed contour is zero.

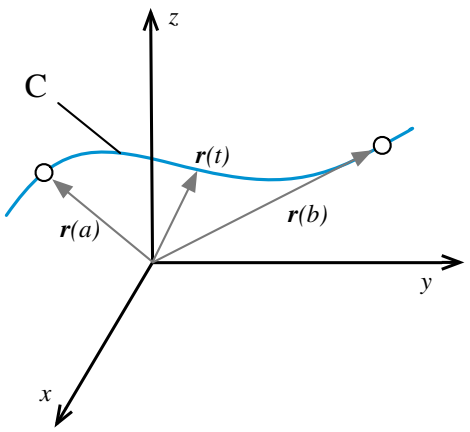


Fig. 2.13 A schematic depiction of a curve C of a scalar function s showing its end points and the position vector $\mathbf{r}(t)$

2.5.2 Green's Theorem

Green's theorem expresses the contour integral of a simple closed curve C in terms of the double integral of the two dimensional region R bounded by C [23–26].

Let C denotes the closed contour (Fig. 2.14) of a two dimensional region R. If $u(x, y)$ and $v(x, y)$ are functions of continuous partial derivatives defined on R, then

$$\oint_C (udx + vdy) = \iint_R \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) dxdy \tag{2.80}$$

In Eq. (2.80) the contour integral along C is taken positive in the counter-clockwise direction.

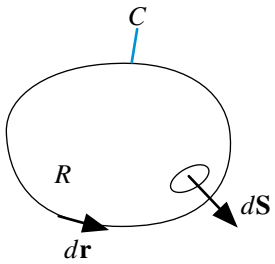


Fig. 2.14 Schematic of a region R and its closed contour C

Green's theorem can be written in a more compact form using vectors. For that purpose defining $d\mathbf{r}$, $\mathbf{v}$ and the area vector $d\mathbf{S}$ as

$$d\mathbf{r} = dx\mathbf{i} + dy\mathbf{j} \qquad \mathbf{v} = u\mathbf{i} + v\mathbf{j} \qquad d\mathbf{S} = dxdy\mathbf{k} \tag{2.81}$$

then the vector form of Green's theorem is given by

$$\oint_C \mathbf{v} \cdot d\mathbf{r} = \iint_R [\nabla \times \mathbf{v}] \cdot d\mathbf{S} \tag{2.82}$$

Green's theorem is helpful for computing line integrals arising in two-dimensional flows.

Example 7

Compute $\oint_C 2y^3\,dx + 3xy^2\,dy$ *where* C *is the CCW-oriented boundary of the region* R *shown in* Fig. 2.15.

The vector field in the above integral is $(u, v) = (2y^3, 3xy^2)$. The line integral can be computed directly. But, it is more easily computed using Green's theorem using a double integral. Applying Green's theorem the integrand is obtained as

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = 3y^2 - 6y^2 = -3y^2$$

Since the line integral is over a semi circle, the region R is mathematically given by

$$-1 \le x \le 1$$
$$0 \le y \le \sqrt{1 - x^2}$$

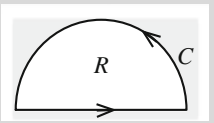


Fig. 2.15 Example 7

The value of the integral is obtained as

$$\oint_C 2y^3\,dx + 3xy^2\,dy = \iint_D \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) dA = -3\int_{-1}^{1}\int_{0}^{\sqrt{1-x^2}} y^2\,dy dx$$
$$= -3\int_{-1}^{1}\left(\frac{y^3}{3}\Big|_{y=0}^{y=\sqrt{1-x^2}}\right) dx = -\int_{-1}^{1}\left(1 - x^2\right)^{3/2} dx$$

Let $x = \cos\theta \Rightarrow dx = -\sin\theta d\theta$

Thus

$$\oint_C 2y^3\,dx + 3xy^2\,dy = -\int_0^{\pi} \sin^2\theta d\theta + \int_0^{\pi} \sin^2\theta\cos^2\theta d\theta$$
$$= -\left[\frac{\theta}{2} - \frac{\sin 2\theta}{4}\right]_0^{\pi} + \left[\frac{\theta}{8} - \frac{\sin 4\theta}{32}\right]_0^{\pi}$$
$$= -\frac{3\pi}{8}$$

2.5.3 Stokes' Theorem

Stokes' theorem is a higher dimensional version of Green's theorem [27–29]. Whereas Green's theorem relates a line integral to a double integral, Stokes theorem relates a line integral to a surface integral. Let $\mathbf{v}$ be a vector field, S an oriented surface, and C the boundary curve of S, oriented using the right-hand rule, as depicted in Fig. 2.16. Stokes' theorem states the following:

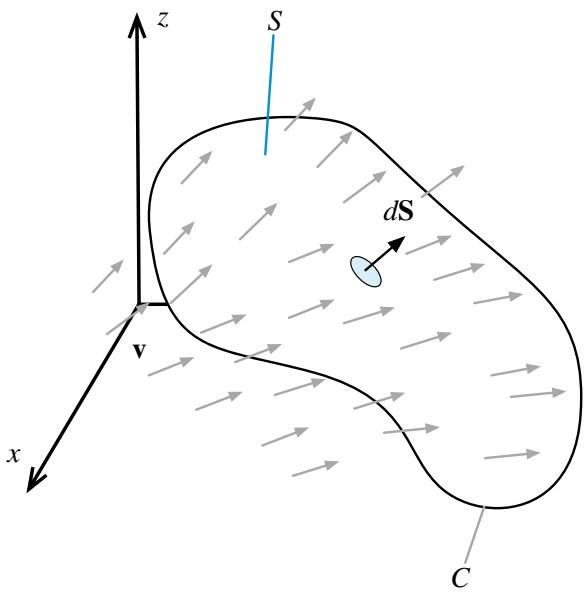


Fig. 2.16 A surface S in a three-dimensional space of contour C

$$\int_S [\nabla \times \mathbf{v}] \cdot d\mathbf{S} = \oint_C \mathbf{v} \cdot d\mathbf{r} \tag{2.83}$$

where $\mathbf{r}$ is such that $d\mathbf{r}/ds$ is the unit tangent vector and s the arc length of C. The curve of the line integral, C, must have positive orientation, meaning that $d\mathbf{r}$ points counterclockwise when the surface normal, $d\mathbf{S}$, points toward the viewer, following the right-hand rule.

2.5.4 Divergence Theorem

Let V represents a volume in three-dimensional space (Fig. 2.17) of boundary S. Let $\mathbf{n}$ be the outward pointing unit vector normal to S. If $\mathbf{v}$ is a vector field defined on V, then the divergence theorem [30, 31] (also known as Gauss' theorem) states that

$$\int_V (\nabla \cdot \mathbf{v}) dV = \oint_S \mathbf{v} \cdot \mathbf{n}\, dS \tag{2.84}$$

The divergence theorem implies that the net flux of a vector field through a closed surface is equal to the total volume of all sources and sinks (i.e., the volume integral of its divergence) over the region inside the surface. It is an important theorem for fluid dynamics.

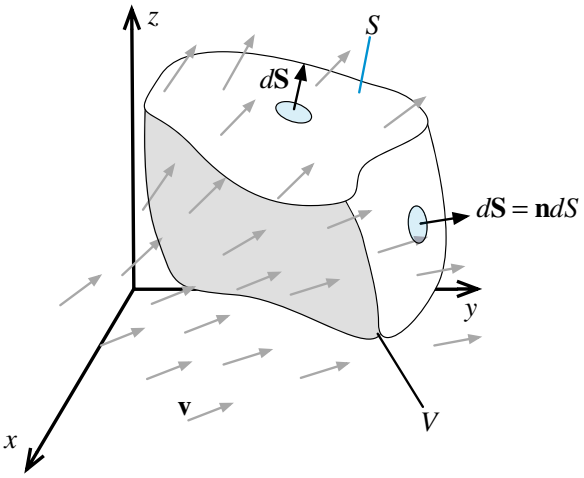


Fig. 2.17 A volume in three-dimensional space with a piecewise smooth boundary S

The divergence theorem can be used in different contexts to derive many other useful identities (corollaries) [32]. In specific it can be applied to the product of a scalar function, s, and a non-zero constant vector, to derive the following important relation:

$$\int_V [\nabla s] dV = \oint_S s d\mathbf{S} \tag{2.85}$$

The divergence theorem is equally applicable to tensors, in which case it is written as

$$\int_V [\nabla \cdot \boldsymbol{\tau}] dV = \oint_S [\boldsymbol{\tau} \cdot \mathbf{n}] dS \tag{2.86}$$

Example 8
Use the divergence theorem to evaluate

$$\oiint_{\partial V} \mathbf{F} \cdot d\mathbf{S}$$

where $\mathbf{F} = \left(3x + z^5\right)\mathbf{i} + \left(y^2 - \sin(x^2 z)\right)\mathbf{j} + \left(xz + ye^{x^5}\right)\mathbf{k}$
and V is a box defined by

$$0 \le x \le 1 \quad 0 \le y \le 3 \quad 0 \le z \le 2$$

with an outward pointing surface

Solution
This is a difficult field to integrate however using the divergence theorem it can be transformed to

$$\oiint_{\partial V} \mathbf{F} \cdot d\mathbf{S} = \iiint_{V} (\nabla \cdot \mathbf{F}) dV$$

where the divergence of $\mathbf{F}$ is obtained as

$$\nabla \cdot \mathbf{F} = 3 + 2y + x$$

integrating over the box, the integral is evaluated as

$$\oiint_{\partial V} \mathbf{F} \cdot d\mathbf{S} = \int_0^1 \int_0^3 \int_0^2 (3 + 2y + x) dz dy dx = \int_0^1 \int_0^3 (6 + 4y + 2x) dy dx$$
$$= \int_0^1 (18 + 18 + 6x) dx = 36 + 3 = 39$$

2.5.5 *Leibniz Integral Rule*

The Leibniz integral rule gives a formula for differentiating a definite integral whose limits are functions of the differential variable [33–36]. Let $\phi(x, t)$ represents a function that depends on a space variable x and time t. Then Leibniz integral rule can be stated as follows

$$\frac{d}{dt} \int_{a(t)}^{b(t)} \phi(x,t) dx = \underbrace{\int_{a(t)}^{b(t)} \frac{\partial \phi}{\partial t} dx}_{Term\ I} + \underbrace{\phi(b(t),t) \frac{\partial b}{\partial t}}_{Term\ II} - \underbrace{\phi(a(t),t) \frac{\partial a}{\partial t}}_{Term\ III} \tag{2.87}$$

The meaning of the various terms in Eq. (2.87) can be inferred from Fig. 2.18. The first term on the right side gives the change in the integral because ϕ is changing with time t, while the second and third terms accounts for the gain and loss in area as the upper and lower bounds are moved, respectively.

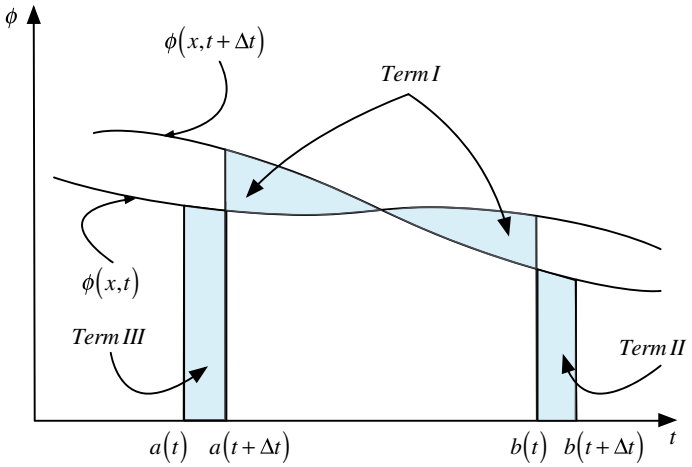


Fig. 2.18 Curves showing the spatial distribution of a function at times t and $t + \Delta t$

The three-dimensional form of this formula applied to a volume $V(t)$ enclosed by a surface $S(t)$ with its surface elements moving with a velocity $\mathbf{v}_s$ can be written as

$$\frac{d}{dt}\int\limits_{V(t)} \phi dV = \int\limits_{V(t)} \frac{\partial \phi}{\partial t} dV + \int\limits_{S(t)} \phi(\mathbf{v}_s \cdot \mathbf{n}) dS \tag{2.88}$$

where $\phi(t, \mathbf{x})$ is a scalar function of space and time. For a non-moving volume V, the equation reduces to

$$\frac{d}{dt}\int\limits_{V} \phi dV = \int\limits_{V} \frac{\partial \phi}{\partial t} dV \tag{2.89}$$

The above equations are also applicable to vectors and tensors.

2.6 Closure

The chapter offered a brief review of vector and tensor operations. In addition the fundamental theorems of vector calculus were presented. The next chapter will rely on information presented in this chapter to derive the conservation equations governing the transfer phenomena of interest in this book.

2.7 Exercises

Exercise 1
Let $\mathbf{v}_1, \mathbf{v}_2$ and $\mathbf{v}_3$ be three vectors given by

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \\ -5 \end{bmatrix} \quad \mathbf{v}_2 = \begin{bmatrix} -1 \\ -1 \\ 10 \end{bmatrix} \quad \mathbf{v}_3 = \begin{bmatrix} 8 \\ -5 \\ -2 \end{bmatrix}$$

Find:

a. $\mathbf{v}_1 + \mathbf{v}_2, \quad \mathbf{v}_1 + 2\mathbf{v}_2, \quad 3\mathbf{v}_2 - 4\mathbf{v}_3$
b. $|\mathbf{v}_1|, |\mathbf{v}_2|, |\mathbf{v}_3|$
c. $\mathbf{v}_1 \cdot \mathbf{v}_2, \quad \mathbf{v}_3 \times \mathbf{v}_2, \quad \mathbf{v}_2 \cdot (\mathbf{v}_1 \times \mathbf{v}_3)$
d. A unit vector in the direction of $(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)$

Exercise 2
Let $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ be unit vectors in the x, y, and z direction, respectively, and let $\mathbf{v}$ be any vector, which in a Cartesian coordinate system is given by

$$\mathbf{v} = u\mathbf{i} + v\mathbf{j} + w\mathbf{k}$$

Prove that

$$\mathbf{v} = C[\mathbf{i} \times (\mathbf{v} \times \mathbf{i}) + \mathbf{j} \times (\mathbf{v} \times \mathbf{j}) + \mathbf{k} \times (\mathbf{v} \times \mathbf{k})]$$

where C is a constant to be determined.

Exercise 3
Find ∇s if s is the scalar function given by

a. $s = y^2 e^{2x-3z}$
b. $s = \mathrm{Ln}(x + y^2 + z^3)$
c. $s = \tan^{-1}\left(\frac{x}{yz}\right)$

Exercise 4
If s is a scalar function and $\mathbf{v}$ is a vector function, prove the following identities:

a. $\nabla \times (\nabla s) = \mathbf{0}$
b. $\nabla \cdot (s\mathbf{v}) = s\nabla \cdot \mathbf{v} + \mathbf{v} \cdot \nabla s$
c. $\nabla \times (s\mathbf{v}) = s\nabla \times \mathbf{v} + \nabla s \times \mathbf{v}$
d. $\nabla \cdot (\mathbf{v}_1 \times \mathbf{v}_2) = \mathbf{v}_2 \cdot (\nabla \times \mathbf{v}_1) - \mathbf{v}_1 \cdot (\nabla \times \mathbf{v}_2)$

Exercise 5
Use Green's theorem to calculate the area enclosed by an ellipse of semi-major and semi-minor axes a and b, respectively.

Exercise 6
Find the Laplacian of the scalar s $(\nabla^2 s)$ for the cases when s is given by:

a. $s = x^3 + z^2 e^{2y-3x}$
b. $s = z + \mathrm{Ln}(x + y)$
c. $s = \sin^{-1}(x + y + z)$

Exercise 7
Verify the divergence theorem for the parallelepiped with centre at the origin and faces in the planes $x = \pm 2, y = \pm 1, z = \pm 4$ and $\mathbf{v}$ given by

a. $\mathbf{v} = 5\mathbf{i} + 7\mathbf{j} - 3\mathbf{k}$
b. $\mathbf{v} = \mathbf{i}(y - z) + \mathbf{j}(x - z) + \mathbf{k}(x - y)$
c. $\mathbf{v} = \mathbf{i}y^2 z + \mathbf{j}xz^2 + \mathbf{k}x^2\mathrm{y}$

Exercise 8
For a surface S representing the upper half of a cube centered at the origin, with one of its vertices at $(1, 1, 1)$, and with edges parallel to the axes, verify Stokes's theorem for the case when the curve C is the intersection of S with the xy plane and the vector $\mathbf{v}$ is given by

$$\mathbf{v} = \mathbf{i}(y + z) + \mathbf{j}(x + z) + \mathbf{k}(x + y)$$

Exercise 9
Find a function F for which the divergence is the given function K in the following cases:

a. $K(x, y, z) = \pi$.
b. $K(x, y, z) = z^2 x$.
c. $K(x, y, z) = \sqrt{(x^2 + z^2)}$

Exercise 10
Use the divergence theorem to evaluate the integral $\iint_{\partial F} (6x\mathbf{i} + 4\mathrm{y}\mathbf{j}) \cdot d\mathbf{F}$ where the surface is a sphere defined as $\partial F \rightarrow x^2 + y^2 + z^2 = 10$.

Exercise 11
Let $\mathbf{F}$ be a radial vector field defined as $\mathbf{F} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ and let C to be a solid cylinder of radius r and height h with its axis coinciding with the x-axis and its bottom and top faces located along the $x = 0$ and $x = b$ plane, respectively. Verify Gauss theorem in both flux and divergence forms.

Exercise 12
Given a square matrix $\mathbf{A}$ defined as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & \ddots & \vdots \\ \cdots & \cdots & \cdots \end{pmatrix}$$

decompose it as

$$\mathbf{A} = \frac{1}{2}\left(\mathbf{A} + \mathbf{A}^{\mathrm{T}}\right) + \frac{1}{2}\left(\mathbf{A} - \mathbf{A}^{\mathrm{T}}\right)$$

and show that

a. $\frac{1}{2}\left(\mathbf{A} + \mathbf{A}^{\mathrm{T}}\right)$ is symmetric
b. $\frac{1}{2}\left(\mathbf{A} - \mathbf{A}^{\mathrm{T}}\right)$ is anti-symmetric

Exercise 13
Given two tensors $\mathbf{A}$ and $\mathbf{B}$ defined as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & \ddots & \vdots \\ \cdots & \cdots & \cdots \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots \\ b_{21} & \ddots & \vdots \\ \cdots & \cdots & \cdots \end{pmatrix}$$

a. Calculate the double inner product $\mathbf{A} : \mathbf{B}$.
b. Prove that $(\mathbf{A} + \mathbf{B})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}} + \mathbf{B}^{\mathrm{T}}$ and $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$
c. Evaluate $\nabla \cdot \mathbf{A} + \nabla \cdot \mathbf{B}$.

References

1. Arfken G (1985) Mathematical methods for physicists, 3rd edn. Academic Press, Orlando, FL
2. Aris R (1989) Vectors, tensors, and the basic equations of fluid mechanics. Dover, New York
3. Crowe MJ (1985) A history of vector analysis: the evolution of the idea of a vectorial system. Dover, New York

4. Marsden JE, Tromba AJ (1996) Vector calculus. WH Freeman, New York
5. Jeffreys H, Jeffreys BS (1988) methods of mathematical physics. Cambridge University Press, Cambridge, England
6. Morse PM, Feshbach H (1953) Methods of theoretical physics, Part I. McGraw-Hill, New York
7. Schey HM (1973) Div, grad, curl, and all that: an informal text on vector calculus. Norton, New York
8. Schwartz M, Green S, Rutledge W (1960) A vector analysis with applications to geometry and physics. Harper Brothers, New York
9. M1 Anton H (1987) Elementary linear algebra. Wiley, New York
10. Bretscher O (2005) Linear algebra with applications. Prentice Hall, New Jersey
11. Bronson R (1989), Schaum's outline of theory and problems of matrix operations. McGraw–Hill, New York
12. Arnold VI, Cooke R (1992) Ordinary differential equations. Springer-Verlag, Berlin, DE; New York, NY
13. Horn RA, Johnson CR (1985) Matrix analysis. Cambridge University Press, Cambridge
14. Brown WC (1991) Matrices and vector spaces. Marcel Dekker, New York
15. Golub GH, Van Loan CF (1996) Matrix Computations. Johns Hopkins, Baltimore
16. Greub WH (1975) Linear algebra, graduate texts in mathematics. Springer-Verlag, Berlin, DE; New York, NY
17. Lang S (1987) Linear algebra. Springer-Verlag, Berlin, DE; New York, NY
18. Mirsky L (1990) An introduction to linear algebra. Courier Dover Publications, New York
19. Nering ED (1970) Linear algebra and matrix theory. Wiley, New York
20. Spiegel MR (1959) Schaum's outline of theory and problems of vector analysis and an introduction to tensor analysis. Schaum, New York
21. Heinbockel JH (2001) Introduction to tensor calculus and continuum mechanics. Trafford Publishing, Victoria
22. Williamson R, Trotter H (2004) Multivariable mathematics. Pearson Education, Inc, New York
23. Cauchy A (1846) Sur les intégrales qui s'étendent à tous les points d'une courbe fermée. Comptes rendus 23:251–255
24. Riley KF, Hobson MP, Bence SJ (2010) Mathematical methods for physics and engineering. Cambridge University Press, Cambridge
25. Spiegel MR, Lipschutz S, Spellman D (2009) Vector analysis. Schaum's Outlines, McGraw Hill (USA)
26. Wrede R, Spiegel MR (2010) Advanced calculus. Schaum's Outline Series
27. Katz VJ (1979) The history of stokes's theorem. Math Mag (Math Assoc Am) 52:146–156
28. Morse PM, Feshbach H (1953) Methods of theoretical physics, Part I. McGraw-Hill, New York
29. Stewart J (2008) Vector calculus, Calculus: early transcendentals. Thomson Brooks/Cole, Connecticut
30. Lerner RG, Trigg GL (1994) Encyclopaedia of physics. VHC
31. Byron F, Fuller R (1992) Mathematics of classical and quantum physics. Dover Publications, New York
32. Spiegel MR, Lipschutz S, Spellman D (2009) Vector analysis. Schaum's Outlines, McGraw Hill
33. Flanders H (1973) Differentiation under the integral sign. Am Math Monthly 80(6):615–627
34. Boros G, Moll V (2004) Irresistible integrals: symbolics, analysis and experiments in the evaluation of integrals. Cambridge University Press, Cambridge, England
35. Hijab O (1997) Introduction to calculus and classical analysis. Springer, New York
36. Kaplan W (1992) Advanced calculus. Addison-Wesley, Reading, MA

Chapter 3
Mathematical Description of Physical Phenomena

Abstract The chapter provides an overview of the conservation principles governing fluid flow, heat and mass transfer, and other related transport phenomena of interest in this book. The physical laws controlling the conservation principles are translated into mathematical relations, written in the form of partial differential equations, representing the needed vehicle for their simulations. First the continuity, momentum, and energy equations (collectively known as the Navier-Stokes equations) expressing the principles of conservation of mass, momentum, and total energy, respectively, are derived. This is followed by the development of a typical conservation equation for a general scalar, vector, or tensor quantity. The mathematical properties of the various terms in these equations are also examined. Moreover, the common practice of writing the conservation equations in a non-dimensional form using dimensionless quantities is explained and some of the dimensionless groups resulting from the application of this procedure, which are very useful for performing parametric studies of engineering problems, are discussed.

3.1 Introduction

Researchers and practitioners of computational fluid dynamics encounter and work with the Navier-Stokes equations [1, 2] almost on daily basis. Many do not realize that these equations are over one hundred seventy years old. Whereas the name Navier-Stokes initially referred to the conservation equation of linear momentum, it is used nowadays to denote collectively the conservation equations of mass, momentum, and energy. These equations can be used to model a wide range of fluid flow configurations, whether it is the flow in a hurricane or in a turbomachine, around an airplane or a submarine, in arteries or in lungs, in pumps or in compressors, the Navier-Stokes equations can describe all these phenomena.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_3

3.2 Classification of Fluid Flows

Fluids, which denote liquids and gases are substances that do not permanently change under a large stress (force per unit area). Whereas a solid resists an applied shear or tangential stress by deforming, a fluid cannot and a shear stress applied to a fluid puts it to motion. Moreover, unlike solids which have well-defined shapes, fluids do not have a definite shape. While gases are fluids that completely fill their domains, liquids are fluids that form a free surface in the presence of a gravitational field.

In analyzing fluid flow phenomena [3–6], attention is focused on what happens at the macroscopic rather than the microscopic scale. It is also assumed that the fluid is a continuum, so that its physical and flow properties are defined at every point in space. Within this assumption, fluid flow behavior can be categorized as either Newtonian or non-Newtonian. Newtonian fluids are characterized by a linear relationship between the shear stress and the shear rate, with the molecular viscosity μ, which is a measure of the ability of a fluid subjected to a stress to resist deformation, representing the slope of the linear function. On the other hand, for non-Newtonian fluids this relationship is nonlinear. Similarly fluid flow can be classified into various classes, such as one-dimensional or multi-dimensional, single phase or multi-phase, steady or unsteady, real (viscous) or ideal (inviscid), compressible or incompressible, turbulent or laminar, and rotational or irrotational, among others. The purpose of these classifications is to simplify the process of analysis and modeling of fluid flow phenomena.

Flows are also classified mathematically according to the partial differential equations describing them. Second order partial differential equations in two independent variables, for example, are categorized as hyperbolic, parabolic, or elliptic. In these equations information travels along two characteristic lines, which may be real and distinct, real and coincident, or complex depending on whether they are of the hyperbolic, parabolic, or elliptic type, respectively. This variation in the nature of the equations necessitates different solution methodologies that should also be recognized by any numerical method used to solve them.

As will be shown in this chapter, fluid flows are governed by the Navier-Stokes equations, which are highly nonlinear second order partial differential equations in four independent variables since, in general, flows are unsteady and three dimensional. Therefore, the above classification does not really apply to them. Nevertheless, the same terminology is used in their categorization as they share many of the properties characterizing second order equations in two independent variables. Transient and supersonic flows are hyperbolic, boundary layer flows are parabolic, and recirculating flows are elliptic. As flows may be subsonic in a certain part of the domain and supersonic in other parts (e.g., flow in a converging-diverging nozzle), or viscous dominated close to walls and essentially inviscid in the core region, it is hard to describe a flow as falling under one of the above three types and in general it is of the mixed type. This categorization is numerically translated into the following: parabolic flows that are affected by upstream locations only, elliptic flows by both upstream and downstream locations, and hyperbolic flows supporting discontinuities in the solution, e.g., shock waves.

3.3 Eulerian and Lagrangian Description of Conservation Laws

The principle of conservation states that for an isolated system certain physical measurable quantities are conserved over a local region. This conservation principle or conservation law is an axiom that cannot be proven mathematically but can be expressed by a mathematical relation. Laws of this type govern several physical quantities such as mass, momentum, and energy (i.e., the Navier-Stokes equations).

The conservation laws involving fluid flow and related transfer phenomena can be mathematically formulated following either a Lagrangian (material volume, MV) or an Eulerian (control volume) approach [7]. In the Lagrangian specification of the flow field (Fig. 3.1a), the fluid is subdivided into fluid parcels and every fluid parcel is followed as it moves through space and time. These parcels are tagged using a time-independent position vector field $\mathbf{x}_0$, usually selected to be the parcels' centre of mass at some initial time t_0, and the flow is described by a function $\mathbf{x}(t, \mathbf{x}_0)$. The path line described by a fluid parcel (Fig. 3.1a) is obtained as the collection of positions occupied at different times.

On the other hand, the Eulerian approach (Fig. 3.1b) focuses on specific locations in the flow region as time passes. Thus the flow variables are functions of position $\mathbf{x}$ and time t and the flow velocity is represented by $\mathbf{v}(t, \mathbf{x})$. As the derivative of the position of a fluid parcel $\mathbf{x}_0$ with respect to time represents its velocity, the two specifications are related by

$$\mathbf{v}(t, \mathbf{x}(\mathbf{x}_0, t)) = \frac{\partial}{\partial t}\mathbf{x}(t, \mathbf{x}_0) \tag{3.1}$$

Based on the above description, changes in the properties of a moving fluid can be measured either on a fixed point in space while fluid particles are crossing it (Eulerian), or by following a fluid parcel along its path (Lagrangian).

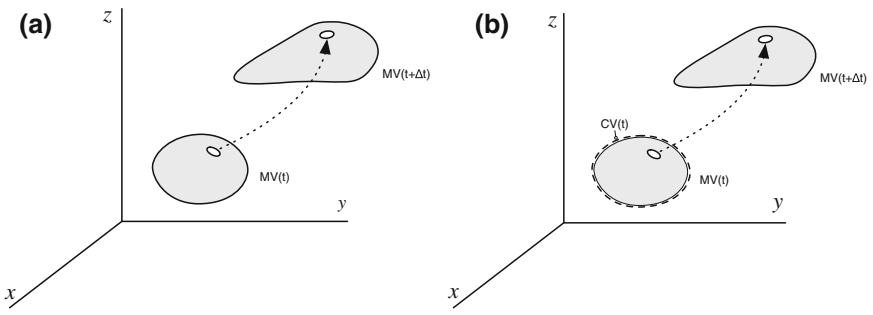


Fig. 3.1 **a** Lagrangian and **b** Eulerian specification of the flow field

3.3.1 Substantial Versus Local Derivative

The derivative (rate of change) of a field variable $\phi(t, \mathbf{x}(t))$, which may be a scalar or a vector quantity representing density, velocity, temperature, etc., with respect to a fixed position in space is called the Eulerian derivative $(\partial\phi/\partial t)$ while the derivative following a moving fluid parcel is called the Lagrangian, substantial, or material derivative and is denoted by $(D\phi/Dt)$. The substantial derivative of variable ϕ, which can be derived through application of the chain rule to account for changes induced by all independent variables along the path, is given by

$$\begin{aligned}\frac{D\phi}{Dt} &= \frac{\partial\phi}{\partial t}\frac{dt}{dt} + \frac{\partial\phi}{\partial x}\underbrace{\frac{dx}{dt}}_{u} + \frac{\partial\phi}{\partial y}\underbrace{\frac{dy}{dt}}_{v} + \frac{\partial\phi}{\partial z}\underbrace{\frac{dz}{dt}}_{w} \\ &= \frac{\partial\phi}{\partial t} + u\frac{\partial\phi}{\partial x} + v\frac{\partial\phi}{\partial y} + w\frac{\partial\phi}{\partial z} \\ &= \underbrace{\frac{\partial\phi}{\partial t}}_{\substack{\text{local rate}\\ \text{of change}}} + \underbrace{\mathbf{v}\cdot\nabla\phi}_{\substack{\text{convective rate}\\ \text{of change}}}\end{aligned} \tag{3.2}$$

where $\mathbf{v}$ is the velocity vector and ∇ is the "del" or "gradient" operator defined earlier [6–8].

Equation (3.2) shows that the total rate of change of the function ϕ as a fluid parcel moves through a flow field described by its Eulerian specification $\mathbf{v}$ from position $\mathbf{x}$ at time t to position $\mathbf{x} + \mathbf{v}\delta t$ at time $t + \delta t$ (Fig. 3.2) is equal to the sum of the local and convective rates of change of ϕ.

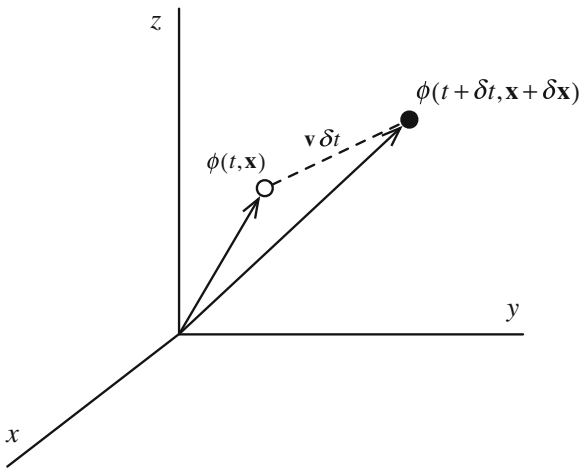


Fig. 3.2 Total rate of change of the field variable ϕ between time t and $t + \delta t$

An important example of a material derivative is $D\mathbf{v}/Dt$, the rate of change of velocity following the flow, which is the acceleration vector given by

$$\frac{D\mathbf{v}}{Dt} = \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} \tag{3.3}$$

In this book, the conservation laws are described following an Eulerian formulation where the focus is on the flow within a specified region in space, called control volume. This choice is based on the fact that the Eulerian approach follows a field (system) rather than a particle approach, it abandons the tedious and often unnecessary task of tracking individual particles, and focuses attention on what happens at a fixed point (or volume) as different particles go by. Moreover, a critical shortcoming of the Lagrangian approach is its inability to control the domain of interest since fluid parcels travel to where the flow takes them, which may not be the region of interest. This limits the usefulness of the approach as in most fluid flow applications fluid properties in a fixed region are required, e.g., the shear stress on the surface of a moving train, and not the properties of moving material volumes. Nevertheless it should be mentioned that the Eulerian approach introduces into the conservation equations the local effect of transport by the fluid flow through the advective rate of change term, $\mathbf{v} \cdot \nabla\phi$, which represents the product of an unknown velocity field and the gradient of an unknown variable field. This nonlinearity leads to the most interesting and most challenging phenomena of fluid flows.

3.3.2 Reynolds Transport Theorem

The conservation laws mentioned above apply to moving material volumes of fluids (Fig. 3.1), and not to fixed points or control volumes. In order to express these laws following an Eulerian approach, there is a need to know the Eulerian equivalent of an integral taken over a moving material volume of fluid. This is provided through the Reynolds transport theorem [9].

The conversion formula differs slightly according to whether the control volume is fixed, moving, or deformable. To derive the formula, let B be any property of the fluid (mass, momentum, energy, etc.) and let $b = dB/dm$ be the intensive value of B (amount of B per unit mass) in any small element of the fluid.

For the arbitrary moving and deformable control volume shown in Fig. 3.1, the instantaneous total change of B in the material volume (MV) is equal to the instantaneous total change of B within the control volume (V) plus the net flow of B into and out of the control volume through its control surface (S). Let ρ denotes the density of the fluid, $\mathbf{n}$ the outward normal to the control volume surface, $\mathbf{v}(t, \mathbf{x})$ the velocity of the fluid, $\mathbf{v}_s(t, \mathbf{x})$ the velocity of the deforming control volume surface, and $\mathbf{v}_r(t, \mathbf{x})$ the relative velocity by which the fluid enters/leaves the control volume [i.e., $\mathbf{v}_r = \mathbf{v}(t, \mathbf{x}) - \mathbf{v}_s(t, \mathbf{x})$], then the Reynolds transport theorem gives

$$\left(\frac{dB}{dt}\right)_{MV} = \frac{d}{dt}\left(\int_{V(t)} b\rho dV\right) + \int_{S(t)} b\rho \mathbf{v}_r \cdot \mathbf{n}\, dS \tag{3.4}$$

For a fixed control volume, $\mathbf{v}_s = 0$ and the geometry is independent of time implying that the time derivative term on the right hand side of Eq. (3.4) can be written using Leibniz rule as

$$\frac{d}{dt}\left(\int_V b\rho\, dV\right) = \int_V \frac{\partial}{\partial t}(b\rho) dV \tag{3.5}$$

Therefore Eq. (3.4) simplifies to

$$\left(\frac{dB}{dt}\right)_{MV} = \int_V \frac{\partial}{\partial t}(b\rho) dV + \int_S b\rho \mathbf{v} \cdot \mathbf{n}\, dS \tag{3.6}$$

Applying the divergence theorem to transform the surface integral into a volume integral, Eq. (3.6) becomes

$$\left(\frac{dB}{dt}\right)_{MV} = \int_V \left[\frac{\partial}{\partial t}(\rho b) + \nabla \cdot (\rho \mathbf{v} b)\right] dV \tag{3.7}$$

An alternative form of Eq. (3.7) can be obtained by expanding the second term in the square bracket and using the substantial derivative to get

$$\left(\frac{dB}{dt}\right)_{MV} = \int_V \left[\frac{D}{Dt}(\rho b) + \rho b \nabla \cdot \mathbf{v}\right] dV \tag{3.8}$$

Equation (3.7) or (3.8) can be used to derive the Eulerian form of the conservation laws in fixed regions.

3.4 Conservation of Mass (Continuity Equation)

The principle of conservation of mass [6, 10] indicates that in the absence of mass sources and sinks, a region will conserve its mass on a local level.

Considering the material volume of fluid shown in Fig. 3.3 of mass m, density ρ, and velocity $\mathbf{v}$, conservation of mass in material (Lagrangian) coordinate system can be written as

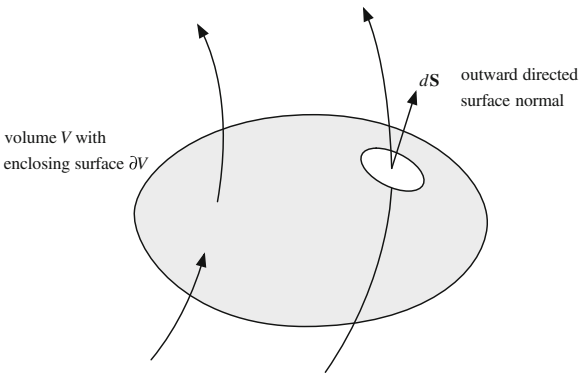


Fig. 3.3 Conservation of mass for a material volume of a fluid of mass m

$$\left(\frac{dm}{dt}\right)_{MV} = 0 \tag{3.9}$$

For $B = m$ the corresponding intensive quantity is $b = 1$, and based on Eq. (3.8) the equivalent expression of mass conservation in an Eulerian coordinate system is

$$\int_V \left[\frac{D\rho}{Dt} + \rho\nabla \cdot \mathbf{v}\right] dV = 0 \tag{3.10}$$

For the integral given in Eq. (3.10) to be true for any control volume V, the integrand should be equal to zero, giving the differential form of the mass conservation or continuity equation as

$$\frac{D\rho}{Dt} + \rho\nabla \cdot \mathbf{v} = 0 \tag{3.11}$$

The flux form of the continuity equation can be derived using Eq. (3.7) and leading to

$$\int_V \left(\frac{\partial\rho}{\partial t} + \nabla \cdot [\rho\mathbf{v}]\right) dV = 0 \tag{3.12}$$

Again for the integral in Eq. (3.12) to be true for any control volume V, the integrand should be equal to zero, giving the flux form of the mass conservation or continuity equation as

$$\frac{\partial\rho}{\partial t} + \nabla \cdot [\rho\mathbf{v}] = 0 \tag{3.13}$$

In the absence of any significant absolute pressure or temperature changes, it is acceptable to assume that the flow is incompressible; that is, the pressure changes

do not have significant effects on density. This is almost invariably the case in liquids, and is a good approximation in gases at speeds much less than that of sound. (Note that sound waves are compressible phenomena.) The most important consequence in fluid dynamics is that the mass conservation (continuity) equation can no longer be used to compute the density.

The incompressibility condition indicates that ρ does not change with the flow, which mathematically can be expressed as $D\rho/Dt = 0$. Using the mass conservation equation given by Eq. (3.11), this is equivalent to saying that the continuity equation for incompressible flow is given by

$$\nabla \cdot \mathbf{v} = 0 \tag{3.14}$$

or in integral form as

$$\int_S (\mathbf{v} \cdot \mathbf{n}) dS = 0 \tag{3.15}$$

Equation (3.15) states that for incompressible flows the net flow across any control volume is zero, i.e., "flow out" = "flow in".

Note also that $D\rho/Dt = 0$ does not imply that ρ is the same everywhere (although this happens to be the case in many hydraulic applications), but that ρ does not change along a streamline. To be more accurate, the incompressibility approximation means that each fluid element keeps its original density as it moves. In practice, density differences are commonly encountered in water due to variation in salt concentration and in air due to temperature differences resulting in important buoyancy forces.

3.5 Conservation of Linear Momentum

The principle of conservation of linear momentum [6, 10] indicates that in the absence of any external force acting on a body, the body retains its total momentum, i.e., the product of its mass and velocity vector. Since momentum is a vector quantity, its components in any direction will also be conserved.

For the material volume of a substance, Newton's Second Law of motion asserts that the momentum of this specific volume can change only in the presence of a net force acting on it, which could include both surface forces and body forces. Therefore, by considering the material volume of fluid shown in Fig. 3.4 of mass m, density ρ, and velocity $\mathbf{v}$, Newton's law in Lagrangian coordinates can be written as

$$\left(\frac{d(m\mathbf{v})}{dt}\right)_{MV} = \left(\int_V \mathbf{f}\, dV\right)_{MV} \tag{3.16}$$

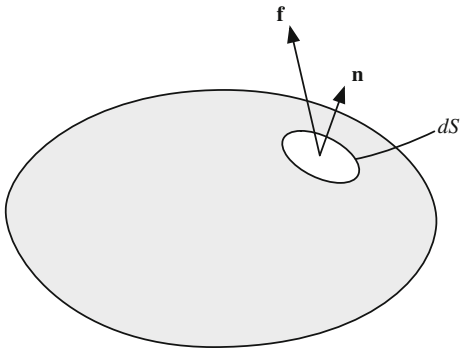


Fig. 3.4 Conservation of linear momentum for a material volume of a fluid of mass m

where $\mathbf{f}$ is the external force per unit volume acting on the material volume. The term on the right hand side of Eq. (3.16) is a volume integral over material coordinates performed over the volume occupied instantaneously by the moving fluid, thus

$$\left(\int_V \mathbf{f}\, dV\right)_{MV} = \int_V \mathbf{f}\, dV \tag{3.17}$$

The equivalent expression of Eq. (3.16) in Eulerian coordinates can be written in two different ways known as the conservative and non-conservative forms.

3.5.1 Non-Conservative Form

Noticing that in this case $b = \mathbf{v}$, the non-conservative form is obtained by using Eq. (3.8) in the derivation yielding

$$\int_V \left[\frac{D}{Dt}[\rho\mathbf{v}] + [\rho\mathbf{v}\nabla\cdot\mathbf{v}] - \mathbf{f}\right] dV = 0 \tag{3.18}$$

Again for the integral to be zero over any control volume, the integrand has to be zero. Thus,

$$\frac{D}{Dt}[\rho\mathbf{v}] + [\rho\mathbf{v}\nabla\cdot\mathbf{v}] = \mathbf{f} \tag{3.19}$$

Expanding the material derivative of the momentum term and regrouping, the non-conservative form is obtained as

$$\rho\frac{D\mathbf{v}}{Dt} + \mathbf{v}\underbrace{\left(\frac{D\rho}{Dt} + \rho\nabla\cdot\mathbf{v}\right)}_{Continuity} = \mathbf{f} \tag{3.20}$$

Applying the continuity constraint and expanding the material derivative, the non-conservative form of the momentum equation reduces to

$$\rho\left[\frac{\partial \mathbf{v}}{\partial t}+(\mathbf{v}\cdot\nabla)\mathbf{v}\right]=\mathbf{f} \tag{3.21}$$

3.5.2 Conservative Form

The conservative (or flux) version is obtained by applying the form of the Reynolds transport theorem given by Eq. (3.7) and is written as

$$\int_V\left[\frac{\partial}{\partial t}[\rho\mathbf{v}]+\nabla\cdot\{\rho\mathbf{v}\mathbf{v}\}-\mathbf{f}\right]dV=0 \tag{3.22}$$

By setting the integrand to zero for the integral to be zero for any volume V, the conservative form of the momentum equation is obtained as

$$\frac{\partial}{\partial t}[\rho\mathbf{v}]+\nabla\cdot\{\rho\mathbf{v}\mathbf{v}\}=\mathbf{f} \tag{3.23}$$

where $\rho\mathbf{v}\mathbf{v}$ is the dyadic product, described in Chap. 2, which is a special case of tensor product with its divergence being a vector.

Both forms will be used in this book for better describing the discretization concepts and for showing actual implementation details. In the derivations to follow the conservative form will be used. The non-conservative form can be easily obtained from the conservative form at any step by invoking the continuity constraint as explained above.

The full form of the momentum equation is obtained once the external surface and body forces acting on the control volume are specified. The force $\mathbf{f}$ is split into two parts one denoted by $\mathbf{f}_s$ representing the surface forces and the second by $\mathbf{f}_b$ representing the body forces such that

$$\mathbf{f}=\mathbf{f}_s+\mathbf{f}_b \tag{3.24}$$

The details of these forces are given next.

3.5.3 Surface Forces

For the arbitrary macroscopic volume element depicted in Fig. 3.4, the forces acting on its surface are due to pressure and viscous stresses which can be expressed in

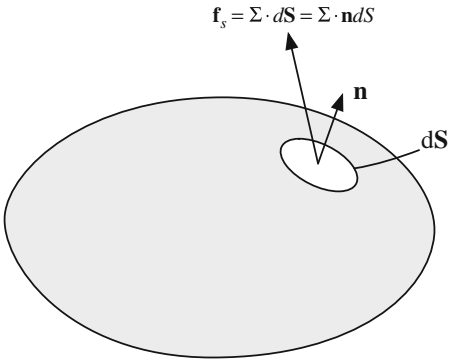


Fig. 3.5 The surface forces acting on a differential surface element expressed in terms of the stress tensor

term of the total stress tensor $\boldsymbol{\Sigma}$, as shown in Fig. 3.5. In general there are nine components of stress at any given point; one normal component and two shear components (parallel to the surface that receives the stress) in each coordinate plane. Thus in Cartesian coordinates the stress tensor is given by

$$\Sigma = \begin{pmatrix} \Sigma_{xx} & \Sigma_{xy} & \Sigma_{xz} \\ \Sigma_{yx} & \Sigma_{yy} & \Sigma_{yz} \\ \Sigma_{zx} & \Sigma_{zy} & \Sigma_{zz} \end{pmatrix} \tag{3.25}$$

where terms of the form Σ_{ii} represent normal stresses and Σ_{ij} shear stresses. A normal stress can be either a compression, if $\Sigma_{ii} \leq 0$, or a tension, if $\Sigma_{ii} \geq 0$. The most important compressive normal stress is usually due to pressure rather than to viscous effects. The component Σ_{ij} represents the stress acting on face i in the j direction with the direction of face i being positive if the outward normal to the face is in the positive direction.

In practice the stress tensor is split into two terms such that

$$\boldsymbol{\Sigma} = -\begin{pmatrix} p & 0 & 0 \\ 0 & p & 0 \\ 0 & 0 & p \end{pmatrix} + \begin{pmatrix} \overbrace{\boldsymbol{\Sigma}_{xx}+p}^{\tau_{xx}} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \overbrace{\boldsymbol{\Sigma}_{yy}+p}^{\tau_{yy}} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \overbrace{\boldsymbol{\Sigma}_{zz}+p}^{\tau_{zz}} \end{pmatrix} = -p\mathbf{I} + \boldsymbol{\tau} \tag{3.26}$$

where $\mathbf{I}$ is the identity tensor of size (3×3), p is the pressure, and $\boldsymbol{\tau}$ is the deviatoric or viscous stress tensor. The pressure is the negative of the mean of the normal stresses and is given by

$$p = -\frac{1}{3}\left(\Sigma_{xx} + \Sigma_{yy} + \Sigma_{zz}\right) \tag{3.27}$$

The surface force acting on a differential surface element of area dS and orientation $\mathbf{n}$, as illustrated in Fig. 3.5, is $(\boldsymbol{\Sigma} \cdot \mathbf{n})dS$. Applying the divergence theorem, the total surface force acting on the control volume is given by

$$\int_V \mathbf{f}_s \, dV = \int_S \boldsymbol{\Sigma} \cdot \mathbf{n} \, dS = \int_V \nabla \cdot \boldsymbol{\Sigma} \, dV \Rightarrow \mathbf{f}_s = [\nabla \cdot \boldsymbol{\Sigma}] = -\nabla p + [\nabla \cdot \boldsymbol{\tau}] \quad (3.28)$$

3.5.4 Body Forces

Body forces, which are presented as forces per unit volume, may also arise due to a variety of effects. There are plenty of examples, but the predominant ones are given next.

3.5.4.1 Gravitational Forces

The force representing the weight of the material volume per unit volume in the presence of a gravitational field is denoted by gravitational force (Fig. 3.6) and given by

$$\mathbf{f}_b = \rho \mathbf{g} \quad (3.29)$$

where $\mathbf{g}$ is the gravitational acceleration vector.

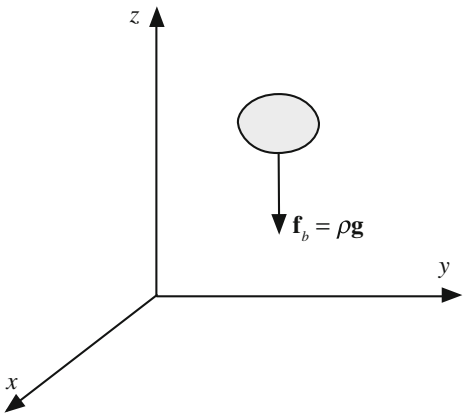


Fig. 3.6 Body forces acting on a differential element

3.5.4.2 System Rotation

When solving fluid flow problems in a rotating frame of reference, the forces arising as a result of the rigid body rotation of the reference frame should be accounted for. These can be viewed as body forces of the form

$$\mathbf{f}_b = \underbrace{-2\rho[\varpi \times \mathbf{v}]}_{Coriolis\,forces} \underbrace{-\rho[\varpi \times [\varpi \times \mathbf{r}]]}_{Centrifugal\,forces} \tag{3.30}$$

where ϖ is the angular velocity of the rotating reference frame and $\mathbf{r}$ is the position vector (Fig. 3.7). Note that gravitational and centrifugal forces are dependent on position but not on velocity. Thus they can be absorbed into a modified pressure and hence effectively ignored as a separate entity unless they appear in boundary conditions. Coriolis forces however have to be treated explicitly. Other forces, such as magnetic and electric, may be added depending on the particular situation. Due to the many possible types of body forces, no specific type will be adopted in the equations to follow and the generic $\mathbf{f}_b$ force is retained.

Substituting the external force $\mathbf{f}$ in Eq. (3.23) by its equivalent expression, the general conservative form of the momentum equation is obtained as

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = -\nabla p + [\nabla \cdot \tau] + \mathbf{f}_b \tag{3.31}$$

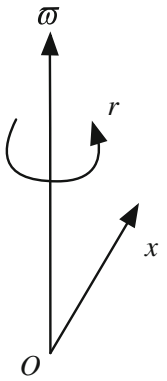


Fig. 3.7 Body forces due to a rigid body rotation in a rotating frame of reference

3.5.5 Stress Tensor and the Momentum Equation for Newtonian Fluids

To proceed further with the momentum equation, the type of fluid should be known in order to relate the stress tensor $\boldsymbol{\tau}$ to the flow variables. For a Newtonian fluid, the stress tensor is a linear function of the strain rate [2] and is given by

$$\boldsymbol{\tau} = \mu\left\{\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right\} + \lambda(\nabla\cdot\mathbf{v})\mathbf{I} \tag{3.32}$$

where μ is the molecular viscosity coefficient, λ the bulk viscosity coefficient usually set equal to $-(2/3)\mu\,(\lambda = -(2/3)\mu)$, the superscript T refers to the transpose of $\nabla\mathbf{v}$, and $\mathbf{I}$ is the unit or identity tensor of size (3×3) defined as

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.33}$$

The expanded form of the stress tensor in a three-dimensional Cartesian coordinate system can be written as

$$\boldsymbol{\tau} = \begin{bmatrix} 2\mu\dfrac{\partial u}{\partial x} + \lambda\nabla\cdot\mathbf{v} & \mu\left(\dfrac{\partial v}{\partial x} + \dfrac{\partial u}{\partial y}\right) & \mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right) \\ \mu\left(\dfrac{\partial v}{\partial x} + \dfrac{\partial u}{\partial y}\right) & 2\mu\dfrac{\partial v}{\partial y} + \lambda\nabla\cdot\mathbf{v} & \mu\left(\dfrac{\partial w}{\partial y} + \dfrac{\partial v}{\partial z}\right) \\ \mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right) & \mu\left(\dfrac{\partial w}{\partial y} + \dfrac{\partial v}{\partial z}\right) & 2\mu\dfrac{\partial w}{\partial z} + \lambda\nabla\cdot\mathbf{v} \end{bmatrix} \tag{3.34}$$

The divergence of the stress tensor is a vector that can be expressed as

$$\begin{aligned} [\nabla\cdot\boldsymbol{\tau}] &= \nabla\cdot\left[\mu\left(\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right)\right] + \nabla(\lambda\nabla\cdot\mathbf{v}) \\ &= \begin{bmatrix} \dfrac{\partial}{\partial x}\left[2\mu\dfrac{\partial u}{\partial x} + \lambda\nabla\cdot\mathbf{v}\right] + \dfrac{\partial}{\partial y}\left[\mu\left(\dfrac{\partial v}{\partial x} + \dfrac{\partial u}{\partial y}\right)\right] + \dfrac{\partial}{\partial z}\left[\mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right)\right] \\ \dfrac{\partial}{\partial x}\left[\mu\left(\dfrac{\partial v}{\partial x} + \dfrac{\partial u}{\partial y}\right)\right] + \dfrac{\partial}{\partial y}\left[2\mu\dfrac{\partial v}{\partial y} + \lambda\nabla\cdot\mathbf{v}\right] + \dfrac{\partial}{\partial z}\left[\mu\left(\dfrac{\partial w}{\partial y} + \dfrac{\partial v}{\partial z}\right)\right] \\ \dfrac{\partial}{\partial x}\left[\mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right)\right] + \dfrac{\partial}{\partial y}\left[\mu\left(\dfrac{\partial w}{\partial y} + \dfrac{\partial v}{\partial z}\right)\right] + \dfrac{\partial}{\partial z}\left[2\mu\dfrac{\partial w}{\partial z} + \lambda\nabla\cdot\mathbf{v}\right] \end{bmatrix} \end{aligned} \tag{3.35}$$

Substituting into Eq. (3.31), the final conservative form of the momentum equation for Newtonian fluids becomes

$$\frac{\partial}{\partial t}[\rho\mathbf{v}] + \nabla\cdot\{\rho\mathbf{v}\mathbf{v}\} = -\nabla p + \nabla\cdot\left\{\mu\left[\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right]\right\} + \nabla(\lambda\nabla\cdot\mathbf{v}) + \mathbf{f}_b \tag{3.36}$$

For later reference the momentum equation is expanded into

$$\frac{\partial}{\partial t}[\rho\mathbf{v}] + \nabla\cdot\{\rho\mathbf{v}\mathbf{v}\} = \nabla\cdot\{\mu\nabla\mathbf{v}\} - \nabla p + \underbrace{\nabla\cdot\left\{\mu(\nabla\mathbf{v})^{\mathrm{T}}\right\} + \nabla(\lambda\nabla\cdot\mathbf{v}) + \mathbf{f}_b}_{\mathbf{Q}^{\mathrm{v}}} \tag{3.37}$$

and rewritten as

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = \nabla \cdot \{\mu \nabla \mathbf{v}\} - \nabla p + \mathbf{Q}^{\mathbf{v}} \tag{3.38}$$

For incompressible flows, the divergence of the velocity vector is zero, i.e., $\nabla \cdot \mathbf{v} = 0$, and the momentum equation reduces to

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = -\nabla p + \nabla \cdot \left\{\mu \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^{\mathrm{T}}\right]\right\} + \mathbf{f}_b \tag{3.39}$$

If the viscosity is constant, the momentum equation can be further simplified. Taking just the first component of the vector equation [Eq. (3.35)], and assuming μ is constant, the following can be written:

$$\begin{aligned}
&\mu \frac{\partial}{\partial x}\left[2\frac{\partial u}{\partial x}\right] + \mu \frac{\partial}{\partial y}\left[\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)\right] + \mu \frac{\partial}{\partial z}\left[\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)\right] \\
&= \mu\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 v}{\partial yx} + \frac{\partial^2 u}{\partial z^2} + \frac{\partial^2 w}{\partial zx}\right] \\
&= \mu\left[\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial yx} + \frac{\partial^2 w}{\partial zx}\right] \\
&= \mu\left[\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right]
\end{aligned} \tag{3.40}$$

Substitution in Eq. (3.37) yields after simplification

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = -\nabla p + \mu \nabla^2 \mathbf{v} + \mathbf{f}_b \tag{3.41}$$

For inviscid flows the viscosity is zero and the momentum equation for incompressible and compressible inviscid flows becomes

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = -\nabla p + \mathbf{f}_b \tag{3.42}$$

3.6 Conservation of Energy

The conservation of energy [6, 10] is governed by the first law of thermodynamics which states that energy can be neither created nor destroyed during a process; it can only change from one form (mechanical, kinetic, chemical, etc.) into another. Consequently, the sum of all forms of energy in an isolated system remains constant.

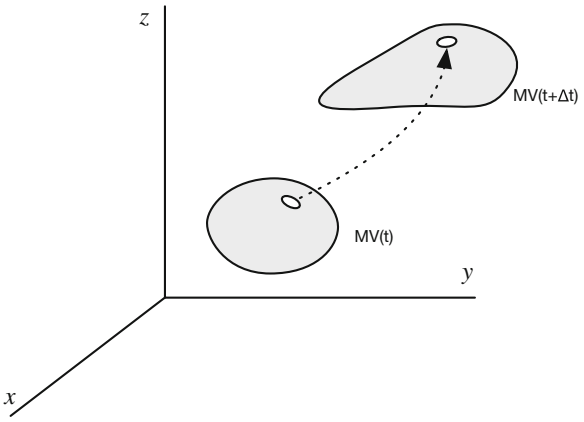


Fig. 3.8 A material volume moves with the particles it encloses

Considering the material volume shown in Fig. 3.8, of mass m, density ρ, and moving with a velocity $\mathbf{v}$. Defining the total energy E of the material volume at time t as the sum of its internal and kinetic energies, then E can be written as

$$E = m\left(\hat{u} + \frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right) \tag{3.43}$$

where $\hat{u}$ is the fluid specific internal energy (internal energy per unit mass). The first law of classical thermodynamics applied to the material volume states that the rate of change of the total energy of the material volume is equal to the rate of heat addition and work extraction through its boundaries. Mathematically this is given by

$$\left(\frac{dE}{dt}\right)_{MV} = \dot{Q} - \dot{W} \tag{3.44}$$

The adopted sign convention is such that heat added to the material volume and work done by the material volume are positive. To apply the Reynolds transport theorem on the material volume, B is set equal to E and b to e (the total energy per unit mass) such that

$$B = E \Rightarrow b = \frac{dE}{dm} = \hat{u} + \frac{1}{2}\mathbf{v}\cdot\mathbf{v} = e \tag{3.45}$$

The net rate of heat transferred to the material element $\dot{Q}$ is the sum of two components. The first component is the rate transferred across the surface of the element $\dot{Q}_S$ and the second generated/destroyed (e.g., due to a chemical reaction) within the material volume $\dot{Q}_V$. Moreover, the net rate of work done by the material volume $\dot{W}$ is due to the rate of work done by the surface forces $\dot{W}_S$ and the rate of work done by the body forces $\dot{W}_b$. Thus the first law can be written as

$$\left(\frac{dE}{dt}\right)_{MV} = \dot{Q}_V + \dot{Q}_S - \dot{W}_b - \dot{W}_S \tag{3.46}$$

By definition, work is due to a force acting through a distance and power is the rate at which work is done. Therefore the rate of work done by body and surface forces can be represented by

$$\begin{aligned} \dot{W}_b &= -\int_V (\mathbf{f}_b \cdot \mathbf{v}) dV \\ \dot{W}_S &= -\int_S (\mathbf{f}_S \cdot \mathbf{v}) dS \end{aligned} \tag{3.47}$$

The rate of work due to surface forces can be expanded by replacing $\mathbf{f}_S$ by its equivalent expression as given in Eq. (3.26) through (3.28). This leads to

$$\dot{W}_S = -\int_S [\boldsymbol{\Sigma} \cdot \mathbf{v}] \cdot \mathbf{n}\, dS = -\int_V \nabla \cdot [\boldsymbol{\Sigma} \cdot \mathbf{v}] dV = -\int_V \nabla \cdot [(-p\mathbf{I} + \boldsymbol{\tau}) \cdot \mathbf{v}] dV \tag{3.48}$$

After manipulation, $\dot{W}_S$ can be rewritten as

$$\dot{W}_S = -\int_V (-\nabla \cdot [p\mathbf{v}] + \nabla \cdot [\boldsymbol{\tau} \cdot \mathbf{v}]) dV \tag{3.49}$$

If $\dot{q}_V$ represents the rate of heat source or sink within the material volume per unit volume and $\dot{q}_S$ the rate of heat transfer per unit area across the surface area of the material element, then $\dot{Q}_V$ and $\dot{Q}_S$ can be written as

$$\dot{Q}_V = \int_V \dot{q}_V dV \quad \dot{Q}_S = -\int_S \dot{q}_s \cdot \mathbf{n}\, dS = -\int_V \nabla \cdot \dot{q}_s dV \tag{3.50}$$

Applying the Reynolds transport theorem and substituting the rate of work and heat terms by their equivalent expressions, Eq. (3.46) becomes

$$\begin{aligned} \left(\frac{dE}{dt}\right)_{MV} &= \int_V \left[\frac{\partial}{\partial t}(\rho e) + \nabla \cdot [\rho \mathbf{v} e]\right] dV \\ &= -\int_V \nabla \cdot \dot{q}_s dV + \int_V (-\nabla \cdot [p\mathbf{v}] + \nabla \cdot [\boldsymbol{\tau} \cdot \mathbf{v}]) dV + \int_V (\mathbf{f}_b \cdot \mathbf{v}) dV + \int_V \dot{q}_V dV \end{aligned} \tag{3.51}$$

Collecting terms together, the above equation is transformed to

$$\int_V \left[\frac{\partial}{\partial t}(\rho e) + \nabla \cdot [\rho \mathbf{v} e] + \nabla \cdot \dot{q}_s + \nabla \cdot [p\mathbf{v}] - \nabla \cdot [\boldsymbol{\tau} \cdot \mathbf{v}] - \mathbf{f}_b \cdot \mathbf{v} - \dot{q}_V\right] dV = 0 \tag{3.52}$$

For the volume integral in Eq. (3.52) to be true for any control volume, the integrand has to be zero. Thus,

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot [\rho \mathbf{v} e] = -\nabla \cdot \dot{q}_s - \nabla \cdot [p\mathbf{v}] + \nabla \cdot [\boldsymbol{\tau} \cdot \mathbf{v}] + \mathbf{f}_b \cdot \mathbf{v} + \dot{q}_V \tag{3.53}$$

which represents the mathematical description of energy conservation or simply the energy equation written in terms of specific total energy. The energy equation may also be written in terms of specific internal energy, specific static enthalpy (or simply specific enthalpy), specific total enthalpy, and under special conditions in terms of temperature.

3.6.1 Conservation of Energy in Terms of Specific Internal Energy

To rewrite the energy equation [Eq. (3.53)] in terms of specific internal energy, the dot product of the momentum equation [Eq. (3.23)] with the velocity vector is performed resulting in

$$\left[\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\}\right] \cdot \mathbf{v} = \mathbf{f} \cdot \mathbf{v} \tag{3.54}$$

After some manipulations Eq. (3.54) becomes

$$\frac{\partial}{\partial t}(\rho \mathbf{v} \cdot \mathbf{v}) - \rho \mathbf{v} \cdot \frac{\partial \mathbf{v}}{\partial t} + \nabla \cdot [\rho(\mathbf{v} \cdot \mathbf{v})\mathbf{v}] - \rho \mathbf{v} \cdot [(\mathbf{v} \cdot \nabla)\mathbf{v}] = \mathbf{f} \cdot \mathbf{v} \tag{3.55}$$

Rearranging and collecting terms the following is obtained:

$$\frac{\partial}{\partial t}(\rho \mathbf{v} \cdot \mathbf{v}) + \nabla \cdot [\rho(\mathbf{v} \cdot \mathbf{v})\mathbf{v}] - \mathbf{v} \cdot \underbrace{\rho\left[\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v}\right]}_{\substack{= \mathbf{f} \\ Eq.\,(3.21)}} = \mathbf{f} \cdot \mathbf{v} \tag{3.56}$$

Noticing that the third term on the left side is $\mathbf{v} \cdot \mathbf{f}$ and replacing $\mathbf{f}$ by its equivalent expression, an equation for the flow kinetic energy is obtained as

$$\frac{\partial}{\partial t}\left(\rho\frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right)+\nabla\cdot\left[\rho\left(\frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right)\mathbf{v}\right]=-\mathbf{v}\cdot\nabla p+\mathbf{v}\cdot[\nabla\cdot\boldsymbol{\tau}]+\mathbf{f}_b\cdot\mathbf{v} \tag{3.57}$$

This equation can be modified and rewritten in the following form:

$$\begin{aligned}&\frac{\partial}{\partial t}\left(\rho\frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right)+\nabla\cdot\left[\rho\left(\frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right)\mathbf{v}\right]\\&\quad=-\nabla\cdot[p\mathbf{v}]+p\nabla\cdot\mathbf{v}+\nabla\cdot[\boldsymbol{\tau}\cdot\mathbf{v}]-(\boldsymbol{\tau}:\nabla\mathbf{v})+\mathbf{f}_b\cdot\mathbf{v}\end{aligned} \tag{3.58}$$

Subtracting Eq. (3.58) from Eq. (3.53), the energy equation with specific internal energy as its main variable is obtained as

$$\frac{\partial}{\partial t}(\rho\hat{u})+\nabla\cdot[\rho\mathbf{v}\hat{u}]=-\nabla\cdot\dot{q}_s-p\nabla\cdot\mathbf{v}+(\boldsymbol{\tau}:\nabla\mathbf{v})+\dot{q}_V \tag{3.59}$$

3.6.2 Conservation of Energy in Terms of Specific Enthalpy

Rewriting the energy equation in terms of specific enthalpy is straightforward and follows directly from its definition according to which the specific internal energy and specific enthalpy are related by

$$\hat{u}=\hat{h}-\frac{p}{\rho} \tag{3.60}$$

Substituting $\left(\hat{h}-p/\rho\right)$ for $\hat{u}$ in Eq. (3.59) and performing some algebraic manipulations, the energy equation in terms of specific enthalpy evolves as

$$\frac{\partial}{\partial t}\left(\rho\hat{h}\right)+\nabla\cdot\left[\rho\mathbf{v}\hat{\mathrm{h}}\right]=-\nabla\cdot\dot{q}_s+\frac{Dp}{Dt}+(\boldsymbol{\tau}:\nabla\mathbf{v})+\dot{q}_V \tag{3.61}$$

3.6.3 Conservation of Energy in Terms of Specific Total Enthalpy

The energy equation in terms of specific total enthalpy can be derived by expressing e in terms of $\hat{h}_0$ to get

$$e=\hat{u}+\frac{1}{2}\mathbf{v}\cdot\mathbf{v}=\hat{h}-\frac{p}{\rho}+\frac{1}{2}\mathbf{v}\cdot\mathbf{v}=\hat{h}_0-\frac{p}{\rho} \tag{3.62}$$

Then by substituting $\left(\hat{h} - p/\rho\right)$ for e in Eq. (3.53) and performing some algebraic manipulations, the energy equation in terms of specific total enthalpy is obtained as

$$\frac{\partial}{\partial t}\left(\rho\hat{h}_0\right) + \nabla \cdot \left[\rho \mathbf{v}\hat{h}_0\right] = -\nabla \cdot \dot{q}_s + \frac{\partial p}{\partial t} + \nabla \cdot [\boldsymbol{\tau} \cdot \mathbf{v}] + \mathbf{f}_b \cdot \mathbf{v} + \dot{q}_V \tag{3.63}$$

All forms of the energy equation presented so far are general and applicable to Newtonian and non-Newtonian fluids. The only limitation is that they are applicable to a fixed control volume.

3.6.4 Conservation of Energy in Terms of Temperature

To be able to write the energy equation with temperature as the main variable some constraints have to be imposed. Assuming $\hat{h}$ to be a function of p and T, the fluid is expected to be Newtonian. Therefore the derivations to follow are applicable to Newtonian fluids only. If $\hat{h} = \hat{h}(p, T)$, then $d\hat{h}$ can be written as

$$d\hat{h} = \left(\frac{\partial \hat{h}}{\partial T}\right)_p dT + \left(\frac{\partial \hat{h}}{\partial p}\right)_T dp \tag{3.64}$$

Using the following ordinary equilibrium thermodynamics relation:

$$\left(\frac{\partial \hat{h}}{\partial p}\right)_T = \hat{V} - T\left(\frac{\partial \hat{V}}{\partial T}\right)_p \tag{3.65}$$

where $\hat{V}$ is the specific volume, the expression for $d\hat{h}$ can be modified to

$$d\hat{h} = c_p dT + \left[\hat{V} - T\left(\frac{\partial \hat{V}}{\partial T}\right)_p\right] dp \tag{3.66}$$

The left side of the specific enthalpy [Eq. (3.61)], with $d\hat{h}$ given by Eq. (3.66), can be rewritten in terms of T as

$$\begin{aligned}
\frac{\partial}{\partial t}\left(\rho\hat{h}\right) + \nabla \cdot \left[\rho \mathbf{v}\hat{h}\right] = \rho\frac{D\hat{h}}{Dt} &= \rho c_p \frac{DT}{Dt} + \rho\left[\hat{V} - T\left(\frac{\partial \hat{V}}{\partial T}\right)_p\right]\frac{DP}{Dt} \\
&= \rho c_p \frac{DT}{Dt} + \rho\left[\frac{1}{\rho} - T\left(\frac{\partial(1/\rho)}{\partial T}\right)_p\right]\frac{DP}{Dt} \\
&= \rho c_p \frac{DT}{Dt} + \left[1 + \left(\frac{\partial(Ln\rho)}{\partial(LnT)}\right)_p\right]\frac{DP}{Dt}
\end{aligned} \tag{3.67}$$

Substituting Eq. (3.67) into Eq. (3.61) gives the energy equation with T as its main variable as

$$\rho c_p \frac{DT}{Dt} = -\nabla \cdot \dot{q}_s - \left(\frac{\partial(Ln\rho)}{\partial(LnT)}\right)_p \frac{Dp}{Dt} + (\boldsymbol{\tau} : \nabla \mathbf{v}) + \dot{q}_V \tag{3.68}$$

The above equation is equivalently given by

$$c_p \left[\frac{\partial}{\partial t}(\rho T) + \nabla \cdot [\rho \mathbf{v} T]\right] = -\nabla \cdot \dot{q}_s - \left(\frac{\partial(Ln\rho)}{\partial(LnT)}\right)_p \frac{Dp}{Dt} + (\boldsymbol{\tau} : \nabla \mathbf{v}) + \dot{q}_V \tag{3.69}$$

The heat flux $\dot{q}_S$ appearing in all forms of the energy equation represents heat transfer by diffusion, which is a phenomenon occurring at the molecular level and is governed by Fourier's law according to

$$\dot{q}_s = -[k \nabla T] \tag{3.70}$$

where k is the thermal conductivity of the substance. The above equation states that heat flows in the direction of temperature gradient and assumes that the material has no preferred direction for heat transfer with the same thermal conductivity in all directions, i.e., the medium is isotropic. However some solids are anisotropic for which Eq. (3.70) is replaced by

$$\dot{q}_s = -[\boldsymbol{\kappa} \cdot \nabla T] \tag{3.71}$$

where $\boldsymbol{\kappa}$ is a second order symmetric tensor called the thermal conductivity tensor. Consequently, the heat flux in anisotropic medium is not in the direction of the temperature gradient. In the derivations to follow the medium is assumed to be isotropic and Eq. (3.70) is applicable. Replacing $\dot{q}_s$ using Fourier's law, the energy equation, Eq. (3.69), becomes

$$c_p \left[\frac{\partial}{\partial t}(\rho T) + \nabla \cdot [\rho \mathbf{v} T]\right] = \nabla \cdot [k \nabla T] - \left(\frac{\partial(Ln\rho)}{\partial(LnT)}\right)_p \frac{Dp}{Dt} + (\boldsymbol{\tau} : \nabla \mathbf{v}) + \dot{q}_V \tag{3.72}$$

The expression for $(\boldsymbol{\tau} : \nabla \mathbf{v})$ in terms of the flow variables in a three-dimensional Cartesian coordinate system is given by

$$(\tau : \nabla \mathbf{v}) = \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)^2 + \mu \left(\begin{array}{l} 2\left(\frac{\partial u}{\partial x}\right)^2 + 2\left(\frac{\partial v}{\partial y}\right)^2 + 2\left(\frac{\partial w}{\partial z}\right)^2 \\ + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)^2 \end{array} \right) \tag{3.73}$$

Defining Ψ and Φ as

$$\Psi = \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)^2 \tag{3.74}$$

$$\Phi = 2\left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial z}\right)^2\right] + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)^2 \tag{3.75}$$

The energy equation in terms of temperature reduces to

$$c_p\left[\frac{\partial}{\partial t}(\rho T) + \nabla \cdot [\rho \mathbf{v} T]\right] = \nabla \cdot [k \nabla T] - \left(\frac{\partial (Ln\,\rho)}{\partial (Ln\,T)}\right)_p \frac{Dp}{Dt} + \lambda \Psi + \mu \Phi + \dot{q}_V \tag{3.76}$$

For later reference the energy equation is expanded to

$$\frac{\partial}{\partial t}(\rho c_p T) + \nabla \cdot [\rho c_p \mathbf{v} T] = \nabla \cdot [k \nabla T] + \underbrace{\rho T \frac{Dc_p}{Dt} - \left(\frac{\partial (Ln\,\rho)}{\partial (Ln\,T)}\right)_p \frac{Dp}{Dt} + \lambda \Psi + \mu \Phi + \dot{q}_V}_{Q^T} \tag{3.77}$$

and rewritten as

$$\frac{\partial}{\partial t}(\rho c_p T) + \nabla \cdot [\rho c_p \mathbf{v} T] = \nabla \cdot [k \nabla T] + Q^T \tag{3.78}$$

The energy equation is rarely solved in its full form and depending on the physical situation several simplified versions can be developed. The dissipation term Φ has negligible values except for large velocity gradients at supersonic speeds. Moreover, for incompressible fluids the continuity equation implies that $\Psi = 0$ and because the density is constant it follows that $(\partial(Ln\rho)/\partial(LnT)) = 0$. Therefore the energy equation [Eq. (3.77)] for incompressible fluid flow is simplified to

$$\frac{\partial}{\partial t}(\rho c_p T) + \nabla \cdot [\rho c_p \mathbf{v} T] = \nabla \cdot [k \nabla T] + \underbrace{\dot{q}_V + \rho T \frac{Dc_p}{Dt}}_{Q^T} \tag{3.79}$$

Equation (3.79) is also applicable for a fluid flowing in a constant pressure system. For the case of a solid, the density is constant, the velocity is zero, and if changes in temperature are not large then the thermal conductivity may be considered constant, in which case the energy equation becomes

$$\rho c_p \frac{\partial T}{\partial t} = k\nabla^2 T + \dot{q}_V \tag{3.80}$$

For ideal gases $(\partial(Ln\rho)/\partial(LnT)) = -1$ and the energy equation for compressible flow of ideal gases reduces to

$$c_p \left[\frac{\partial}{\partial t}(\rho T) + \nabla \cdot [\rho \mathbf{v} T]\right] = \nabla \cdot [k\nabla T] + \frac{Dp}{Dt} + \lambda\Psi + \mu\Phi + \dot{q}_V \tag{3.81}$$

If viscosity is neglected (i.e. the flow is inviscid), Eq. (3.81) is further simplified to

$$c_p \left[\frac{\partial}{\partial t}(\rho T) + \nabla \cdot [\rho \mathbf{v} T]\right] = \nabla \cdot [k\nabla T] + \frac{Dp}{Dt} + \dot{q}_V \tag{3.82}$$

3.7 General Conservation Equation

From the above, the governing equations describing the conservation of mass, momentum, and energy are written in terms of *specific* quantities or *intensive* properties, i.e., quantities expressed on a per unit mass basis. The momentum equation, for example, expressed the principle of conservation of linear momentum in terms of the momentum per unit mass, i.e., velocity. The same type of conservation equation may be applied to any intensive property ϕ, e.g., concentration of salt in a solution or the mass fraction of a chemical species. The variation of ϕ in the control volume over time can be expressed as a balance equation of the form

Change of of ϕ over time Δt within the material volume (MV)	=	Surface flux of ϕ over time Δt across the control volume	+	Source/sink of ϕ over time Δt within control volume
Term I		Term II		Term III

For the fixed control volume shown in Fig. 3.9, the change of ϕ over time within the material volume can be written using the Reynolds transport theorem as

$$\text{Term I} = \frac{d}{dt}\left(\int_{MV} (\rho\phi) dV\right) = \int_V \left[\frac{\partial}{\partial t}(\rho\phi) + \nabla \cdot (\rho \mathbf{v}\phi)\right] dV \tag{3.83}$$

where ρ is the fluid density and V the volume of the control volume of surface area S. The term $\rho\mathbf{v}\phi$ represents the transport of ϕ by the flow field and is denoted by the convective flux, i.e.,

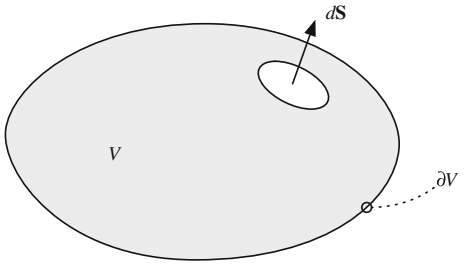


Fig. 3.9 Arbitrary fixed control volume

$$\mathbf{J}^{\phi}_{convection} = \rho \mathbf{v} \phi \tag{3.84}$$

The second term represents variation of ϕ due to physical phenomena occurring across the control volume surface. For the physical phenomena of interest in this book, the mechanism causing the influx/out flux of ϕ is due to diffusion, which is produced by molecular collision and is designated by $\mathbf{J}^{\phi}_{diffusion}$. Denoting the diffusion coefficient of ϕ by Γ^{ϕ}, the diffusion flux may be written as

$$\mathbf{J}^{\phi}_{diffusion} = -\Gamma^{\phi} \nabla \phi \tag{3.85}$$

and Term II becomes

$$\text{Term II} = -\int_S \mathbf{J}^{\phi}_{diffusion} \cdot \mathbf{n}\, dS = -\int_V \nabla \cdot \mathbf{J}^{\phi}_{diffusion} dV = \int_V \nabla \cdot \left(\Gamma^{\phi} \nabla \phi\right) dV \tag{3.86}$$

where $\mathbf{n}$ is the outward unit vector normal to the surface and the negative sign is due to the adopted sign convention (i.e., inward flux is positive). Term III can be written as

$$\text{Term III} = \int_V Q^{\phi} dV \tag{3.87}$$

where Q^{ϕ} is the generation/destruction of ϕ within the control volume per unit volume, which is also called the source term. Thus the conservation equation can be expressed as

$$\int_V \left[\frac{\partial}{\partial t}(\rho \phi) + \nabla \cdot (\rho \phi \mathbf{v})\right] dV = \int_V \nabla \cdot \left(\Gamma^{\phi} \nabla \phi\right) dV + \int_V Q^{\phi} dV \tag{3.88}$$

which can be rearranged into

$$\int_V \left[\frac{\partial}{\partial t}(\rho\phi) + \nabla\cdot(\rho\mathbf{v}\phi) - \nabla\cdot\left(\Gamma^\phi\nabla\phi\right) - Q^\phi\right]dV = 0 \tag{3.89}$$

For the integral to be zero for any control volume, the integrand has to be zero giving the conservation equation in differential form as

$$\frac{\partial}{\partial t}(\rho\phi) + \nabla\cdot(\rho\mathbf{v}\phi) - \nabla\cdot\left(\Gamma^\phi\nabla\phi\right) - Q^\phi = 0 \tag{3.90}$$

For later reference the above equation may be rewritten as

$$\frac{\partial}{\partial t}(\rho\phi) + \nabla\cdot\mathbf{J}^\phi - Q^\phi = 0 \tag{3.91}$$

where the total flux $\mathbf{J}^\phi$ is the sum of the convective and diffusive fluxes given by

$$\mathbf{J}^\phi = \mathbf{J}^{\phi,C} + \mathbf{J}^{\phi,D} = \rho\mathbf{v}\phi - \Gamma^\phi\nabla\phi \tag{3.92}$$

The final form of the general conservation equation, Eq. (3.90), for the transport of a property ϕ is expressed as

$$\underbrace{\frac{\partial}{\partial t}(\rho\phi)}_{\text{unsteady term}} + \underbrace{\nabla\cdot(\rho\mathbf{v}\phi)}_{\text{convection term}} = \underbrace{\nabla\cdot\left(\Gamma^\phi\nabla\phi\right)}_{\text{diffusion term}} + \underbrace{Q^\phi}_{\text{source term}} \tag{3.93}$$

By comparing Eq. (3.93) to the various conservation equations derived earlier, it can be easily inferred that by assigning the right values for ϕ, Γ^ϕ, and Q^ϕ, Eq. (3.93) is a general equation that can represent any of the conservation equations. This is a very important observation that will reduce the necessary developments of the numerical techniques in the coming chapters by concentrating on the general equation [Eq. (3.93)] rather than the individual conservation equations.

3.8 Non-dimensionalization Procedure

The differential equations representing conservation laws are rarely solved using dimensional variables. The common practice is to write these equations in a non-dimensional form using dimensionless quantities that are obtained through the use of proper characteristic scales. The use of non-dimensional variables has several advantages. It allows reducing the number of appropriate parameters for the problem considered and helps revealing the relative magnitude of the various terms in the conservation equation and consequently those that can be neglected.

This simplifies the equation to be solved and leaves only terms of similar order of magnitude, which results in better numerical accuracy. In addition, the generated solution will be applicable to all dynamically similar problems.

A dimensional variable is transformed into a non-dimensional one by dividing the variable by a quantity (composed of one or more physical properties) that has the same dimension as the original variable. For example spatial coordinates can be divided by a characteristic length; velocity can be divided by a characteristic velocity or a combination of quantities $\left(\mu_{ref}/\rho_{ref}l_{ref}\right)$ that have collectively the same units as velocity (m/s); pressure is usually divided by a reference dynamic pressure $\left(\rho_{ref}\left|\mathbf{v}_{ref}\right|^2\right)$; time can be divided by the ratio of a characteristic length to a reference velocity $\left(l_{ref}/\left|\mathbf{v}_{ref}\right|\right)$, and so on. The best way to fully understand how to write equations in non-dimensional form is through an example. For that purpose, an incompressible viscous flow with constant viscosity and thermal conductivity, and with body forces acting in the y-direction (i.e., the gravitational acceleration is given by $\mathbf{g} = (0, -g, 0)$) is considered. The equations governing conservation of mass, momentum, and energy in a three-dimensional Cartesian coordinate system with no heat generation are given by

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{3.94}$$

$$\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho uu) + \frac{\partial}{\partial y}(\rho vu) + \frac{\partial}{\partial z}(\rho wu) = -\frac{\partial p}{\partial x} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \tag{3.95}$$

$$\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho uv) + \frac{\partial}{\partial y}(\rho vv) + \frac{\partial}{\partial z}(\rho wv) = -\frac{\partial p}{\partial y} + \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) - \rho g \tag{3.96}$$

$$\frac{\partial}{\partial t}(\rho w) + \frac{\partial}{\partial x}(\rho uw) + \frac{\partial}{\partial y}(\rho vw) + \frac{\partial}{\partial z}(\rho ww) = -\frac{\partial p}{\partial z} + \mu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) \tag{3.97}$$

$$c_p\left[\frac{\partial}{\partial t}(\rho T) + \frac{\partial}{\partial x}(\rho uT) + \frac{\partial}{\partial y}(\rho vT) + \frac{\partial}{\partial z}(\rho wT)\right] = k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{3.98}$$

When body forces are of negligible magnitude in comparison with other forces, the term $\rho\mathbf{g}$ can be set to zero and removed from the equations. In such situation, the flow field is independent of the temperature field and solution for the velocity field can be established separately followed by the solution to the temperature field. However for a flow to exist the fluid should be forced through the domain.

Therefore the fluid should possess an inlet velocity. This velocity becomes an important parameter (the characteristic velocity) when writing the equations in non-dimensional forms and the heat transfer mechanism, if any, is said to occur by forced convection.

On the other hand, when a flow field is naturally established due to a temperature difference in the domain, body forces cannot be neglected. In such situations, variations in temperature cause variations in density (as mentioned earlier), which give rise to buoyancy forces that drive the flow. In this case the transfer of heat is stated to happen by natural convection. As the flow is initiated naturally, a characteristic velocity is not apparent and cannot be part of the dimensionless number since its scale is not known. Therefore in order to write the velocity in a non-dimensional form, a combination of physical quantities that has the same dimension as a velocity should be used. The following discussion assumes a natural convection problem.

If the difference in temperature $\Delta T = T - T_\infty$ (where T_∞ is a reference temperature between the minimum and maximum temperature in the domain, usually taken as the average value) is small such that terms of order ΔT^2 or higher can be neglected, then the value of density at any temperature T can be written as a function of its value at the reference temperature T_∞ using a truncated Taylor series expansion as

$$\rho = \rho|_{T=T_\infty} + \left.\frac{d\rho}{dT}\right|_{T=T_\infty} (T - T_\infty) \tag{3.99}$$

where terms of order ΔT^2 or higher are omitted. Introducing the coefficient of volume expansion β defined as

$$\beta = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p \tag{3.100}$$

the equation for density (or equation of state) becomes

$$\rho = \rho_\infty[1 - \beta(T - T_\infty)] \tag{3.101}$$

which is known in the literature by the Boussinesq approximation [11]. Using this expression for ρ in the body force term only and denoting the constant density value by ρ to simplify the notation, the y-momentum equation is transformed to

$$\begin{aligned} &\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho uv) + \frac{\partial}{\partial y}(\rho vv) + \frac{\partial}{\partial z}(\rho wv) \\ &\quad = -\frac{\partial}{\partial y}(p + \rho g y) + \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + \rho g \beta (T - T_\infty) \end{aligned} \tag{3.102}$$

This clearly shows that in solving natural convection problems the momentum and energy equations are coupled together necessitating a simultaneous solution of both equations.

The non-dimensional forms of the conservation equations are obtained by defining the following dimensionless parameters:

$$\begin{aligned}&\hat{x}=\frac{x}{L},\hat{y}=\frac{y}{L},\hat{z}=\frac{z}{L},\hat{u}=\frac{u}{\mu/(\rho L)},\hat{v}=\frac{v}{\mu/(\rho L)},\hat{w}=\frac{w}{\mu/(\rho L)}\\&\hat{t}=\frac{t}{\rho L^2/\mu},\hat{p}=\frac{p+\rho gy}{\mu^2/(\rho L^2)},\hat{T}=\frac{T-T_\infty}{T_{\max}-T_\infty}\end{aligned}\tag{3.103}$$

where L is a characteristic length, μ the dynamic viscosity of the fluid, T_{max} the maximum temperature in the domain, and the over $\wedge$ is used to designate non-dimensional quantities. The various expressions in the conservation equations are written in terms of the new variables as described next. The procedure is explained by considering a typical term from each category.

Typical term in the continuity equation:

$$\frac{\partial u}{\partial x}=\frac{\partial[\mu\hat{u}/(\rho L)]}{\partial(L\hat{x})}=\frac{\mu/(\rho L)}{L}\frac{\partial\hat{u}}{\partial\hat{x}}=\frac{\mu}{\rho L^2}\frac{\partial\hat{u}}{\partial\hat{x}}\tag{3.104}$$

Typical terms in the momentum equations:

$$\frac{\partial}{\partial t}(\rho u)=\frac{\partial(\mu\hat{u}/L)}{\partial(\rho L^2\hat{t}/\mu)}=\frac{\mu/L}{\rho L^2/\mu}\frac{\partial\hat{u}}{\partial\hat{t}}=\frac{\mu^2}{\rho L^3}\frac{\partial\hat{u}}{\partial\hat{t}}\tag{3.105}$$

$$\frac{\partial}{\partial t}(\rho uu)=\frac{\partial[\mu^2/(\rho L^2)\hat{u}\hat{u}]}{\partial(L\hat{x})}=\frac{\mu^2/(\rho L^2)}{L}\frac{\partial}{\partial\hat{x}}(\hat{u}\hat{u})=\frac{\mu^2}{\rho L^3}\frac{\partial}{\partial\hat{x}}(\hat{u}\hat{u})\tag{3.106}$$

$$\begin{aligned}\hat{p}=\frac{p+\rho gy}{\mu^2/(\rho L^2)}\Rightarrow\frac{\partial\hat{p}}{\partial\hat{x}}&=\frac{\partial\{(p+\rho gy)/[\mu^2/(\rho L^2)]\}}{\partial(x/L)}\\&=\frac{\rho L^3}{\mu^2}\frac{\partial p}{\partial x}\Rightarrow\frac{\partial p}{\partial x}=\frac{\mu^2}{\rho L^3}\frac{\partial\hat{p}}{\partial\hat{x}}\end{aligned}\tag{3.107}$$

$$\mu\frac{\partial^2 u}{\partial x^2}=\mu\frac{\partial^2[\mu\hat{u}/(\rho L)]}{\partial(L\hat{x})^2}=\mu\frac{\mu/(\rho L)}{L^2}\frac{\partial^2\hat{u}}{\partial\hat{x}^2}=\frac{\mu^2}{\rho L^3}\frac{\partial^2\hat{u}}{\partial\hat{x}^2}\tag{3.108}$$

$$\rho g\beta(T-T_\infty)=\rho g\beta(T_{\max}-T_\infty)\hat{T}=\rho g\beta(\Delta T)\hat{T}\tag{3.109}$$

Typical terms in the energy equation:

$$\frac{\partial}{\partial t}(\rho T)=\frac{\partial\left[\rho\left(T_\infty+\Delta T\hat{T}\right)\right]}{\partial(\rho L^2\hat{t}/\mu)}=\frac{\mu\Delta T}{L^2}\frac{\partial\hat{T}}{\partial\hat{t}}\tag{3.110}$$

$$\frac{\partial}{\partial x}(\rho u T) = \frac{\partial\left(\mu\hat{u}\left(T_\infty + \Delta T\hat{T}\right)/L\right)}{\partial(L\hat{x})} = \frac{\mu T_\infty}{L^2}\frac{\partial\hat{u}}{\partial\hat{x}} + \frac{\mu\Delta T}{L^2}\frac{\partial}{\partial\hat{x}}\left(\hat{u}\hat{T}\right) \tag{3.111}$$

$$k\frac{\partial^2 T}{\partial x^2} = k\frac{\partial^2\left(T_\infty + \Delta T\hat{T}\right)}{\partial(L\hat{x})^2} = \frac{k\Delta T}{L^2}\frac{\partial^2\hat{T}}{\partial\hat{x}^2} \tag{3.112}$$

Substituting terms by their equivalent expressions, the non-dimensional forms of the continuity, momentum, and energy equations are obtained as

$$\frac{\partial\hat{u}}{\partial\hat{x}} + \frac{\partial\hat{v}}{\partial\hat{y}} + \frac{\partial\hat{w}}{\partial\hat{z}} = 0 \tag{3.113}$$

$$\frac{\partial\hat{u}}{\partial\hat{t}} + \frac{\partial}{\partial\hat{x}}(\hat{u}\hat{u}) + \frac{\partial}{\partial\hat{y}}(\hat{v}\hat{u}) + \frac{\partial}{\partial\hat{z}}(\hat{w}\hat{u}) = -\frac{\partial\hat{p}}{\partial\hat{x}} + \left(\frac{\partial^2\hat{u}}{\partial\hat{x}^2} + \frac{\partial^2\hat{u}}{\partial\hat{y}^2} + \frac{\partial^2\hat{u}}{\partial\hat{z}^2}\right) \tag{3.114}$$

$$\frac{\partial\hat{v}}{\partial\hat{t}} + \frac{\partial}{\partial\hat{x}}(\hat{u}\hat{v}) + \frac{\partial}{\partial\hat{y}}(\hat{v}\hat{v}) + \frac{\partial}{\partial\hat{z}}(\hat{w}\hat{v}) = -\frac{\partial\hat{p}}{\partial\hat{y}} + \left(\frac{\partial^2\hat{v}}{\partial\hat{x}^2} + \frac{\partial^2\hat{v}}{\partial\hat{y}^2} + \frac{\partial^2\hat{v}}{\partial\hat{z}^2}\right) + Gr\hat{T} \tag{3.115}$$

$$\frac{\partial\hat{w}}{\partial\hat{t}} + \frac{\partial}{\partial\hat{x}}(\hat{u}\hat{w}) + \frac{\partial}{\partial\hat{y}}(\hat{v}\hat{w}) + \frac{\partial}{\partial\hat{z}}(\hat{w}\hat{w}) = -\frac{\partial\hat{p}}{\partial\hat{z}} + \left(\frac{\partial^2\hat{w}}{\partial\hat{x}^2} + \frac{\partial^2\hat{w}}{\partial\hat{y}^2} + \frac{\partial^2\hat{w}}{\partial\hat{z}^2}\right) \tag{3.116}$$

$$\frac{\partial\hat{T}}{\partial\hat{t}} + \frac{\partial}{\partial\hat{x}}\left(\hat{u}\hat{T}\right) + \frac{\partial}{\partial\hat{y}}\left(\hat{v}\hat{T}\right) + \frac{\partial}{\partial\hat{z}}\left(\hat{w}\hat{T}\right) = \frac{1}{Pr}\left(\frac{\partial^2\hat{T}}{\partial\hat{x}^2} + \frac{\partial^2\hat{T}}{\partial\hat{y}^2} + \frac{\partial^2\hat{T}}{\partial\hat{z}^2}\right) \tag{3.117}$$

where Gr is the Grashof number, Pr is the Prandtl number, and v the kinematic viscosity defined as

$$Gr = \frac{g\beta\Delta TL^3}{v^2} \quad Pr = \frac{\mu c_p}{k} \quad v = \frac{\mu}{\rho} \tag{3.118}$$

The Grashof and Prandtl numbers [12–14] are dimensionless groups formed of a combination of the involved physical properties. Therefore the number of parameters affecting the solution was reduced to two and solutions can be generated for different values of these two parameters. Moreover any single solution will be valid for many combinations of the physical properties of which these two numbers are composed, as long as these combinations result in the Gr and Pr values for which the solution was obtained. The physical significance of these two dimensionless numbers and others that may arise when writing the conservation equations in non-dimensional forms using other dimensionless parameters under different conditions is discussed next.

3.9 Dimensionless Numbers

Writing the conservation equations in non-dimensional forms, results in dimensionless numbers that are very useful for performing parametric studies of engineering problems. For incompressible viscous flow, the dimensionless parameters governing natural convection heat transfer were reduced to the two dimensionless numbers *Gr* [12–14] and *Pr* [12–14]. Under different conditions (e.g., compressible flows, Porous flows, etc.) other types of fluid forces and dissipation terms may be included in the governing equations resulting in different non-dimensional groups. For flow in porous media, for example, Darcy number (Da) [15, 16] emerges as an important parameter, for a free surface flow the Weber number (We) [17, 18], for an open channel flow the Froude number (Fr) [19], for a compressible flow the Mach number (M) [20], and so on. Some of the most important dimensionless groups are discussed below.

3.9.1 Reynolds Number

The Reynolds number (Re) [12, 13] is defined as

$$Re = \frac{\rho UL}{\mu} \tag{3.119}$$

and may be interpreted as a measure of the relative importance of advection (inertia) to diffusion (viscous) momentum fluxes. If the momentum fluxes are in the same direction then the Reynolds number reveals the boundary layer characteristics of the flow. If the fluxes are defined such that the diffusion is in the cross stream direction, then as shown in Fig. 3.10 *Re* conveys the flow regime (i.e. laminar, transitional, or turbulent).

An example showing the flow field for different values of Reynolds number is depicted in Fig. 3.11. It represents a driven flow in a square cavity of side L generated by the velocity U imparted to its top wall. The streamlines shown in Fig. 3.11 indicates that the strength of the flow increases as $Re = \rho UL/\mu$ increases.

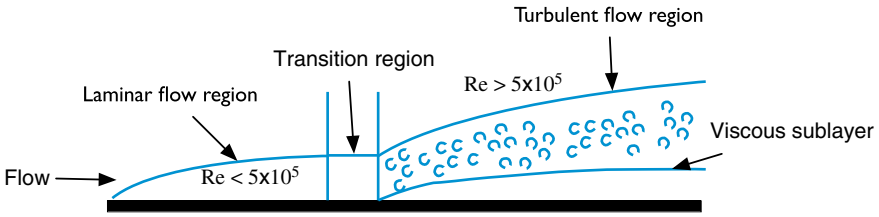


Fig. 3.10 Schematic of the flow over a flat plate showing the laminar, transitional, and turbulent flow regimes based on the value of *Re*

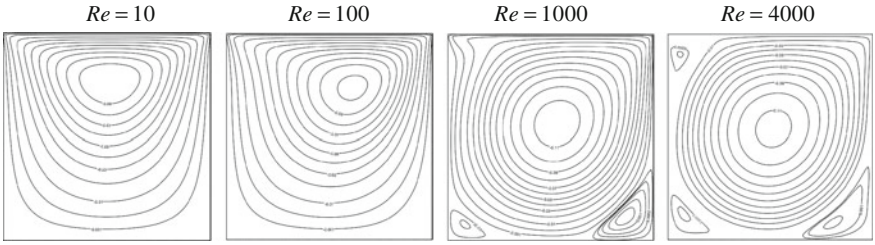


Fig. 3.11 Streamlines at increasing values of Reynolds number for driven flow in a square cavity

3.9.2 Grashof Number

As derived above the Grashof number [12–14] is given by

$$Gr = \frac{g\beta\Delta TL^3}{\nu^2} \tag{3.120}$$

The Grashof number represents the ratio of buoyant to viscous forces. It plays in natural convection the same role played by the Reynolds number in forced convection. An example showing the effect of Grashof number is depicted in Fig. 3.12. The physical situation represents natural convection heat transfer in the annulus between a hot circular cylinder and its cold square enclosure. Isotherms displayed in the figure are seen to become more distorted at higher values of Gr due to higher natural convection effects caused by a stronger flow field.

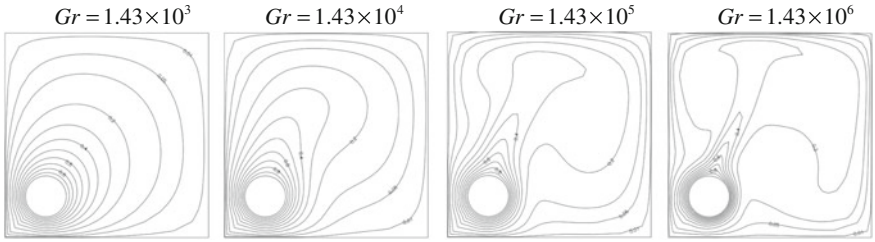


Fig. 3.12 Isotherms at increasing values of Grashof number for natural convection in the annulus between eccentric horizontal hot circular and cold square cylinders

3.9.3 Prandtl Number

The Prandtl number [12–14] is defined as the ratio of momentum diffusivity (kinematic viscosity ν) to thermal diffusivity (α), i.e.,

$$Pr = \frac{\mu c_p}{k} = \frac{\mu/\rho}{k/\rho c_p} = \frac{\nu}{\alpha} \tag{3.121}$$

The Prandtl number represents the ratio of hydrodynamic boundary layer to thermal boundary layer. As displayed in Fig. 3.13, the thermal boundary layer is larger than the hydrodynamic boundary layer for $Pr < 1$ (Fig. 3.13a) and the opposite is true for $Pr > 1$ (Fig. 3.13b). Both layers coincide for $Pr = 1$.

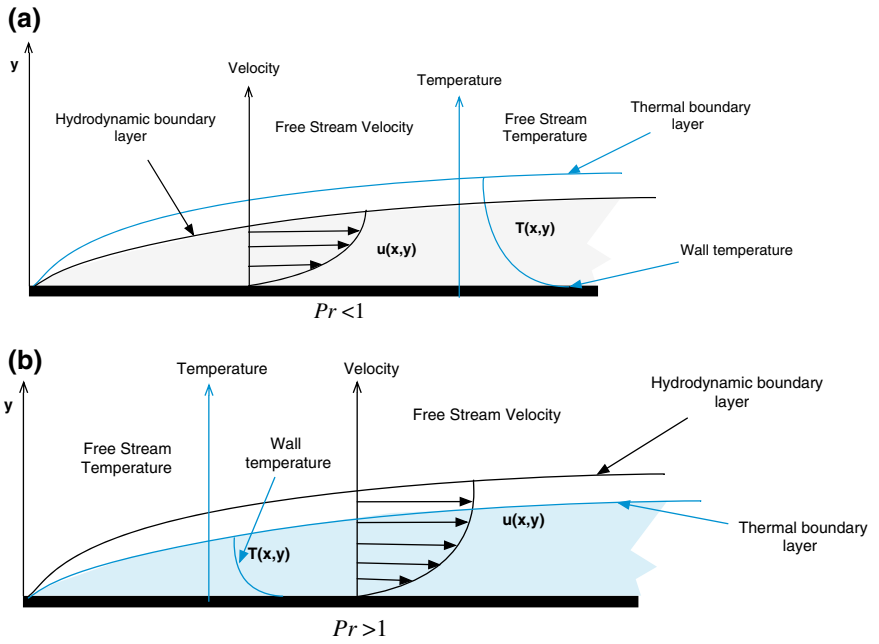


Fig. 3.13 The thermal and hydrodynamic boundary layer thicknesses for **a** $Pr < 1$ and **b** $Pr > 1$

For the driven flow in a square cavity problem presented above, isotherms over the domain are depicted in Fig. 3.14 for different values of Pr while holding Re constant at 100. The increase in convection over conduction as Pr increases can be easily inferred from the plots.

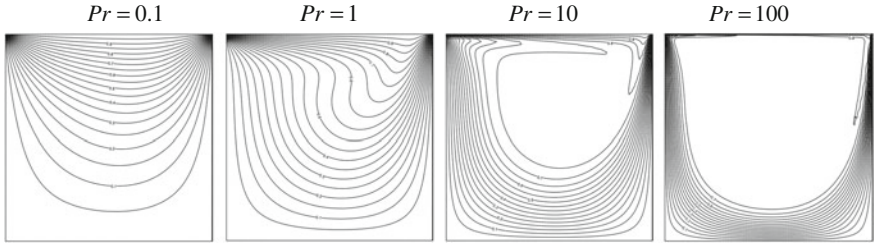


Fig. 3.14 Isotherms at increasing values of Prandtl number for driven flow in a square cavity ($Re = 100$)

3.9.4 Péclet Number

The Péclet number [5] is defined as the ratio of the advective transport rate of a physical quantity to its diffusive transport rate. For the case of heat transfer, the Péclet number is given by

$$Pe = \frac{\rho U L c_p}{k} = \frac{UL}{\alpha} = Re^* Pr \tag{3.122}$$

In this situation the Pe is equivalent to the product of the Reynolds number and the Prandtl number. An example of the effects of Pe is shown in Fig. 3.15, where isotherms over a flat hot plate are displayed at different values of Péclet number. Heat transfer is seen to be dominated by conduction at low values of Pe with convection gaining increasing importance as Pe increases to become clearly the dominant heat transfer mode at $Pe = 1000$.

Fig. 3.15 Isotherms at increasing values of Péclet number for fluid flow over a flat plate maintained at a hot uniform temperature

For mass transport, the Péclet number is given by

$$Pe = \frac{UL}{D} = Re^* Sc \tag{3.123}$$

where D is the mass diffusivity and Sc the Schmidt number. In this case Pe is equivalent to the product of the Reynolds number and the Schmidt number.

A large Péclet number indicates low dependence of the flow on downstream locations and high dependence on upstream locations. Therefore simpler computational models can be adopted for simulating situations with high Péclet numbers.

3.9.5 Schmidt Number

The Schmidt number [14] is defined as

$$Sc = \frac{\nu}{D} \tag{3.124}$$

The Schmidt number in mass transfer is the counterpart of the Prandtl number in heat transfer. It represents the ratio of the momentum diffusivity (ν) to mass

diffusivity (D). Physically, the Sc relates the thicknesses of the hydrodynamic and mass transfer boundary layers. An example showing the effect of Schmidt number is shown in Fig. 3.16.

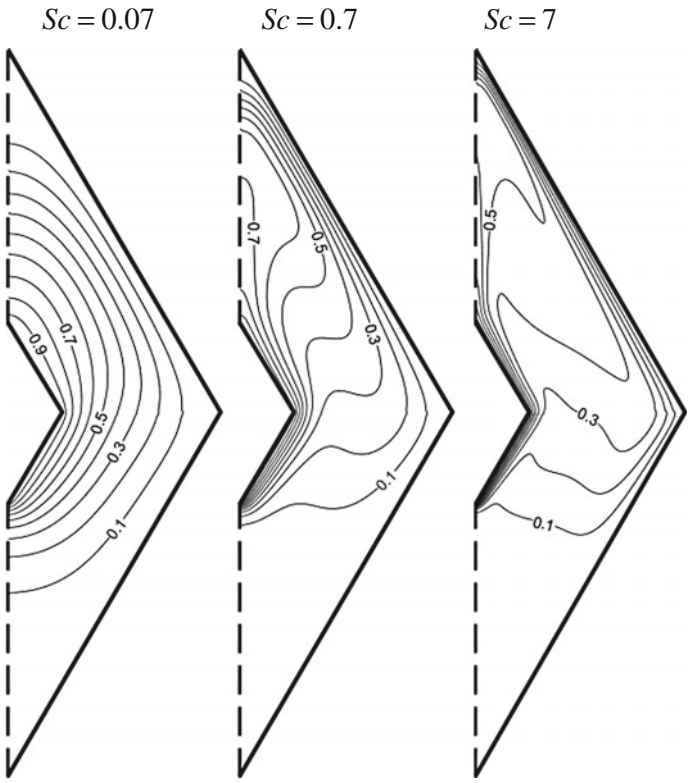


Fig. 3.16 Iso-concentrations at increasing values of Schmidt number (other parameters held fixed) for natural convection mass transfer in the annulus between concentric horizontal cylinders of rhombic cross sections with larger solute concentration on the inner wall

The figure above represents natural convection mass transfer in the annulus between two horizontal pipes of rhombic cross sections. The solute concentration is higher along the inner wall of the enclosure. The concentration non-uniformity causes variations in density establishing a flow field. The strength of the flow increases with increasing Sc values as manifested by the higher distortion of iso-concentration lines that indicates an increase in convection mass transfer over diffusion mass transfer, which dominates at low Sc values.

3.9.6 Nusselt Number

The Nusselt number [12–14] expressed as

$$Nu = \frac{hL}{k} \tag{3.125}$$

is the dimensionless form of the convection heat transfer coefficient h and provides a measure of the convection heat transfer at a solid surface. The Nusselt number does not arise as a dimensionless group when writing the conservation equations in non-dimensional forms; rather, it is widely used to report convection heat transfer data.

3.9.7 Mach Number

The Mach number (M) [20] is defined as the ratio of speed of an object moving through a fluid and the local speed of sound [20]. Mathematically it is written as

$$M = \frac{|\mathbf{v}|}{a} \tag{3.126}$$

where $|\mathbf{v}|$ is the local magnitude of the fluid velocity relative to the medium in which it is flowing and a is the speed of sound. The general equation for the speed of sound is given by

$$a = \sqrt{\gamma \left(\frac{\partial p}{\partial \rho}\right)_T} \tag{3.127}$$

For an ideal gas, it reduces to

$$a = \sqrt{\gamma RT} \tag{3.128}$$

where γ is the ratio of specific heat at constant pressure to specific heat at constant volume $\left(c_p/c_v\right)$ and R is the gas constant.

Flows for which the Mach number is less than 0.2 can be treated as incompressible. For $M < 1$ the flow is called subsonic, for $M = 1$ sonic, for $1 < M < 5$ supersonic, and for $M > 5$ hypersonic. Moreover, a flow accelerating from subsonic to supersonic is called a transonic flow. The value of Mach number (less than 1 or greater than 1) at the boundaries of a domain dictates the number of required boundary conditions there.

Examples of subsonic, transonic, and supersonic flow fields are presented in Fig. 3.17 via Mach contours. The physical situation represents a fluid flowing over

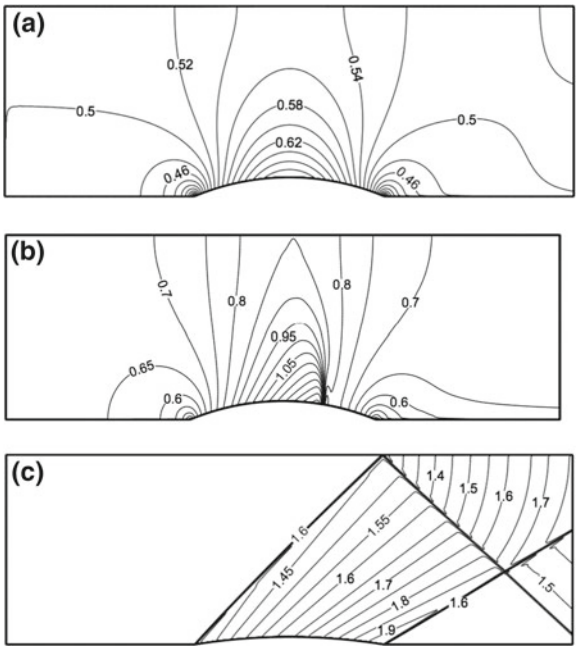


Fig. 3.17 Mach contours for the flow over a circular arc bump at **a** subsonic, **b** transonic, and **c** supersonic speeds

a circular arc bump with a maximum curvature of 10 % the channel height in the subsonic and transonic cases and of 4 % in the supersonic case. The change in the flow type from elliptic to hyperbolic (with discontinuities in the form of shock waves) as the Mach number increases from subsonic $(M<1)$ to supersonic $(M>1)$ is apparent.

3.9.8 Eckert Number

The Eckert number (Ec) [21] is a dimensionless number relating the kinetic energy of the flow to its enthalpy and is computed as

$$Ec = \frac{\mathbf{v}\cdot\mathbf{v}}{c_p \Delta T} \tag{3.129}$$

where ΔT is a characteristic temperature difference. This dimensionless number appears as a factor multiplying the viscous dissipation term Φ, when non-dimensionalizing the compressible energy equation. A large value of Ec indicates high viscous dissipation occurring at high speed of the flow (high kinetic energy). For small Eckert number $(Ec \ll 1)$ several terms in the energy equation become negligible (e.g., viscous dissipation, body forces, etc.). This reduces the energy equation to its incompressible form (i.e., a balance between conduction and convection).

3.9.9 Froude Number

The Froude number (Fr) [19] is a dimensionless number defined as the ratio of a characteristic velocity (U) to a gravitational wave velocity $(\sqrt{gL})$ as

$$Fr = \frac{U}{\sqrt{gL}} \tag{3.130}$$

It is a measure of the resistance of partially immersed objects moving through fluids, with higher Fr values indicating higher fluid resistance.

For the free-surface flow shown in Fig. 3.18, the nature of the flow is dictated by the value of Froude number. For $Fr > 1$ the flow is supercritical and for $Fr < 1$ it is subcritical. The flow at the interface between the two regions, known as the "hydraulic jump", is just critical and is characterized by a Froude number value of 1.

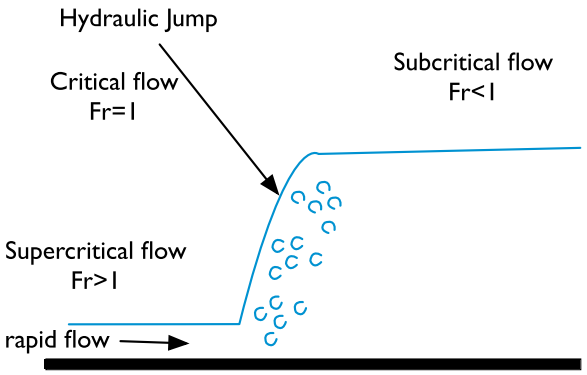


Fig. 3.18 A free surface flow showing the supercritical, critical, and subcritical regions

3.9.10 Weber Number

The dimensionless Weber number (We) [17, 18] is defined as

$$We = \frac{\rho U^2 L}{\sigma} \tag{3.131}$$

where U (m/s) and L (m) are the characteristic velocity and length, respectively, and σ the surface tension (N/m). The Weber number, which represents the ratio of inertia to surface tension forces, is helpful in analyzing multiphase flows involving interfaces between two different fluids, with curved surfaces such as droplets and bubbles.

3.10 Closure

This chapter has shown that many physical phenomena can be modeled through conservation equations. These equations are derived from first principles by writing balances over a finite volume. It was also shown that the conservation equations governing the transport of mass, momentum, energy, and other specific quantities have a common form embodied in the general scalar transport equation. This equation has transient, convection, diffusion and source terms. Each term brings a characteristic contribution to the equation that needs to be reproduced by the discretization procedure.

3.11 Exercises

Exercise 1
By comparing the continuity, momentum, and energy equations with the general scalar transport equation, derive expressions for ϕ, Γ^{ϕ} and S^{ϕ}.

Exercise 2
Show that for an incompressible flow of constant viscosity the following holds:

$$\nabla \cdot \left\{ \mu \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^{\mathrm{T}} \right] \right\} = \mu \nabla^2 \mathbf{v}$$

Exercise 3
A steady incompressible flow field is defined by the following velocity vector:

$$\mathbf{v} = (x + y)\mathbf{i} + (y + z)\mathbf{j} + 2(x - z)\mathbf{k}$$

(a) Verify that it satisfies the continuity equation.
(b) Assuming constant viscosity μ, calculate the viscous stress tensor $\boldsymbol{\tau}$.
(c) Denoting the fluid density by ρ and neglecting body forces, develop an equation for the pressure gradient.

Exercise 4
The vorticity ω of a flow field is defined as the curl of the velocity vector, i.e.,

$$\omega = \nabla \times \mathbf{v}$$

Using the above definition of vorticity, show that for an incompressible fluid, the following relation between the velocity and vorticity vectors holds:

$$\nabla \cdot [(\mathbf{v} \cdot \nabla)\mathbf{v}] = 0.5\nabla^2(\mathbf{v} \cdot \mathbf{v}) - \mathbf{v} \cdot \left[\nabla^2 \mathbf{v}\right] - \omega \cdot \omega$$

Exercise 5
A flow is said to be irrotational if its vorticity (defined in Exercise 4 above) is zero, i.e., $\omega = 0$. Show that for a steady two dimensional incompressible irrotational flow the velocity field satisfies the following Laplace equation

$$\nabla^2 \mathbf{v} = 0$$

Exercise 6
Consider a two-dimensional square enclosure of side L. The enclosure is filled with an incompressible fluid of viscosity μ and density ρ. The top side of the enclosure is covered with an infinite horizontal wall moving with a constant velocity U. The other sides are fixed in place. Due to the motion of the top wall a flow field is established within the enclosure. Using appropriate dimensionless variables, write the simplified momentum equation for the flow field in dimensionless form showing that the Reynolds number ($Re = \rho UL/\mu$) is the only dimensionless group affecting the flow.

Exercise 7
Consider the steady two dimensional mixed convection heat transfer in a vertical rectangular channel of width W. A cold fluid of density ρ_{in} and temperature T_{in} enters the channel with a velocity V_{in}. As it flows vertically upward, the fluid is heated by the duct walls, which are maintained at the uniform hot temperature T_w. Taking buoyancy forces into consideration through the Boussinesq approximation and using the following dimensionless variables

$$x^* = \frac{x}{W}, y^* = \frac{y}{W}, u^* = \frac{u}{V_{in}}, v^* = \frac{v}{V_{in}}, \theta = \frac{T - T_{in}}{T_w - T_{in}}, p^* = \frac{p + \rho_{in} g y}{\rho_{in} V_{in}^2}$$

write the conservation equations of mass, momentum, and energy in dimensionless forms. What are the dimensionless groups governing the flow and heat transfer in the channel? What does each of them represent?

Exercise 8
Estimates the Reynolds number of the following flows:

(a) Water flowing at a speed of 15 km/hr over a whale 10 m long.
(b) Air flowing at a speed of 800 km/hr over the wing of an F16 airplane of mean chord length 3.450336 m.
(c) Glycerine of dynamic viscosity 0.96 kg/ms and density 1258 kg/m^3 flowing at a speed of 2.8 m/s in a pipe inclined at 25° to the horizontal and of diameter 250 mm.

Exercise 9 Starting from the incompressible version of the Navier-Stokes equations derive simplified equations based on the following assumptions:

(a) Viscous effects are much more significant than any effects of fluid acceleration, i.e.,

$$\frac{\partial}{\partial t}(\mathbf{v}) + \nabla \cdot [\mathbf{vv}] \ll \nabla \cdot [\mu \nabla \mathbf{v}]$$

which corresponds to $Re = \rho UL/\mu \ll 1$ (Stokes Equations).

(b) Inertial effects dominate and viscous effects are considered to be negligible throughout the flow domain, i.e.,

$$\frac{\partial}{\partial t}(\mathbf{v}) + \nabla \cdot [\mathbf{vv}] \gg \nabla \cdot [\mu \nabla \mathbf{v}]$$

which corresponds to $Re = \rho UL/\mu \gg 1$ (Euler equations).

(c) Derive the Bernoulli equation from momentum conservation with the following hypothesis: one dimensional steady state conditions of a frictionless fluid $\mu = 0$.

References

1. Navier CLMH (1823) Mem Acad R Sci Paris 6:389–416
2. Stokes GG (1845) Trans Camb Phil Soc 8:287–305
3. Hauke G (2008) An Introduction to Fluid Mechanics and transport phenomena. Springer, New York
4. Ferziger JH, Peric M (2002) Computational methods for fluid dynamics. Springer, New York
5. Patankar SV (1980) Numerical heat transfer and fluid flow. Hemisphere Publishing Corporation, USA
6. Bird RB, Stewart WE, Lightfoot EN (2006) Transport phenomena, 2nd edn. Wiley, USA
7. Falkovich G (2011) Fluid mechanics (A Short Course for Physicists). Cambridge University Press, Cambridge
8. Emanuel G (2000) Analytical fluid dynamics. CRC Press, Boca Raton
9. Reynolds O (1903) Papers on mechanical and physical subjects-the sub-mechanics of the universe. Collected Work, vol III, Cambridge University Press, Cambridge
10. Fay JA (1994) Introduction to fluid mechanics. MIT Press, Cambridge Massachussets
11. Boussinesq J (1897) Theorie de l'ecoulement tourbillonnant et tumulueux des liquides and les lits rectilignes a grande section. Gauthier-Villars et Fils, Des Comptes Rendus des Seances de L'academie des Sciences, Paris
12. Cengel YA (2003) Heat and mass transfer: a practical approach, 3rd edn. McGraw-Hill, Boston
13. Incropera FP, DeWitt DP (2007) Fundamentals of heat and mass transfer, 6th edn. Wiley, Hoboken
14. Bejan A (1984) Convection heat transfer. Wiley, USA
15. Moukalled F, Darwish M (2013) Double diffusive natural convection in a porous rhombic annulus. Numer Heat Transfer Part A: Appl 65(5):378–399
16. Moukalled F, Darwish M (2010) Natural convection heat transfer in a porous rhombic annulus. Numer Heat Transfer, Part A: Appl 58(2):101–124
17. Frohn A, Roth N (2000) Dynamics of droplets. Springer, New York

18. Day P, Manz A, Zhang Y (2012) Microdroplet technology: principles and emerging applications in biology and chemistry. Springer, New York
19. Chanson H (2004) Hydraulics of open channel flow: an introduction, 2nd edn. Butterworth–Heinemann, Oxford
20. Oosthuizen PH, Carscallen WE (1997) Compressible fluid flow. McGraw-Hill, Singapore
21. Kreith F, Bohn MS (1993) Principles of heat transfer, 5th edn. West Publishing Company, USA

Chapter 4
The Discretization Process

Abstract This chapter introduces the different steps of the discretization process, which include: (i) modeling of the geometric domain and the physical phenomena of interest; (ii) discretization of the modeled geometric domain into a grid or mesh that forms the computational domain (this process, also known as meshing or domain discretization, results in a set of non-overlapping elements, denoted also by cells, that cover the computational domain); (iii) numerical or equation discretization that transforms the set of conservation partial differential equations governing the physical processes into an equivalent system of algebraic equations defined over each of the elements of the computational domain; and (iv) the solution of the resulting set of equations using an iterative solver to yield an intermediate or final solution field. Throughout the chapter, computer implementation issues are introduced.

4.1 The Discretization Process

The numerical solution of a partial differential equation consists of finding the values of the dependent variable ϕ at specified points from which its distribution over the domain of interest can be constructed. These points are called *grid elements*, or *grid nodes and result from the discretization of original geometry into a set of non overlapping discrete elements, a process known as meshing. The resulting nodes or variables* are generally positioned at *cell centroids* or at vertices depending on the adopted discretization procedure. In all methods the focus is on replacing the continuous exact solution of the partial differential equation with discrete values. The distribution of ϕ is hence *discretized*, and it is appropriate to refer to this process of converting the governing equation into a set of algebraic equations for the discrete values of ϕ as the *discretization process* and the specific methods employed to bring about this conversion as the *discretization methods*. The discrete values of ϕ are typically computed by solving a set of *algebraic equations* relating the values at neighboring grid elements to each other; these discretized or

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_4

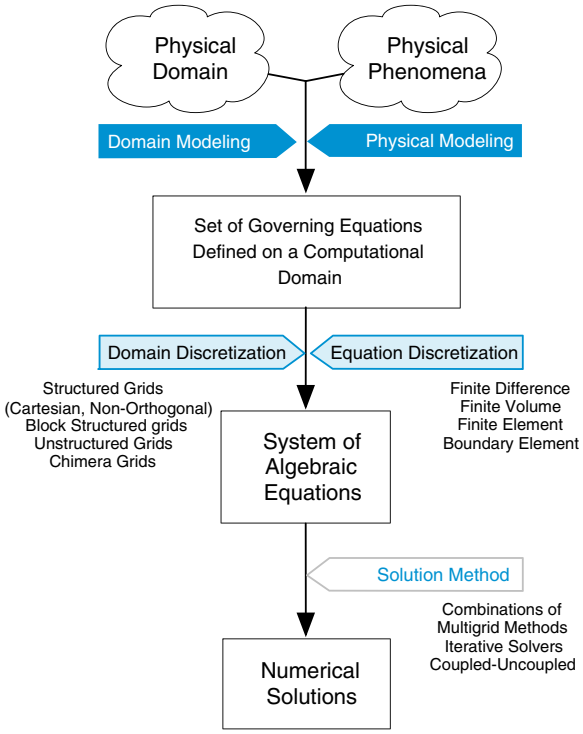


Fig. 4.1 The discretization process

algebraic equations are derived from the conservation equation governing ϕ. Once the values of ϕ are computed, the data is processed to extract any needed information. The various stages of the discretization process are illustrated in Fig. 4.1.

Figure 4.2 shows the process applied to the study of heat transfer from a microprocessor connected to a heat sink with a copper base that acts as a heat spreader. The example of Fig. 4.2 will be used to present various concepts related to the discretization process. Since the intention is to introduce a numerical technique for solving the physical processes of interest and since the method has to be implemented in a computer program, the discretization process will be explained along that spirit. For example, reference will be made to how to store values in a code as related to interior elements, boundary elements, variables, etc. Throughout the book a Matlab® based program denoted by "uFVM" will be used as a development vehicle to present various details relating to the implementation of these methods. Furthermore OpenFOAM®, a very popular finite volume-based open source code will also be presented both from a user and a developer perspective, again with the aim of moving from the numerics to the implementation details.

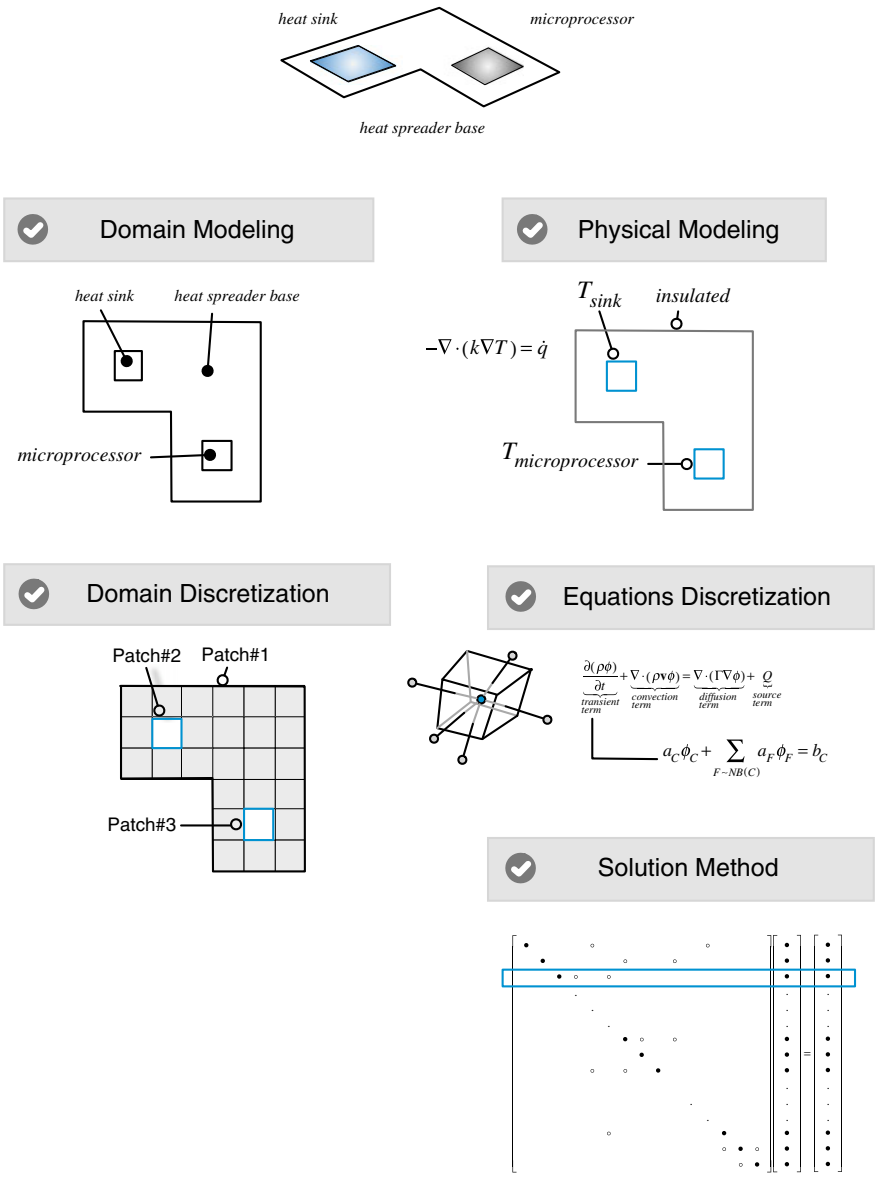


Fig. 4.2 An illustration of the discretization process

4.1.1 Step I: Geometric and Physical Modeling

Modeling of physical phenomena is in a way at the heart of the scientific enterprise. A physical phenomenon cannot generally be considered as understood unless it can

be mathematically formulated, and this formulation tested and validated. For our purpose two levels of modeling are performed, one in relation to the geometry of the physical domain and a second in relation to the physical phenomena of interest. At both levels details that are neither relevant nor of interest are ignored or simplified. For example a three dimensional domain could be turned into a two dimensional depiction, or symmetry can be taken into account to decrease the size of the study domain. In some cases, physical components may be removed and replaced with appropriate mathematical representations.

In the example of Fig. 4.2 a microprocessor is connected to a heat sink with a copper base that acts as a heat spreader. A first model of this system simplifies both its physics and its geometry. The heat sink and processor are replaced by boundary conditions that specify the estimated temperature of the heat sink and the expected operating temperature of the processor, respectively. The physical domain is modeled as a two dimensional computational domain since the temperature variation through the thickness of the heat sink will be minimal. For a steady state solution of the heat flow and temperature distribution in the copper base, only heat conduction is considered. The result of the modeling process is a system of linear (or non-linear if k depends on T) partial differential equations, which in this case involves a simplified form of the energy equation given by

$$-\nabla \cdot (k\nabla T) = \dot{q} \tag{4.1}$$

where k is the conductivity of the heat spreader base, and $\dot{q}$ is the heat source/sink per unit volume.

4.1.2 Step II: Domain Discretization

The geometric discretization of the physical domain results in a mesh on which the conservation equations are eventually solved. This requires the subdivision of the domain into discrete non-overlapping cells or elements that completely fill the computational domain to yield a grid or mesh system. This is accomplished by a variety of techniques resulting in a wide range of mesh types. These meshes are classified according to several characteristics: structure, orthogonality, blocks, cell shape, variable arrangement, etc. In all cases the mesh is composed of discrete elements defined by a set of vertices and bounded by faces. For the mesh to be a useful platform for equation discretization, information related to the topology of the mesh elements, in addition to some derived geometric information, are needed. These include element to element relations, face to elements relations, geometric information of the surfaces, element centroid and volume, face centroid, area and normal direction, etc. This information is usually inferred from the basic mesh data. For certain mesh topologies, details about the mesh can be easily deduced from the element indices as in structured grids, while for others it has to be constructed and stored in lists for later retrieval, as is the case with unstructured grids.

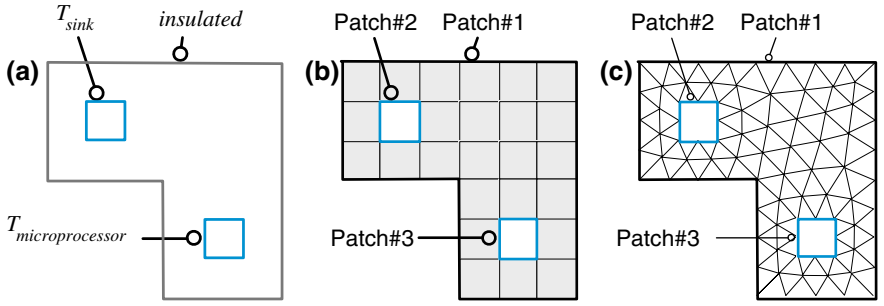


Fig. 4.3 **a** Computational domain; **b** computational mesh (*quadrilateral*); **c** computational mesh (*triangular*)

Consider the simple domain shown in Fig. 4.3a. The domain consists of a volume (area for the two dimensional case) and boundaries that account for the heating or cooling of the microprocessor, a heat sink, and the heat spreader base. The domain is shown discretized with a simple mesh in Fig. 4.3b. The mesh boundary is divided into three patches of boundary faces that are assigned numbers, i.e., Patch#1, Patch#2, and Patch#3. These patches are used to define the physical boundary conditions for the problem at hand. The mesh consists of 25 non-overlapping elements whose geometry is defined by 40 points (vertices of the cells). The elements are also bounded by 66 faces (lines in a two dimensional case), 34 of which are interior faces. The algebraic equations that result from the discretization of the governing equations, as will be explained in step III, are described for each element in the computational domain with the solution expressed as an element field with values defined at the centroid of each element. In this example the elements have a square shape, though other shapes could have been used (e.g., triangular elements, as shown in Fig. 4.3c).

The mesh can be described from different perspectives. At the most elementary level it is a list of **vertices** or **points** representing locations in one dimensional, two dimensional, or three dimensional spaces. The mesh also represents the discretized domain subdivided into non-overlapping elements, which can be of arbitrary convex polyhedral shapes. Elements are completely bounded by faces that are generally shared by neighboring elements, except at the boundaries. Elements can be defined either in terms of the points that delimit them or in terms of the faces that bound them. The mesh faces, which are stored in a list, are of two types: (i) interior faces that are shared by (or connect) two elements, and (ii) boundary faces that coincide with the domain boundary; these boundary faces have only one contiguous element. While interior faces are derived from information related to the element topology, it is essential to provide boundary faces as they define the domain physical boundary. In two dimensions faces are described in terms of their defining points. In three dimensions the defining points describe edges that bound the face. The direction of the normal to an interior face is usually defined based on the topology of the neighboring elements. On the other hand, the direction of the

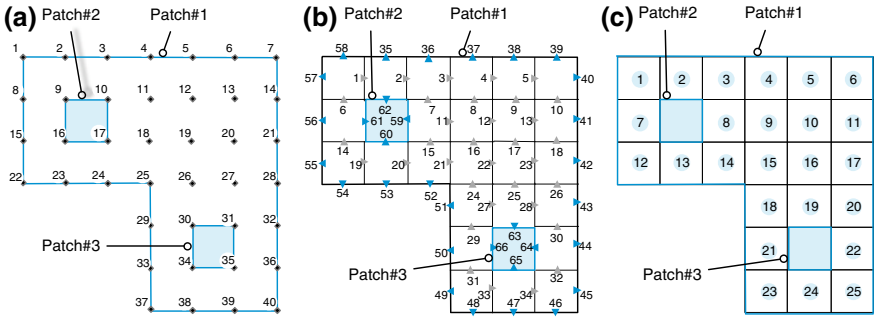


Fig. 4.4 **a** Mesh vertices, **b** faces, and **c** elements

normal to a boundary face always points outward of the domain. Figure 4.4 shows some of the components (vertices, faces, and elements shown in Fig. 4.4a–c respectively) of a mesh. Furthermore the boundary faces are organized into lists of faces based on the boundary patch to which they belong.

4.1.3 Mesh Topology

During discretization, the partial differential equations are integrated over each element in the mesh resulting in a set of algebraic equations with each one linking the value of the variable at an element to the values at its neighbors. The algebraic equations are then assembled into global matrices and vectors and the coefficients of every equation stored at the row and column locations corresponding to the various element indices. The integration of the equations over each element is referred to as local assembly while the construction of the overall system of equations from these contributions is referred to as global assembly. Thus while the discretization of the equations is derived in terms of neighbor elements, the assembly of the equations in the global matrix accounts for the actual indices of the elements. This procedure will be detailed in later chapters, however the enabling ingredients of this procedure are briefly introduced next at their most elementary level, which is in the form of topological information about elements, faces, and vertices that are represented in terms of connectivity lists.

Element connectivity relates the local assembly matrix to the global matrix so that the equations formed for one element are consistent with the equations formed for the other elements in the computational domain. Generally element to element, element to face, and element to vertex connectivities are setup. These relate the element to the neighboring elements, bounding faces, and defining vertices, respectively. Considering Fig. 4.4, the connectivity for element 9 is shown in Fig. 4.5.

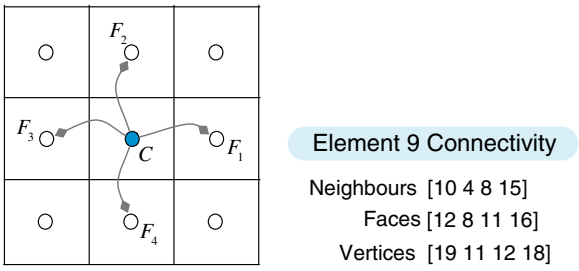


Fig. 4.5 Element connectivity

Generally for arbitrary elements it is more efficient to assemble flux terms by looping over faces. In this case it is essential that information about the face element neighbors be readily available; this is defined in the **Face connectivity**. For faces, information about elements sharing the face is stored for use during computations. The orientation of the face is such that the normal vector to the face points from one element denoted by *element 1* or *owner* to the second element denoted by *element 2* or *neighbor*. Boundary faces bound only one element, defined as element 1, thus the normal vector of boundary faces is always oriented outside of the domain. The connectivity for Face 12 is shown in Fig. 4.6.

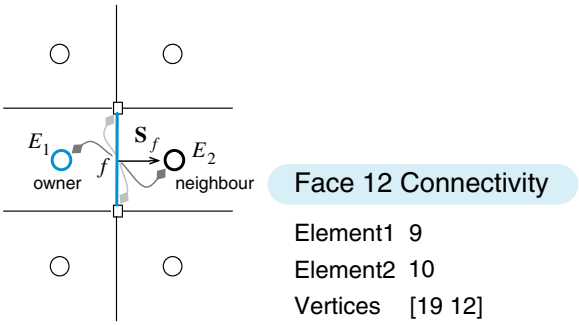


Fig. 4.6 Face connectivity

Vertex connectivity is useful for post processing and for gradient computation. As shown in Fig. 4.7, generally it involves the lists of elements and faces that share the vertex.

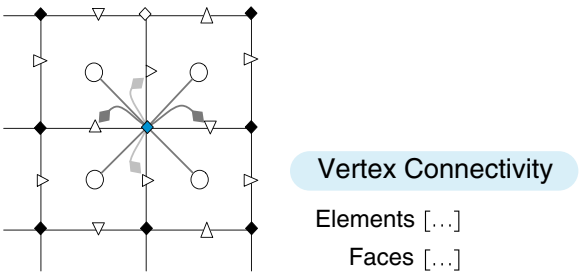


Fig. 4.7 Vertex connectivity

The mapping between local and global indices is briefly illustrated in Fig. 4.8 for a mesh of five elements.

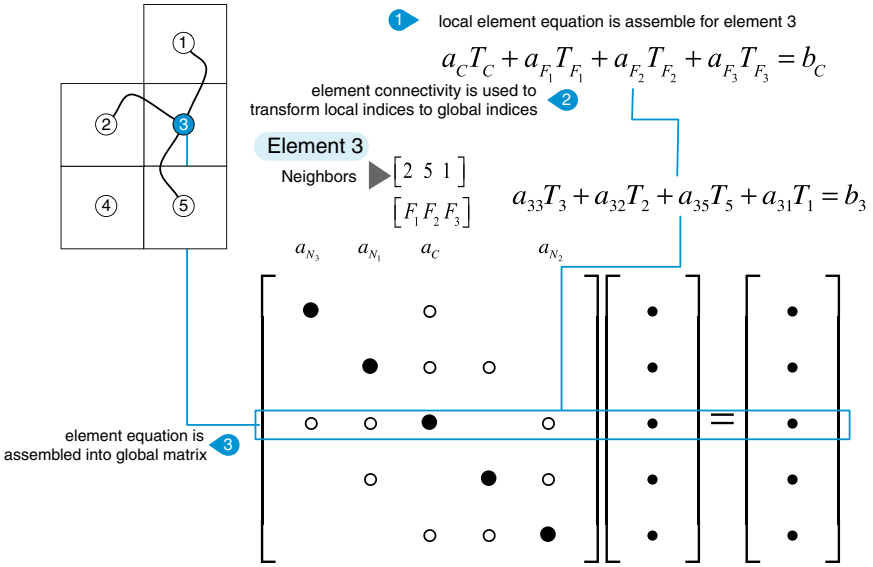


Fig. 4.8 Local element matrix assembly into global matrix

Example 1

For the mesh shown below, derive the element connectivity and represent it in a global matrix (Fig. 4.9)

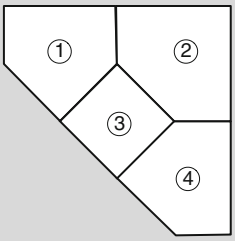


Fig. 4.9 Mesh for example 1

Solution

In this mesh element counting starts from 1. The connectivity for the various elements are given by

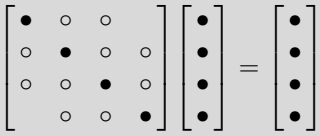
$1 \rightarrow 2, 3$

$2 \rightarrow 1, 3, 4$

$3 \rightarrow 1, 2, 4$

$4 \rightarrow 2, 3$

this can be represented in a global matrix as



(4.2)

4.1.4 *Step III: Equation Discretization*

In step III, the governing partial differential equations, are transformed into a set of algebraic equations, one for each element in the computational domain. These algebraic equations are then assembled into a global matrix and vectors that can be expressed in the form

$$\mathbf{A}[T] = \mathbf{b} \tag{4.2}$$

where the unknown variable T is defined at each interior element and at the boundary of the computational domain. Boundary values for T are generally obtained from the specified boundary conditions. To this end an element field has to be defined for T, and generally for each governing equation.

As schematically depicted in Fig. 4.10, the **element field** consists of an array of values defined at the centroid of each element, designated by the interior element field, which is represented by one array of size equal to the total number of interior and boundary elements.

The equation discretization step is performed over each element of the computational domain to yield an algebraic relation that connects the value of a variable

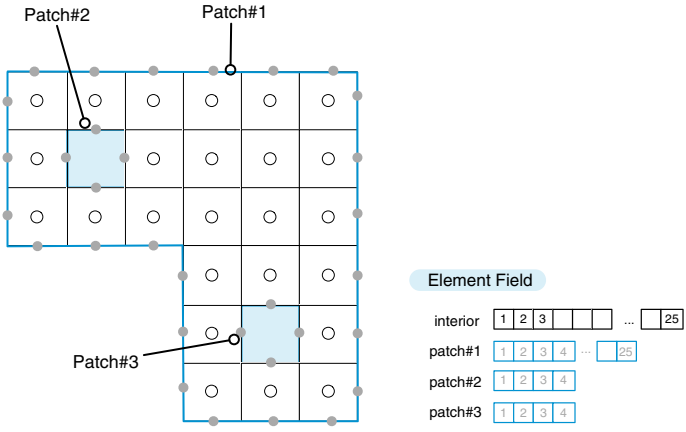


Fig. 4.10 Element field

in an element to the values of the variable in the neighboring elements. This algebraic equation is derived by discretizing the differential equation, which for the example considered is the energy equation written in terms of temperature T, (i.e., T is the unknown variable). As shown below, in the finite volume method the discretization of the equation is performed by first integrating the differential equation over a control volume or cell to obtain a semi discretized form of the equation and then approximating the variation of the dependent variable between grid elements through imposed profiles to obtain the final discretized form. The fact that only a few grid elements participate in a given discretization equation is a consequence of the piecewise nature of the chosen profiles. The value of T at a grid point thereby influences the distribution of T only in its immediate neighborhood. As the number of grid elements increases, the solution of the discretized equations is expected to approach the exact solution of the corresponding differential equation. This follows from the consideration that, as the grid elements get closer together, changes in T between neighboring grid elements become small, and then the actual details of the profile assumption become unimportant.

For a given differential equation, the possible discretization equations are by no means unique, although all types of discretization techniques in the limit of a very large number of grid elements are expected to give the same solution. The different types arise from the differences in the profile assumptions and the methods of derivation.

As an example of the equation discretization step using the finite volume method, the discretized form of the energy equation over the control volume C shown in Fig. (4.11) is sought. The process starts by integrating Eq. (4.1) over element C that enables recovering its integral balance form, which was described in Chap. 3, as

$$-\iint_{V_C} \nabla \cdot (k \nabla T) dV = \iint_{V_C} \dot{q} dV \tag{4.3}$$

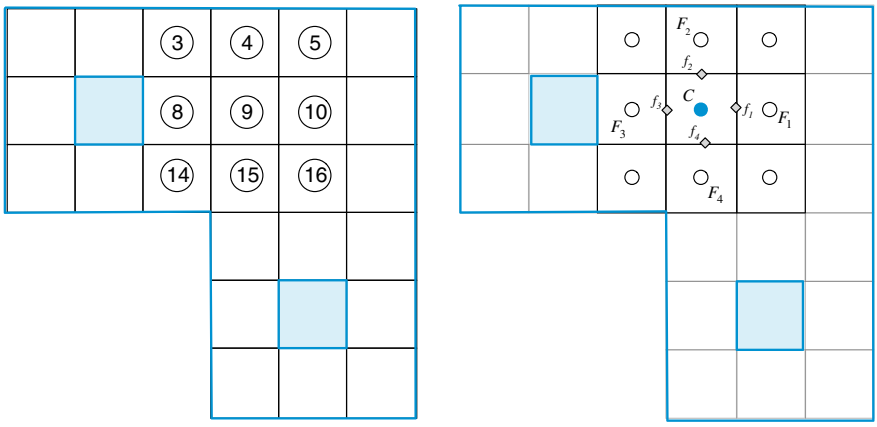


Fig. 4.11 Discretization stencil

Then, using the divergence theorem, the volume integral is transformed into a surface integral yielding

$$-\int_{S_C} (k\nabla T)\cdot d\mathbf{S} = \dot{q}_C V_C \tag{4.4}$$

This equation is actually a heat balance over element C. It is basically the integral form of the original partial differential equation and involves no approximation. Replacing the surface integral by a summation over the control volume faces, Eq. (4.4) becomes

$$-\sum_{f\sim nb(C)} (k\nabla T)_f \cdot \mathbf{S}_f = \dot{q}_C V_C \tag{4.5}$$

where f represents the integration point at the centroid of the bounding face. This transformation is the first approximation introduced. Therefore the integral in Eq. (4.4) is numerically approximated by the fluxes at the centroids of the faces. This is a second order approximation as will be demonstrated in a later chapter.

Expanding the summation, Eq. (4.5) can be written as

$$-(k\nabla T)_{f_1}\cdot\mathbf{S}_{f_1} - (k\nabla T)_{f_2}\cdot\mathbf{S}_{f_2} - (k\nabla T)_{f_3}\cdot\mathbf{S}_{f_3} - (k\nabla T)_{f_4}\cdot\mathbf{S}_{f_4} = \dot{q}_C V_C \tag{4.6}$$

Considering face f_1 shown in Fig. (4.12), the surface vector and temperature gradient in Eq. (4.6) are given by

$$\begin{aligned} \mathbf{S}_{f_1} &= \Delta y_{f_1}\mathbf{i} \\ \delta x_{f_1} &= x_{F_1} - x_C \\ \nabla T_{f_1} &= \left(\frac{\partial T}{\partial x}\right)_{f_1}\mathbf{i} + \left(\frac{\partial T}{\partial y}\right)_{f_1}\mathbf{j} \end{aligned} \tag{4.7}$$

where

x_C is the x-coordinate of the centroid of element C.
x_{F_1} is the x-coordinate of the centroid of element F_1.
Δy_{f_1} is the area of face f_1.
$\mathbf{S}_{f_1}$ is the surface vector of face f_1 directed out of element C.
∇T_{f_1} is the gradient of T at the centroid of face f_1.

and by substitution, the first term in Eq. (4.6) is converted to

$$\begin{aligned} \nabla T_{f_1}\cdot\mathbf{S}_{f_1} &= \left(\frac{\partial T}{\partial x}\mathbf{i} + \frac{\partial T}{\partial y}\mathbf{j}\right)_{f_1}\cdot\Delta y_{f_1}\mathbf{i} \\ &= \left(\frac{\partial T}{\partial x}\right)_{f_1}\Delta y_{f_1} \end{aligned} \tag{4.8}$$

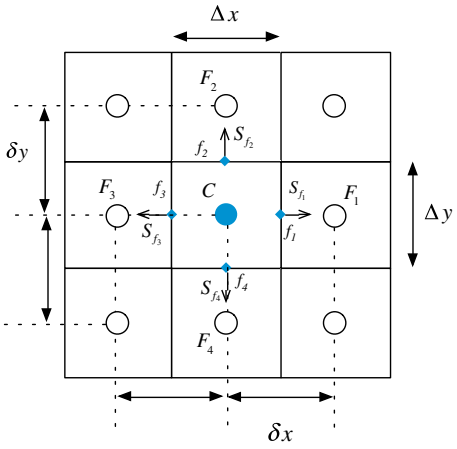


Fig. 4.12 Finite volume notation

To proceed further, a profile approximating the variation of T between C and F_1 is needed. Assuming linear variation of T, the x-component of the gradient at the face f_1 can be written as

$$\left(\frac{\partial T}{\partial x}\right)_{f_1} = \frac{T_{F_1} - T_C}{\delta x_{f_1}} \tag{4.9}$$

thus Eq. (4.8) can be approximated as

$$\nabla T_{f_1} \cdot \mathbf{S}_{f_1} = \frac{T_{F_1} - T_C}{\delta x_{f_1}} \Delta y_{f_1} \tag{4.10}$$

or more generally as

$$-(k\nabla T)_{f_1} \cdot \mathbf{S}_{f_1} = a_{F_1}(T_{F_1} - T_C) \tag{4.11}$$

where

$$a_{F_1} = -k\frac{\Delta y_{f_1}}{\delta x_{f_1}} \tag{4.12}$$

Repeating for each of the remaining faces, the following coefficients are obtained:

$$
\begin{aligned}
a_{F_2} &= -k\frac{\Delta x_{f_2}}{\delta y_{f_2}} \\
a_{F_3} &= -k\frac{\Delta y_{f_3}}{\delta x_{f_3}} \\
a_{F_4} &= -k\frac{\Delta x_{f_4}}{\delta y_{f_4}}
\end{aligned}
\tag{4.13}
$$

which when substituted into Eq. (4.6) yields

$$
\begin{aligned}
-\sum_{f\sim nb(C)}(k\nabla T)_f\cdot\mathbf{S}_f &= \sum_{F\sim NB(C)} a_F(T_F - T_C) \\
&= -(a_{F_1}+a_{F_2}+a_{F_3}+a_{F_4})T_C + a_{F_1}T_{F_1} + a_{F_2}T_{F_2} + a_{F_3}T_{F_3} + a_{F_4}T_{F_4} \\
&= \dot{q}_C V_C
\end{aligned}
\tag{4.14}
$$

or more compactly

$$
a_C T_C + \sum_{F\sim NB(C)} a_F T_F = b_C \tag{4.15}
$$

where

$$
\begin{aligned}
a_C &= -\sum_{F\sim NB(C)} a_F = -(a_{F_1}+a_{F_2}+a_{F_3}+a_{F_4}) \\
b_C &= \dot{q}_C V_C
\end{aligned}
\tag{4.16}
$$

Equations similar to Eq. (4.15) may be derived for all cells in the domain, yielding a set of algebraic equations, which can be solved using a variety of direct or iterative methods. Focusing on element C in Fig. (4.13), Eq. (4.15) implies a relation

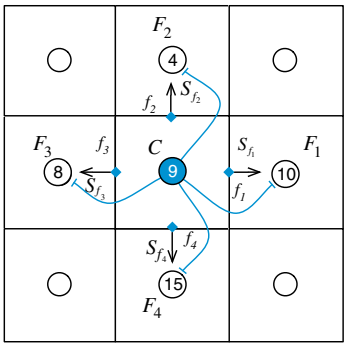


Fig. 4.13 Element notation

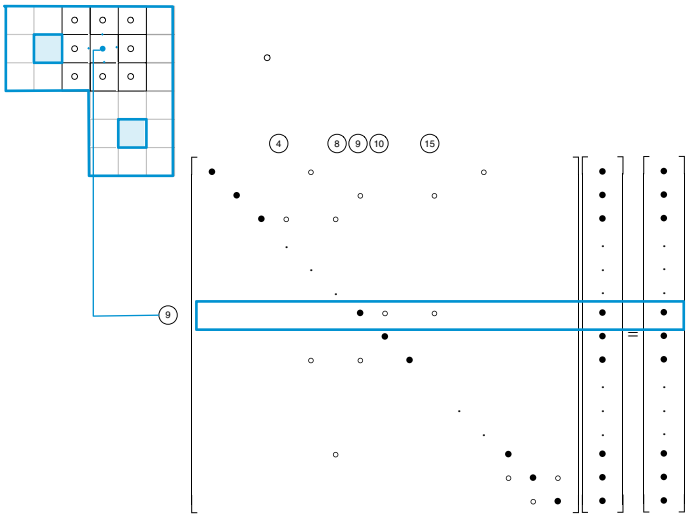


Fig. 4.14 System of equations

between T_C and the temperatures at its four neighbors namely T_{F_1}, T_{F_2}, T_{F_3}, and T_{F_4}, which in global assembly would be T_9, and T_{10}, T_4, T_8 and T_{15}.

Similar equations are also derived for boundary elements and their collection yields the set of equations illustrated in Fig. (4.14), which can be represented in matrix form as given in Eq. (4.2), where **A** is the matrix of coefficients, [*T*] the solution vector, and **b** is a vector composed of terms that cannot be included in **A**. The methodology to solve Eq. (4.2) is presented in the next section.

Finally it should be stated that the properties of the finite volume method as related to accuracy, robustness, and other characteristics will be reviewed in later chapters. This includes examining in more details the finite volume discretization of the diffusion term, which was presented above for a rectangular Cartesian grid.

4.1.5 Step IV: Solution of the Discretized Equations

The discretization of the differential equation results in a set of discrete algebraic equations, which must be solved to obtain the discrete values of *T*. The coefficients of these equations may be independent of *T* (i.e., linear) or dependent on *T* (i.e. non-linear). The techniques to solve this algebraic system of equations are independent of the discretization method, and represent the various trajectories that can be followed to obtain a solution. For the linear algebraic sets encountered in this book, the uniqueness of the solution is guaranteed. Therefore if the adopted solution method gives a solution, it will be the desired solution. All solution methods (i.e., all paths to solution) which arrive at a solution will give the same solution for the same set of discrete equations.

The solution methods for solving systems of algebraic equations may be broadly classified as direct or iterative and are briefly reviewed below.

4.1.5.1 Direct Methods

In a direct method the solution to the system of equations [e.g., Eq. (4.2)] is obtained by applying a relatively complex algorithm, in comparison with an iterative method, only once to obtain the solution for a given set of coefficients. An example of a direct method is matrix inversion whereby the solution is obtained as

$$[T] = \mathbf{A}^{-1}\mathbf{b} \tag{4.17}$$

Therefore a solution for $[T]$ is guaranteed if $\mathbf{A}^{-1}$ can be found. However, the operation count for the inversion of an $N \times N$ matrix is $O(N^3)$, which is computationally expensive. Consequently, inversion is almost never employed in practical problems. More efficient methods for linear systems are available. For the discretization methods of interest here, $\mathbf{A}$ is sparse, and for structured meshes it is banded. For certain types of equations (e.g., pure diffusion), the matrix is symmetric. Matrix manipulation can take into account the special structure of $\mathbf{A}$ in devising efficient solution techniques. Such methods will be reviewed in Chap. 10.

In general, direct methods are rarely used in computational fluid dynamics because of their large computational and storage requirements. Most industrial CFD problems today involve hundreds of thousands of cells, with 5–10 unknowns per cell even for simple problems. Thus the matrix $\mathbf{A}$ is usually very large, and most direct methods become impractical for these large problems. Furthermore, the matrix $\mathbf{A}$ is usually non-linear, so that the direct method must be embedded within an iterative loop to update nonlinearities in $\mathbf{A}$. Thus, the direct method is applied over and over again, making it all the more time-consuming.

4.1.5.2 Iterative Methods

Iterative methods follow a guess-and-correct procedure to gradually refine the estimated solution by repeatedly solving the discrete system of equations. Let us consider an extremely simple Gauss-Seidel iterative method. The overall solution loop for this method may be written as follows:

(a) Guess the discrete values of T at all grid elements in the domain.
(b) Visit each grid element in turn. Update T using

$$T_C = \frac{-\sum_{F \sim NB(C)} a_F T_F + b_C}{a_C} \tag{4.18}$$

The neighboring values are required for the update of T_C. These are assumed known at prevailing values. Thus, grid elements which have already been visited will have updated values of T and those that have not will have old values.

(c) Sweep the domain until all grid elements are covered. This completes one iteration.

(d) Check if an appropriate convergence criterion is met. The requirement, for example, could be that the maximum change in the grid-point values of T be less than 1 %. If the criterion is met, stop. Else, go back to step b and repeat.

The iteration procedure described here is not guaranteed to converge to a solution for arbitrary combinations of a_C and a_{NB}. Convergence of the process is guaranteed for linear problems if the *Scarborough criterion* is satisfied. The Scarborough criterion requires that a_C and a_{NB} should satisfy

$$\frac{-\sum_{F\sim NB(C)} a_F}{a_C} \begin{cases} \leq 1 & \text{for all grid points} \\ < 1 & \text{for at least one point} \end{cases} \tag{4.19}$$

Matrices which satisfy the Scarborough criterion have *diagonal dominance*.

The Gauss-Seidel scheme can be implemented with very little storage. All that is required is storage for the discrete values of T at the grid elements. The coefficients can be computed on the fly if desired, since the entire coefficient matrix for the domain is not required when updating the value of T at any grid point. Also, the iterative nature of the scheme makes it particularly suitable for non-linear problems. If the coefficients depend on T, they may be updated using prevailing values of T as iterations proceed. Nevertheless, the Gauss-Seidel scheme is rarely used in practice for solving the systems encountered in CFD. The rate of convergence of the scheme decreases to unacceptably low levels if the system of equations is large. In Chap. 10, an algebraic *multigrid method* will be used to accelerate the rate of convergence of iterative schemes and improve their performance.

4.1.6 Other Types of Fields

In addition to the element field introduced above, other fields are defined for different purposes. Two such fields include the face field and the vertex field that are briefly described below.

The **face field** consists of the array of values defined at the centre of the faces. As shown in Fig. 4.15, it defines a number of arrays for the interior faces and the various patch faces. The face field is used, for example, to define the face mass fluxes for use when solving advective and flow problems.

The **vertex field** schematically depicted in Fig. 4.16 stores variables at the vertices; these again are grouped into interior vertices and patch vertices. Vertex fields are usually used for post processing, and in some cases for gradient computation.

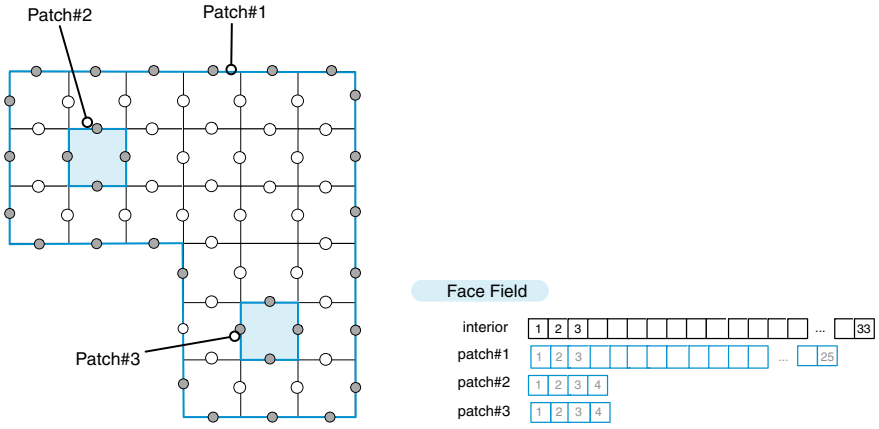


Fig. 4.15 Face field

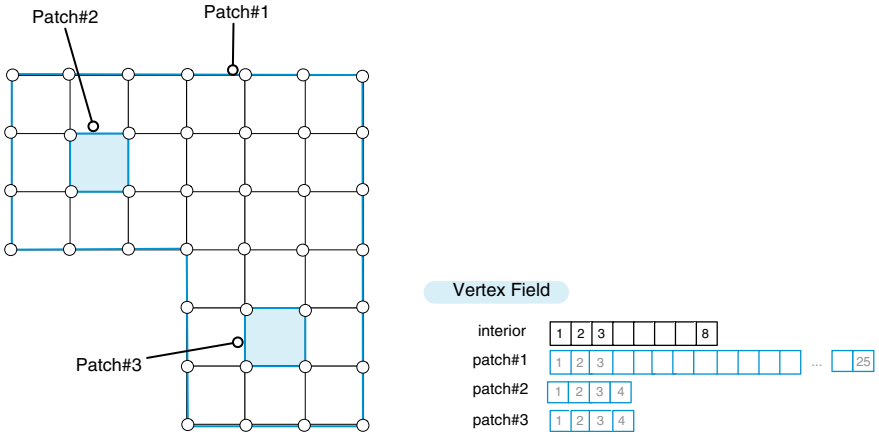


Fig. 4.16 Vertex field

4.2 Closure

This chapter overviewed the discretization process, underlining on the way the basic ingredients needed for the development of a CFD code. The coming chapters will take each of these ingredients and dissect it, while developing the "uFVM" and learning about the industrial open source CFD library, denoted by OpenFOAM®. The next two chapters will further expand on the idea of finite volume mesh and finite volume discretization.

Chapter 5
The Finite Volume Method

Abstract Similar to other numerical methods developed for the simulation of fluid flow, the finite volume method transforms the set of partial differential equations into a system of linear algebraic equations. Nevertheless, the discretization procedure used in the finite volume method is distinctive and involves two basic steps. In the first step, the partial differential equations are integrated and transformed into balance equations over an element. This involves changing the surface and volume integrals into discrete algebraic relations over elements and their surfaces using an integration quadrature of a specified order of accuracy. The result is a set of semi-discretized equations. In the second step, interpolation profiles are chosen to approximate the variation of the variables within the element and relate the surface values of the variables to their cell values and thus transform the algebraic relations into algebraic equations. The current chapter details the first discretization step and presents a broad review of numerical issues pertaining to the finite volume method. This provides a solid foundation on which to expand in the coming chapters where the focus will be on the discretization of the various parts of the general conservation equation. In both steps, the selected approximations affect the accuracy and robustness of the resulting numerics. It is therefore important to define some guiding principles for informing the selection process.

5.1 Introduction

The popularity of the Finite Volume Method (FVM) [1–3] in Computational Fluid Dynamics (CFD) stems from the high flexibility it offers as a discretization method. Though it was preceded for many years by the finite difference [4, 5] and finite element methods [6], the FVM assumed a particularly prominent role in the simulation of fluid flow problems and related transport phenomena as a result of the work done by the CFD group at Imperial College in the early 70 s under the direction of Professor Spalding [7], with such contributors as Patankar [8], Gosman [9], and Runchal [10, 11] to cite a few. The FVM owes much of its flexibility and popularity to the fact that discretization is carried out directly in the physical space with no need for any transformation between the physical and the computational

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_5

coordinate system. Furthermore its adoption of a collocated arrangement [12] made it suitable for solving flows in complex geometries. These developments have expanded the applicability of the FVM to encompass a wide range of applications while retaining the simplicity of its mathematical formulation. Another important aspect of the FVM is that its numerics mirrors the physics and the conservation principles it models, such as the integral property of the governing equations, and the characteristics of the terms it discretizes. In what follows the semi-discretized form of a general scalar equation is derived. Then the properties required from the discretization method are discussed along with some guiding principles. The chapter ends with a discussion of a number of issues pertinent to the FVM. The transformation of the semi-discretized equation into algebraic equations will be the subject of a number of chapters to follow.

5.2 The Semi-Discretized Equation

In step 1 of the finite volume discretization process, the governing equations are integrated over the elements (or finite volumes) into which the domain has been subdivided, then the Gauss theorem is applied to transform the volume integrals of the convection and diffusion terms into surface integrals. Following this step, the surface and volume integrals are transformed into discrete ones and integrated numerically through the use of integration points (ip). To clarify this approach and to develop an adequate appreciation for the subtleties of the advanced discretization schemes discussed in later sections, the following example illustrates the application of the technique for a two-dimensional transport problem.

As presented in Chap. 3, the conservation equation for a general scalar variable ϕ can be expressed as

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient term}} + \underbrace{\nabla\cdot(\rho\mathbf{v}\phi)}_{\text{convective term}} = \underbrace{\nabla\cdot\left(\Gamma^{\phi}\nabla\phi\right)}_{\text{diffusion term}} + \underbrace{Q^{\phi}}_{\text{source term}} \tag{5.1}$$

The steady-state form of the above equation is obtained by dropping the transient term and is given by

$$\nabla\cdot(\rho\mathbf{v}\phi) = \nabla\cdot\left(\Gamma^{\phi}\nabla\phi\right) + Q^{\phi} \tag{5.2}$$

By integrating the above equation over the element C shown in Fig. 5.1; Eq. (5.2) is transformed to

$$\int_{V_C} \nabla\cdot(\rho\mathbf{v}\phi)dV = \int_{V_C} \nabla\cdot\left(\Gamma^{\phi}\nabla\phi\right)dV + \int_{V_C} Q^{\phi}dV \tag{5.3}$$

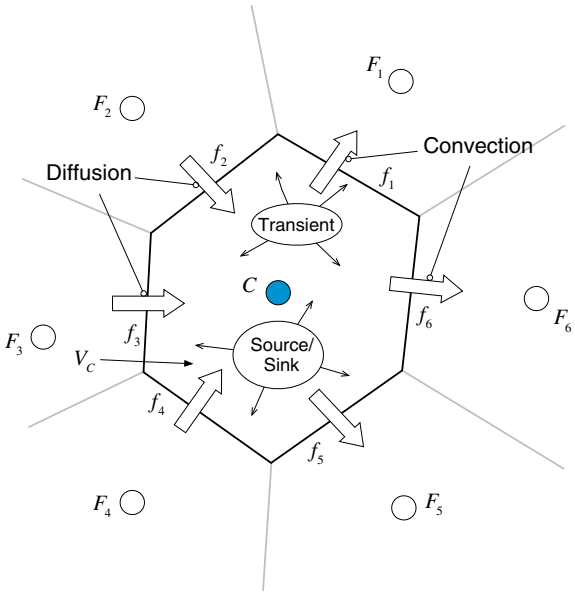


Fig. 5.1 Conservation in a discrete element

Replacing the volume integrals of the convection and diffusion terms by surface integrals through the use of the divergence theorem, the above equation becomes

$$\oint_{\partial V_C} (\rho \mathbf{v}\phi) \cdot d\mathbf{S} = \oint_{\partial V_C} \left(\Gamma^{\phi} \nabla \phi\right) \cdot d\mathbf{S} + \int_{V_C} Q^{\phi} dV \tag{5.4}$$

where **bold** letters indicate vectors, $(\cdot)$ is the dot product operator, Q^{ϕ} represents the source term, $\mathbf{S}$ the surface vector, $\mathbf{v}$ the velocity vector, ϕ the conserved quantity, and $\oint_{\partial V_C}$ the surface integral over the volume V_C.

5.2.1 Flux Integration Over Element Faces

Denoting the convection and diffusion flux terms by $\mathbf{J}^{\phi,C}$ and $\mathbf{J}^{\phi,D}$, respectively, their expressions are given by

$$\mathbf{J}^{\phi,C} = \rho \mathbf{v} \phi \tag{5.5}$$

$$\mathbf{J}^{\phi,D} = -\Gamma^{\phi} \nabla \phi \tag{5.6}$$

Further, defining the total flux $\mathbf{J}^{\phi}$ as the sum of the convection and diffusion fluxes, it can be written as

$$\mathbf{J}^{\phi} = \mathbf{J}^{\phi,C} + \mathbf{J}^{\phi,D} \tag{5.7}$$

Replacing the surface integral over cell C by a summation of the flux terms over the faces of element C, the surface integrals of the convection, diffusion, and total fluxes become

$$\oint_{\partial V_C} \mathbf{J}^{\phi,C} \cdot d\mathbf{S} = \sum_{f \sim faces(V_C)} \left(\int_f (\rho \mathbf{v} \phi) \cdot d\mathbf{S} \right) \tag{5.8}$$

$$\oint_{\partial V_C} \mathbf{J}^{\phi,D} \cdot d\mathbf{S} = \sum_{f \sim faces(V_C)} \left(\int_f \left(\Gamma^{\phi} \nabla \phi \right) \cdot d\mathbf{S} \right) \tag{5.9}$$

$$\oint_{\partial V_C} \mathbf{J}^{\phi} \cdot d\mathbf{S} = \sum_{f \sim faces(V_C)} \left(\int_f \mathbf{J}_f^{\phi} \cdot d\mathbf{S} \right) \tag{5.10}$$

In Eqs. (5.8)–(5.10) the surface fluxes are evaluated at the faces of the element rather than integrated within it. This transformation has important consequences on the properties of the FVM, one of which is that it renders the method conservative, as will be discussed later.

To proceed further with the discretization, the surface integral at each face of the element in addition to the volume integral of the source term have to be evaluated. Using a Gaussian quadrature the integral at the face f of the element becomes

$$\int_f \mathbf{J}^{\phi} \cdot d\mathbf{S} = \int_f \left(\mathbf{J}^{\phi} \cdot \mathbf{n} \right) dS = \sum_{ip \sim ip(f)} \left(\mathbf{J}^{\phi} \cdot \mathbf{n} \right)_{ip} \omega_{ip} S_f \tag{5.11}$$

where ip refers to an integration point and $ip(f)$ the number of integration points along surface f. As seen in Fig. 5.2, a number of options are available with their accuracy depending on the number of integration points used and the weighing function ω_{ip}. For a simple mean value integration (Fig. 5.2a), also known as the trapezoidal rule, only one integration point located at the centroid of the face is used with a weighing function of value equal to 1 (i.e., $ip = \omega_{ip} = 1$). This approximation is second order accurate and is applicable in two and three dimensions. Another option (Fig. 5.2b) in two dimensions, which is third order accurate, involves two integration points ($ip = 2$) positioned at $\xi_1 = (3 - \sqrt{3})/6$ and $\xi_2 = (3 + \sqrt{3})/6$ where ξ is distance along the face measured from one end and normalized by the total length, with weights $\omega_1 = \omega_2 = 1/2$. A third option depicted in Fig. 5.2c uses

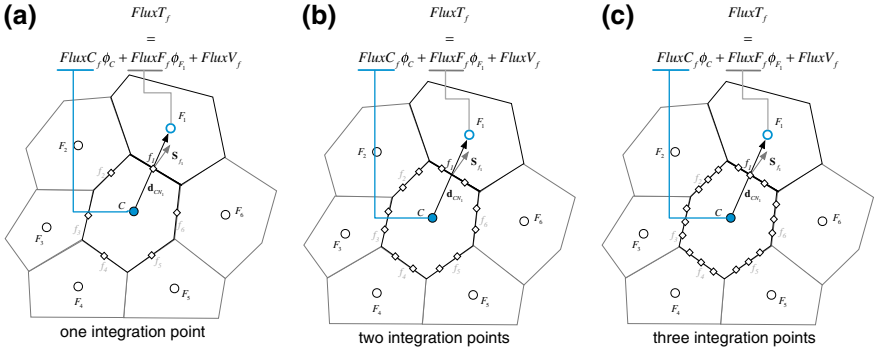


Fig. 5.2 Surface integration of fluxes using **a** one integration point, **b** two integration points, and **c** three integration points

three integration points (ip = 3) positioned at $\xi_1 = (5 - \sqrt{15})/10$, $\xi_2 = 1/2$, and $\xi_3 = (5 + \sqrt{15})/10$, with weights ω_1 = 5/18, ω_2 = 4/9, and ω_3 = 5/18. It is clear that the computational cost rises with the number of integration points used in the approximation. With $ip(f)$ denoting the number of integration points along face f, the general discretized relations for the convection and diffusion terms become

$$\oint_{\partial V_C} (\rho \mathbf{v}\phi) \cdot d\mathbf{S} = \sum_{f \sim faces(V)} \sum_{ip \sim ip(f)} \left(\omega_{ip} (\rho \mathbf{v}\phi)_{ip} \cdot \mathbf{S}_f \right) \tag{5.12}$$

$$\oint_{\partial V_C} \left(-\Gamma^{\phi} \nabla \phi \right) \cdot d\mathbf{S} = \sum_{f \sim faces(V)} \sum_{ip \sim ip(f)} \left(\omega_{ip} \left(-\Gamma^{\phi} \nabla \phi \right)_{ip} \cdot \mathbf{S}_f \right) \tag{5.13}$$

5.2.2 Source Term Volume Integration

Volume integration is used for the source term. Adopting a Gaussian quadrature integration, the volume integral of the source term is computed as

$$\int_V Q^{\phi} dV = \sum_{ip \sim ip(V)} \left(Q^{\phi}_{ip} \omega_{ip} V \right) \tag{5.14}$$

As with surface flux integration, Fig. 5.3 shows different options for volume integration with their accuracy depending on the number of integration points used (ip) and the weighing function ω_{ip}.

For one point Gauss integration (Fig. 5.3a), $ip = \omega_{ip} = 1$ with the integration point located at the centroid of the element. This approximation is second order accurate and is applicable in two and three dimensions. In two dimensions, four point gauss

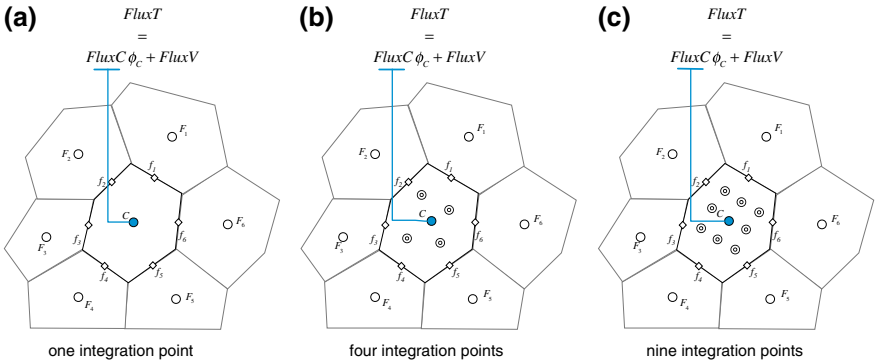


Fig. 5.3 Volume integration of source terms using **a** one integration point, **b** four integration points, and **c** nine integration points

integration (Fig. 5.3b) involves the use of four integration points. The weights are computed as the product of the one dimensional weights and the integration points (ξ, η) are obtained from the one dimensional profiles. Therefore the function is calculated at $\left[\left(3-\sqrt{3}\right)/6, \left(3+\sqrt{3}\right)/6\right]$, $\left[\left(3+\sqrt{3}\right)/6, \left(3+\sqrt{3}\right)/6\right]$, $\left[\left(3-\sqrt{3}\right)/6, \left(3-\sqrt{3}\right)/6\right]$, and $\left[\left(3+\sqrt{3}\right)/6, \left(3-\sqrt{3}\right)/6\right]$ with the weights being equal (ω_{ip} = 1/4 *for ip* = 1 *to* 4). The nine point gauss integration method (Fig. 5.3c) involves the use of nine integration points. The accuracy increases with increasing the number of integration points but so does the computational cost.

5.2.3 The Discrete Conservation Equation for One Integration Point

While the above terms can be discretized with any specified number of integration points, it is customary for the finite volume method to use one integration point, yielding second order accuracy. This was found to be a good compromise between accuracy and flexibility while keeping the method simple and relatively of low computational cost. Following the mid-point integration approximation, the semi-discrete steady state finite volume equation for element C shown in Fig. 5.4 can be finally simplified to

$$\sum_{f\sim nb(C)} \left(\rho \mathbf{v}\phi - \Gamma^{\phi}\nabla\phi\right)_f \cdot \mathbf{S}_f = Q_C^{\phi} V_C \tag{5.15}$$

The aim of the second stage of the discretization process is to transform Eq. (5.15) into an algebraic equation by expressing the face and volume fluxes in terms of the values of the variable at the neighboring cell centers. This **linearization of the fluxes** is at the core of the second discretization step.

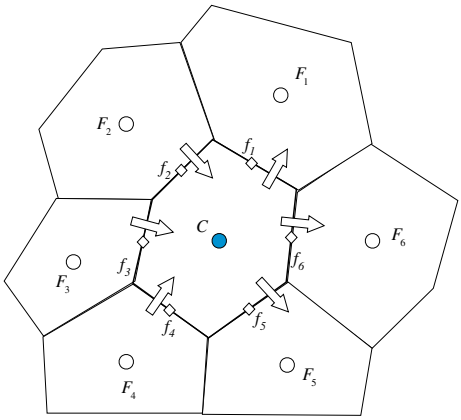


Fig. 5.4 Fluxes at element surfaces

5.2.4 Flux Linearization

As schematically depicted in Fig. 5.2a, the face flux can be split into a linear part, function of the ϕ values at the nodes straddling the face (*i.e.,* ϕ_C *and* ϕ_F), and a non-linear part, which includes the remaining portion that cannot be expressed in terms of ϕ_C and ϕ_F. The resulting equation can be written as

$$\mathbf{J}_f^{\phi} \cdot \mathbf{S}_f = \underbrace{FluxT_f}_{\substack{\text{total flux} \\ \text{for face } f}} = \underbrace{FluxC_f}_{\substack{\text{flux linearization} \\ \text{coefficient for } C}} \phi_C + \underbrace{FluxF_f}_{\substack{\text{flux linearization} \\ \text{coefficient for } F}} \phi_F + \underbrace{FluxV_f}_{non-linearized\ part} \tag{5.16}$$

where $FluxT_f$ represents the total flux through face f, and is decomposed into three terms. The first two terms represent the contributions of the two elements sharing the face and are written via the linearization coefficients $FluxC_f$ and $FluxF_f$. The last term describes the nonlinear contribution that cannot be expressed in terms of ϕ_C and ϕ_F and is given by the non-linear term $FluxV_f$. The values of $FluxC_f$, $FluxF_f$, and $FluxV_f$ obviously depend on the discretized term and the scheme used for its discretization.

The flux linearization is thus obtained by substituting Eq (5.16) into the left hand side of Eq. (5.15). Repeating for all cell faces yields

$$\begin{aligned} \sum_{f \sim nb(C)} \left(\mathbf{J}_f^{\phi} \cdot \mathbf{S}_f \right) &= \sum_{f \sim nb(C)} \left(FluxT_f \right) \\ &= \sum_{f \sim nb(C)} \left(FluxC_f\, \phi_C + FluxF_f\, \phi_F + FluxV_f \right) \end{aligned} \tag{5.17}$$

The linearization of the volume flux is performed, as shown in Fig. 5.3a, by expressing it as a linear function of the element node value ϕ_C and is given by

$$\begin{aligned} Q_C^{\phi} V_C &= FluxT \\ &= FluxC\ \phi_C + FluxV \end{aligned} \tag{5.18}$$

In the case of a constant source term, the volume flux, which represents the right hand side of Eq. (5.15), reduces to

$$\begin{aligned} FluxC &= 0 \\ FluxV &= Q_C^{\phi} V_C \end{aligned} \tag{5.19}$$

Substitution of Eqs. (5.17) and (5.18) in Eq. (5.15), yields the sought after algebraic relation as

$$a_C \phi_C + \sum_{F \sim NB(C)} (a_F \phi_F) = b_C \tag{5.20}$$

where the relations between equation coefficients and flux linearization coefficients are expressed as

$$\begin{aligned} a_C &= \sum_{f \sim nb(C)} FluxC_f - FluxC \\ a_F &= FluxF_f \\ b_C &= -\sum_{f \sim nb(C)} FluxV_f + FluxV \end{aligned} \tag{5.21}$$

Example 1

Find the linearization coefficients for the discretization of the convection term when the velocity field is in the positive direction using the approximation $\phi_f = \phi_C$ *(this is known as the upwind scheme).*

Solution

$$\mathbf{J}_f^{\phi} = (\rho \mathbf{v} \phi)_f$$

thus

$$\mathbf{J}_f^{\phi} \cdot \mathbf{S}_f = (\rho \mathbf{v} \phi)_f \cdot \mathbf{S}_f = \left(\rho_f \mathbf{v}_f \cdot \mathbf{S}_f\right) \phi_f = \dot{m}_f \phi_f$$

With $\phi_f = \phi_C$, the coefficients in the total flux equation are obtained as
$FluxC_f = \dot{m}_f$
$FluxF_f = 0$
$FluxV_f = 0$
and the total flux is expressed as
$FluxT_f = \dot{m}_f \phi_C + 0\phi_F + 0$

5.3 Boundary Conditions

The evaluation of the fluxes at the faces of a domain boundary does not require, in general, a profile assumption. Rather a direct substitution is usually performed. The type of boundary conditions are numerous. However, two of the most widely used ones for general scalars are the Dirichlet and the Neumann boundary conditions. In mathematical terms these are respectively a value specified (or a first type) and a flux specified (or a second type) boundary condition.

5.3.1 Value Specified (Dirichlet Boundary Condition)

Consider the case where some scalar ϕ is being convected through an inlet. Assuming the diffusion of ϕ to be negligible, the boundary condition can be expressed as

$$\phi_b = \phi_{b,specified} \tag{5.22}$$

For the boundary face shown in Fig. 5.5, the boundary flux is evaluated using the known value of ϕ_b given by Eq. (5.22). Therefore the value of the boundary flux is not unknown, rather it can be directly evaluated as

$$\begin{aligned} \mathbf{J}_b^{\phi} \cdot \mathbf{S}_b &= \mathbf{J}_b^{\phi,C} \cdot \mathbf{S}_b \\ &= (\rho \mathbf{v} \phi)_b \cdot \mathbf{S}_b \\ &= FluxC_b \phi_C + FluxV_b \\ &= (\rho_b \mathbf{v}_b \cdot \mathbf{S}_b)\phi_b = \dot{m}_f \phi_{b,specified} \end{aligned} \tag{5.23}$$

Thus

$$\begin{aligned} FluxC_b &= 0 \\ FluxV_b &= \dot{m}_f \phi_{b,specified} \end{aligned} \tag{5.24}$$

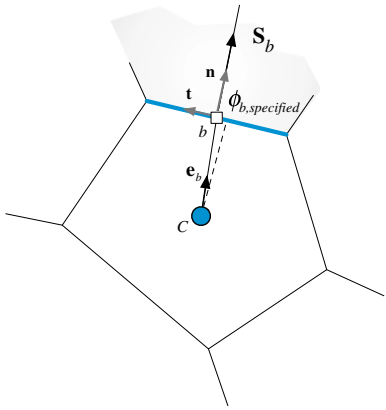


Fig. 5.5 Dirichlet boundary condition

5.3.2 *Flux Specified (Neumann Boundary Condition)*

Considering the case shown in Fig. 5.6 where the boundary face of the element C represents physically a wall where a flux for the quantity ϕ is specified. Mathematically this is equivalent to writing

$$\begin{aligned} \mathbf{J}_b^{\phi} \cdot \mathbf{S}_b &= \underbrace{\mathbf{J}_b^{\phi} \cdot \mathbf{n}_b}_{specified\,flux} S_b \\ &= q_{b,specified} S_b \end{aligned} \tag{5.25}$$

In the above equation $q_{b,specified}$ is a known quantity specified by the user, representing the flux per unit area.

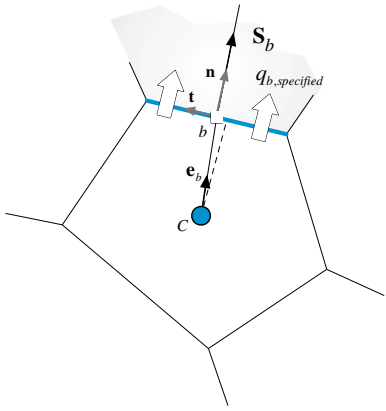


Fig. 5.6 Neumann boundary condition

Thus

$$\begin{aligned} FluxC_b &= 0 \\ FluxV_b &= q_{b,specified} S_b \end{aligned} \tag{5.26}$$

The treatment of these two boundary conditions and others will be detailed in the following chapters along with the treatment of the terms related to step two discretization.

5.4 Order of Accuracy

As discussed earlier, fluxes at the faces and sources over the element are evaluated following the mean value approach, i.e., using the value at the centroid of the surface (midpoint rule) and cell, respectively. This treatment, in addition to the assumed variation of ϕ in space around point C, i.e., $\phi = \phi(\mathbf{x})$, determine the accuracy of the discretization procedure. In the adopted method, ϕ is assumed to vary linearly in space, i.e.,

$$\phi(\mathbf{x}) = \phi_C + (\mathbf{x} - \mathbf{x}_C) \cdot (\nabla\phi)_C \text{ where } \phi_C = \phi(\mathbf{x}_C) \tag{5.27}$$

5.4.1 Spatial Variation Approximation

The spatial variation of the variable $\phi = \phi(\mathbf{x})$ within the element shown in Fig. 5.7 can be described via a Taylor series expansion around point $\mathbf{x}_C$ as

$$\begin{aligned} \phi(\mathbf{x}) = \phi_C &+ (\mathbf{x} - \mathbf{x}_C) \cdot (\nabla\phi)_C + \frac{1}{2}(\mathbf{x} - \mathbf{x}_C)^2 \colon (\nabla\nabla\phi)_C \\ &+ \frac{1}{3!}(\mathbf{x} - \mathbf{x}_C)^3 :: (\nabla\nabla\nabla\phi)_C + \ldots\ldots\ldots\ldots\ldots \\ &+ \frac{1}{n!}(\mathbf{x} - \mathbf{x}_C)^n \underbrace{:: \ldots ::}_{(n-1)times} \left(\underbrace{\nabla\nabla \ldots \nabla\phi}_{n\ times} \right)_C + \ldots\ldots\ldots\ldots\ldots \end{aligned} \tag{5.28}$$

The expression $(\mathbf{x} - \mathbf{x}_C)^n$ in the equation and consequent ones represents the nth tensorial product of the vector $(\mathbf{x} - \mathbf{x}_C)$ with itself, producing an nth rank tensor. The operator (:) is the inner product of two 2nd rank tensors, (::) is the inner product of two 3rd rank tensors, and more generally "$\underbrace{:: \ldots ::}_{(n-1)times}$" is the inner product of two nth rank tensors, all yielding a scalar.

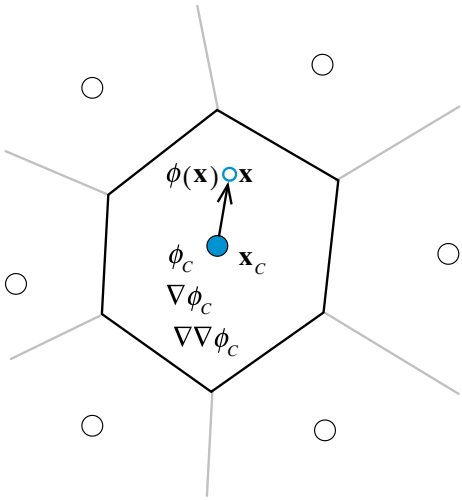


Fig. 5.7 Variation within an element

Comparison between the assumed variation given by Eq. (5.27) and the Taylor series expansion [Eq. (5.28)] indicates an error proportional to $|(\mathbf{x} - \mathbf{x}_C)^2|$ and implying a second order spatial accuracy.

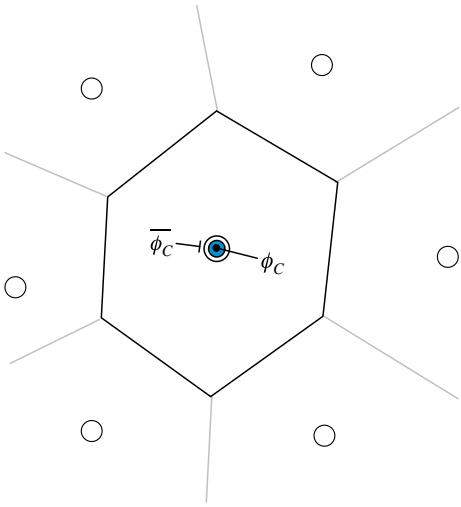
5.4.2 Mean Value Approximation

The accuracy of the mean value approximation can be derived by integrating the variable $\phi(\mathbf{x})$ over the cell of centroid C depicted in Fig. 5.8 and leading to

$$\begin{aligned}
\overline{\phi}_C &= \frac{1}{V_C}\int_{V_C} \phi\, dV \\
&= \frac{1}{V_C}\int_{V_C} \left[\phi_C + (\mathbf{x}-\mathbf{x}_C)\cdot(\nabla\phi)_C + O\left(|\mathbf{x}-\mathbf{x}_C|^2\right)\right] dV \\
&= \frac{\phi_C}{V_C}\int_{V_C} dV + \frac{1}{V_C}\int_{V_C} (\mathbf{x}-\mathbf{x}_C)\cdot(\nabla\phi)_C\, dV + \frac{1}{V_C}\int_{V_C} O\left(|\mathbf{x}-\mathbf{x}_C|^2\right) dV \\
&= \phi_C + O\left(|\mathbf{x}-\mathbf{x}|^2\right)
\end{aligned} \tag{5.29}$$

where V_C is the volume of the element. The second term is equal to zero because point C is the centroid of the element. Neglecting the last term in the equation introduces an error of order 2, demonstrating that the mean value approximation is second order accurate.

Fig. 5.8 Mean-value theorem



The convective flux at face f of element C shown in Fig. 5.9 is computed as

$$\begin{aligned}\overline{(\rho_f \mathbf{v}_f \cdot \mathbf{S}_f)\phi_f} &= \int_f (\rho \mathbf{v}\phi) \cdot d\mathbf{S} \\ &= \int_f \rho_f \mathbf{v}_f \left[\phi_f + (\mathbf{x} - \mathbf{x}_f) \cdot (\nabla\phi)_f + O\left(|\mathbf{x} - \mathbf{x}_f|^2\right)\right] \cdot d\mathbf{S} \\ &= \rho_f \mathbf{v}_f \phi_f \cdot \int_f d\mathbf{S} + \int_f (\mathbf{x} - \mathbf{x}_f) \cdot (\nabla\phi)_f \rho_f \mathbf{v}_f \cdot d\mathbf{S} + \int_f O\left(|\mathbf{x} - \mathbf{x}_f|^2\right) \rho_f \mathbf{v}_f \cdot d\mathbf{S} \\ &= (\rho_f \mathbf{v}_f \cdot \mathbf{S}_f)\left[\phi_f + O\left(|\mathbf{x} - \mathbf{x}_f|^2\right)\right]\end{aligned} \tag{5.30}$$

where again the second term is equal to zero as $\mathbf{x}_f$ is the centroid of the respective element surface. Here subscript f denotes the value of the variable, in this case ϕ, at the face centroid and $\mathbf{S}$ is the outward pointing face surface vector.

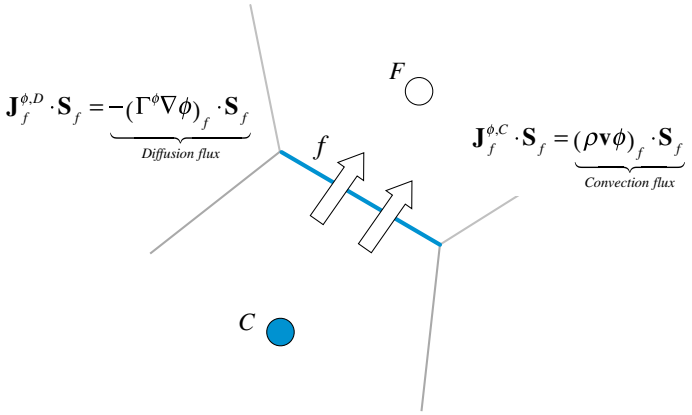


Fig. 5.9 Convection and diffusion fluxes at element faces

For the diffusive flux, truncating second order terms and higher, the following can be written:

$$\begin{aligned}\int_f (\Gamma^\phi \nabla\phi)\cdot d\mathbf{S} &= \int_f \left[(\Gamma^\phi\nabla\phi)_f + (\mathbf{x}-\mathbf{x}_f)\cdot(\nabla(\Gamma^\phi\nabla\phi))_f + O\left(|\mathbf{x}-\mathbf{x}_f|^2\right)\right]\cdot d\mathbf{S} \\ &= (\Gamma^\phi\nabla\phi)_f \cdot \int_f d\mathbf{S} + \left[\int_f (\mathbf{x}-\mathbf{x}_f)d\mathbf{S}\right] : (\nabla(\Gamma^\phi\nabla\phi))_f + O\left(|\mathbf{x}-\mathbf{x}_f|^2\right) \\ &= (\Gamma^\phi\nabla\phi)_f \cdot \mathbf{S}_f + O\left(|\mathbf{x}-\mathbf{x}_f|^2\right)\end{aligned} \tag{5.31}$$

Hence for a second order method, the surface integral and the variation of ϕ within the element should be second order accurate.

Higher order accuracy can be achieved by increasing the order of accuracy of the surface integral or of the assumed profile for ϕ. One method developed by Lilek and Peric [13] in two dimensions used a fourth order accurate surface integral, evaluated via Simpson's rule as

$$\int_f \phi d\mathbf{S} = \left(\frac{\phi_{ip_1} + 4\phi_{ip_2} + \phi_{ip_3}}{6}\right)\mathbf{S}_f \tag{5.32}$$

As shown in Fig. 5.10, for face f the value of ϕ is needed at three integration points, i.e., the surface centroid ip_2, and the vertices of the face ip_1 and ip_3.

To obtain a formally fourth order accurate discretization, the assumed profile of ϕ within the element should also be fourth order accurate such that

$$\phi(\mathbf{x}) = \phi_C + (\mathbf{x}-\mathbf{x}_C)\cdot(\nabla\phi)_C + \frac{1}{2}(\mathbf{x}-\mathbf{x}_C)^2 : (\nabla\nabla\phi)_C + \cdots \tag{5.33}$$

In this case the accuracy of the gradient calculation has to be second order or higher and that of the Hermitian first order or higher.

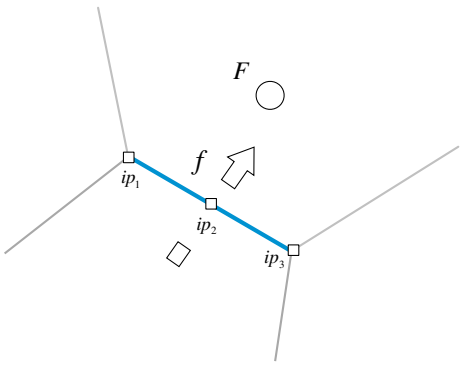


Fig. 5.10 A element face f showing the integration points where values are needed for a higher order approximation of the surface integral

5.5 Transient Semi-Discretized Equation

For unsteady state the temporal term in Eq. (5.1) is retained and in addition to integrating over the volume of the element, integration over time is also needed. In this case the integrated equation becomes

$$\begin{aligned}\int_t^{t+\Delta t}\int_{V_C}\frac{\partial(\rho\phi)}{\partial t}dVdt+\int_t^{t+\Delta t}\left[\sum_{f\sim nb(C)}\left(\int_f(\rho\mathbf{v}\phi)_f\cdot d\mathbf{S}\right)\right]dt\\ -\int_t^{t+\Delta t}\left[\sum_{f\sim nb(C)}\left(\int_f(\Gamma\nabla\phi)_f\cdot d\mathbf{S}\right)\right]dt\\ =\int_t^{t+\Delta t}\left[\int_{V_C}Q^\phi dV\right]dt\end{aligned}\tag{5.34}$$

Further simplification of this equation requires a choice with regard to the time integration accuracy required. This will be dealt with in Chap. 13. For fixed grids, where the volume and surface of each element are constant in time, the first term can be integrated as

$$\int_t^{t+\Delta t}\int_{V_C}\frac{\partial(\rho\phi)}{\partial t}dV\,dt=\int_t^{t+\Delta t}\frac{\partial}{\partial t}\left(\int_{V_C}\rho\phi dV\right)dt=\int_t^{t+\Delta t}\frac{\partial\left(\overline{\rho\phi}\right)}{\partial t}V_C\,dt\tag{5.35}$$

where

$$\overline{\rho\phi}_C=\frac{1}{V_C}\int_{V_C}\rho\phi dV=(\rho\phi)_C+O\left(\Delta^2\right)\tag{5.36}$$

Substituting back, Eq. (5.34) reduces to

$$\begin{aligned}\int_t^{t+\Delta t}\frac{\partial(\rho\phi)}{\partial t}V_C dt+\int_t^{t+\Delta t}\left[\sum_{f\sim nb(C)}\left(\int_f(\rho\mathbf{v}\phi)_f\cdot d\mathbf{S}\right)\right]dt\\ -\int_t^{t+\Delta t}\left[\sum_{f\sim nb(C)}\left(\int_f(\Gamma\nabla\phi)_f\cdot d\mathbf{S}\right)\right]dt\\ =\int_t^{t+\Delta t}\left[\int_{V_C}Q^\phi dV\right]dt\end{aligned}\tag{5.37}$$

and using the midpoint rule, Eq. (5.37) becomes

$$\int_t^{t+\Delta t} \frac{\partial(\rho\phi)}{\partial t} V_C dt + \int_t^{t+\Delta t} \left[\sum_{f\sim nb(C)} (\rho \mathbf{v}\phi)_f \cdot \mathbf{S}_f \right] dt - \int_t^{t+\Delta t} \left[\sum_{f\sim nb(C)} (\Gamma \nabla\phi)_f \cdot \mathbf{S}_f \right] dt = \int_t^{t+\Delta t} Q_C^{\phi} V_C dt \tag{5.38}$$

Going forward beyond this point requires some assumptions as to how the variable is changing in time.

5.6 Properties of the Discretized Equations

As the size of the element tends to zero, the numerical solution is expected to be the exact solution of the general conservation equation [e.g., Eq. (5.2)] irrespective of the interpolation profile used to evaluate the element ϕ values. However, since finite volumes are used, it is crucial for the discretized equations to possess some properties in order to ensure a meaningful solution field. These properties are discussed next.

5.6.1 Conservation

From a physical point of view it is very important for the transported variables, which are generally conservative quantities (e.g., mass, energy, etc.), to be conserved in the discretized solution domain too, otherwise results may be unrealistic. This property is inherent to the FVM because the fluxes integrated at an element face are based on the values of the elements sharing the face [8]. Thus for any surface common to two elements, the flux leaving the face of one element will be exactly equal to the flux entering the other element through that same face. Thus these fluxes are of equal magnitudes but of opposite signs (Fig. 5.11). Any method that possesses this property is said to be conservative.

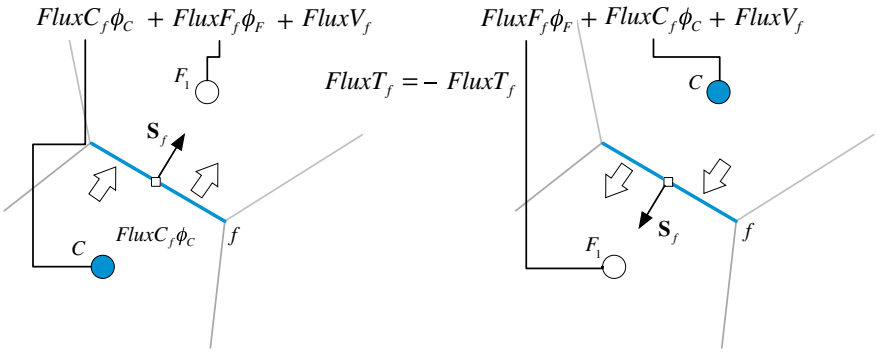


Fig. 5.11 Fluxes on neighboring elements

5.6.2 Accuracy

Accuracy refers to how close a numerical solution is to the exact solution. However, in most of the cases the exact solution for the problem to be solved is unknown. Therefore a direct comparison to check accuracy is not possible. An alternative is to consider the truncation error as a measure of accuracy. The error associated with the first step discretization of the various terms presented above was $O|(\mathbf{x} - \mathbf{x}_f)|^2$, which represents a second order accuracy. This means that if the number of grid points is doubled then the discretization error will be reduced by a factor of 4. The truncation error of a discretization scheme is the largest truncation error of each of the individual terms in the discretized equation. The discretization error does not give the value of the error on a certain grid. It does tell, however, how fast the error will decrease with grid refinement. The higher the order of the error the faster it will decrease with mesh refinement.

5.6.3 Convergence

The nonlinear nature of the conservation equations dealt with here necessitates an iterative approach. Starting with an initial guess, solutions are obtained by repeatedly applying a solution algorithm with the solution at the end of an iteration used as an initial guess for the following iteration. Ideally a solution is said to be converged when it does not change any further as iterations progress. In practice, however, a solution is established converged when changes between two consecutive iterations fall below a vanishing quantity ε. In general convergence is used to indicate the obtainment of a solution with any method. Sometimes the term "convergence" is used to indicate the attainment of a grid independent solution, i.e., a solution that does not change with any further grid refinement.

5.6.4 Consistency

A solution to an algebraic equation approximating a given partial differential equation is said to be ***consistent*** if, at each point in the solution domain, the numerical solution approaches the exact solution of the partial differential equation as the time step and grid spacing tend to zero, i.e., as the discretization error approaches zero. For this to be true, the discretization error should be some power of Δt and/or $\Delta \mathbf{x}$. If the discretization error is expressed in terms of $\Delta \mathbf{x}/\Delta t$ then, for consistency, $\Delta \mathbf{x}$ should tend to zero at a faster rate than Δt.

5.6.5 Stability

Stability refers to the behavior of the discretized equations to be resolved by an iterative solver. It indicates whether the resulting system of algebraic equations can be solved under a variety of initial and boundary conditions. In this sense stability is not really a property of the discretization process but rather a property of the resulting system of equations. As mentioned in a previous chapter, a sufficient condition for a system of linear equations to be stable and converge to a solution is for it to satisfy the Scarborough criterion, i.e., for its matrix of coefficients to be diagonally dominant.

For transient problems, a stable numerical scheme keeps the error in the solution bounded as time marching proceeds. The use of explicit or implicit transient schemes has direct impact on the stability of the numerical method. The stability of explicit methods is ensured by limiting the size of the time step. On the other hand, the stability of implicit methods can be enhanced by under-relaxing the discretized set of algebraic equations either through the use of under relaxation factors or by applying the false-transient approach, as described in a later chapter.

5.6.6 Economy

Economy is an important consideration in the development and application of CFD codes. It is understood to mean that the time required to compute realistic flow problems should not be prohibitive.

5.6.7 Transportiveness

The directional properties exhibited by fluid transport are well known and are signaled by the change in the type of the transport scalar equation [Eq. (5.2)], which

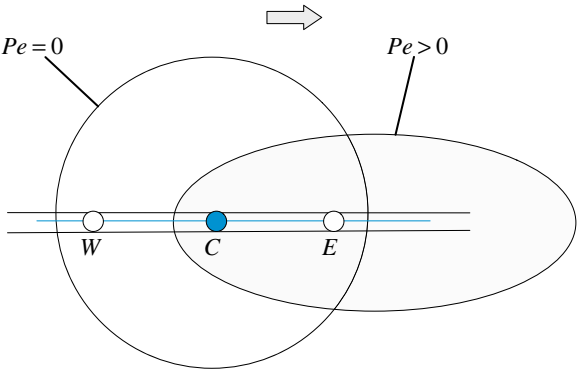


Fig. 5.12 Illustration of the transportive property of fluid flow

may become hyperbolic under certain conditions. The implications on the finite volume equation, visualized in Fig. 5.12, can be explained as follows. If there is a constant source of ϕ within an element C in a flow field with uniform velocity and diffusivity, then the shapes of the contours of constant scalar ϕ will be influenced by the ratio of convection to diffusion strengths, i.e., the Péclet number (Pe) defined as

$$Pe = \frac{Convection\ strength}{Diffusion\ strength} = \frac{\rho u}{\Gamma/\Delta x} \tag{5.39}$$

Based on Eq. (5.39), the case $Pe = 0$ indicates that the transport of ϕ is governed by diffusion, which has an elliptic behavior [8]. In this case, isolines of ϕ are circular and the value of ϕ at C is influenced by the surrounding nodes W and E (Fig. 5.12). Increasing convection effects (i.e., increasing Pe), the circular contours become elliptic in shape and the region influencing the value of ϕ at C shifts in the direction of the flow. Therefore for high Pe flows, events at node C will have a weak or no influence on upstream nodes, while downstream nodes will be strongly affected.

Failure to observe this requirement in the selected discretization schemes can give rise to unstable solutions (i..e., unphysical oscillations).

5.6.8 Boundedness of the Interpolation Profile

Ensuring conservation does not guarantee that other important properties of the original partial differential equation will be maintained by the discretized equation. For example physical considerations lead to the conclusion that in the absence of

sources, the value of the conserved variable ϕ within the domain should be bounded by the values at the domain boundaries [14]. The discretized equation

$$a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C \tag{5.40}$$

would fulfill this requirement when the nodal ϕ_C values satisfy the following constraint:

$$\min_{i=1}^{N} \left(\phi_{F_i}\right) \leq \phi_C \leq \max_{i=1}^{N} \left(\phi_{F_i}\right) \tag{5.41}$$

where F_i represents the ith neighbor of C and N their number. This can be achieved by a judicial control of the discretization schemes and their linearization, as will be detailed in later chapters.

5.7 Variable Arrangement

While the cell-centered variable arrangement is the one selected in OpenFOAM® and generally preferred with the FVM (Fig. 5.13a), vertex-centered [15] (Fig. 5.13b) variable arrangement methods have also been used. Two vertex-centered arrangements have been adopted resulting in either overlapping elements or a dual mesh, respectively. Because the dual mesh method is more popular, it is described next as a representative of vertex-centered methods.

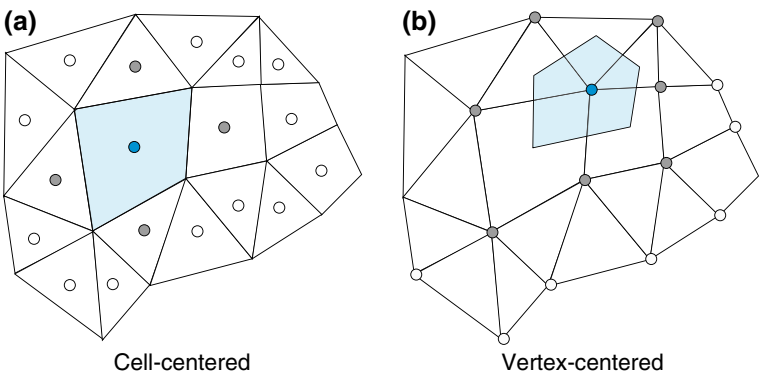


Fig. 5.13 **a** Cell-centered arrangement; **b** vertex-centered arrangement

5.7.1 Vertex-Centered FVM

In a vertex-centered arrangement the flow variables are stored at the vertices (or grid points), with elements constructed around the variable locations by using the concept of a dual mesh and dual cells [16] (Fig. 5.14). Adopting this concept, a cell can be created around a grid point in several ways. In two dimensions, one approach (Fig. 5.14a) is by connecting the centroids of the cells having the grid point in common. As displayed in Fig. 5.14b a second possibility is to join the centroids of the surrounding elements to the centroids of their faces. The same construction procedure can be used in three dimensional situations where an element around a grid point is created by properly connecting surrounding cells' centroids, faces' centroids, and edges' centroids resulting in non-overlapping elements.

The use of a vertex-centered arrangement of variables allows for an explicit profile to be defined over the elements in terms of the vertex variables. In this case the variables represent point values and variation through the element can be computed using shape functions or interpolation profiles. This approach permits an accurate resolution of face fluxes for all mesh topologies, but yields a lower order accuracy of element-based integrations since the vertex is not necessarily at the element centroid. Moreover, it increases the storage requirements due to the creation of larger matrix. Furthermore handling of boundary conditions in cell-vertex schemes, as shown in Fig. 5.14b, requires additional treatment in order to ensure a consistent solution at boundary points shared by multiple grid blocks. Still a major disadvantage is the need to base the mesh on a set of element types for which a shape function can be defined.

Another shortcoming of the vertex-centered scheme with dual elements appears at solid boundaries where only a portion of an element is formed and the node where values are stored is at the wall (Fig. 5.14b). In a regular cell, the integration of face fluxes results in a residual located at the main point inside the element where values are stored. In this case however, the residuals will be stored at the wall

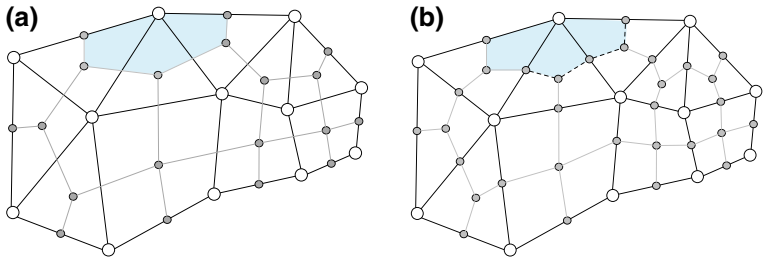


Fig. 5.14 Vertex-centered arrangement: **a** dual cells connecting centroids of cells, and **b** dual cells connecting centroids of cells to centroids of faces

boundary creating a discrepancy and causing an increase in the discretization error there in comparison with cell-centered schemes. Additional complications may also arise at sharp edges and branch cuts.

5.7.2 Cell-Centered FVM

The cell-centered variable arrangement is currently the most popular type of variable arrangement used with the FVM. With this practice, the variables and their related quantities are stored at the centroids of grid cells or elements. Thus, the elements are identical to the discretization elements and, in general, the method is second order accurate since all quantities are computed at element and face centroids, where the difference between the value of the variable and its average is $O(\Delta x^2)$. Variations within the cell can be re-constructed using a Taylor series expansion. Another advantage of the cell-centered formulation is that it allows for the use of general polygonal elements with no need for pre-defined shape functions. This permits a straightforward implementation of a full multigrid strategy.

However two important disadvantages of the method are its treatment of non-conjunctional elements and the manner the diffusion term is discretized on non-orthogonal cells. The first issue influences the accuracy of the method, the second its robustness, while both being affected by the quality of the mesh.

Consider the two-cell arrangement shown in Fig. 5.15. It is clear that any average of a value defined at C and F will be defined at f' rather than f, the centroid of the face. Thus any discretization procedure using this interpolated value will not have an $O(\Delta x^2)$ accuracy.

For a cell-centered scheme the discretization error depends strongly on the smoothness of the grid. Results for such a situation are displayed in Fig. 5.16. The physical situation displayed in Fig. 5.16a represents an annulus between two horizontal cylinders of rhombic cross-sections. The inner cylinder is maintained at the uniform hot temperature T_h while the outer cylinder is kept at the cold temperature T_c. The difference in temperature creates density variation within the enclosure and gives rise to buoyancy forces that establish a flow field. Using the grid system

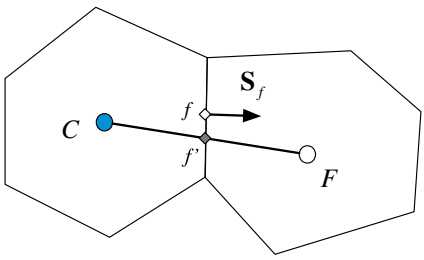


Fig. 5.15 Two non-junctional cell-centered elements

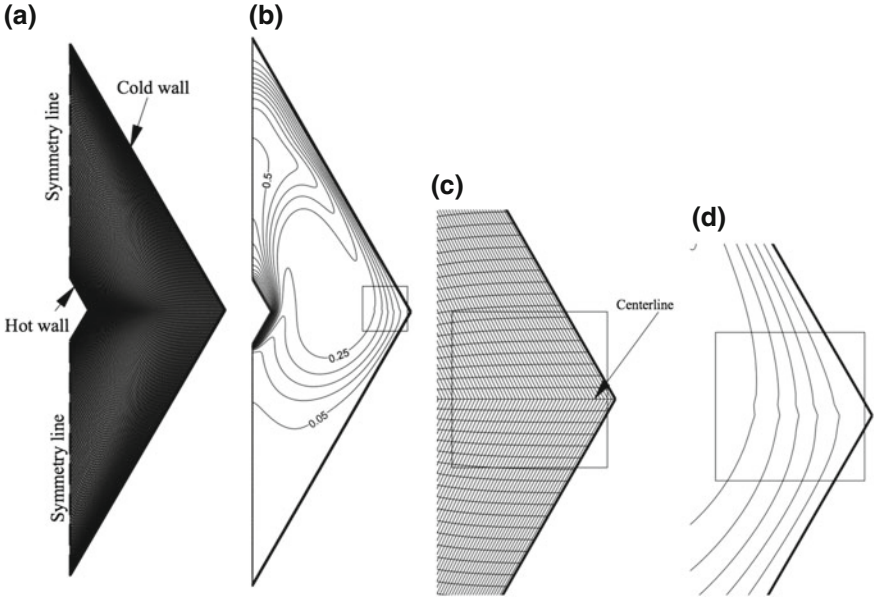


Fig. 5.16 **a** Grid used in solving for natural convection in the annulus between horizontal cylinders of rhombic cross sections; **b** isotherms over the domain; **c** a magnified region showing the grid around the horizontal centerline of the domain; **d** a magnified region around the horizontal centerline showing the kinks in the generated isotherms

displayed in Fig. 5.16a, the velocity and temperature distributions within the enclosure are obtained numerically using a finite volume method. Generated isotherms are displayed in Fig. 5.16b. By carefully inspecting Fig. 5.16b small kinks can be seen in the region around the horizontal centerline of the domain. The region around the kinks is magnified and the grid and isotherms in that region are displayed in Fig. 5.16c, d, respectively. The kinks in the isotherms are clear in the magnified plot and are due to the use of the grid with slope discontinuity causing a discretization error that cannot be reduced independent of how much the grid is refined. This zero order error does not arise with a cell-vertex scheme. Despite this fact, for a sufficiently smooth grid, the cell-centered arrangement can attain accuracy of order two or higher.

The other issue that affects cell centered FVM is the treatment of non-orthogonality in the discretization of the diffusion term. This will be covered in detail in Chap. 8, however a brief explanation is useful at this stage.

In the discretization of the diffusion term (Fig. 15.17) the value of the angle θ extending between the unit vector **e** (which is in the direction of the line joining the centroids of the elements C and F) and the unit vector **n** (which is normal to the face shared by the elements C and F) affects the degree of implicitness and hence the robustness of the method applied in the discretization of the diffusion term.

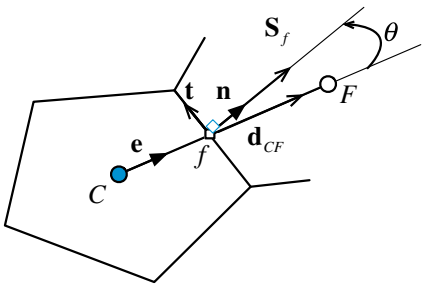


Fig. 15.17 A non-orthogonal element

For discretization, the diffusion term is generally written as

$$\nabla\phi \cdot \mathbf{S} = \underbrace{\nabla\phi \cdot \mathbf{E}}_{\textit{Implicit orthogonal-like contribution}} + \underbrace{\nabla\phi \cdot \mathbf{T}}_{\textit{Explicit non-orthogonal like contribution}} \tag{5.42}$$

The larger the angle θ is, the larger the explicit term will be, and the less robust the discretization method becomes.

To summarize, the vertex-centered scheme with dual elements and the cell-centered scheme are numerically very similar in the interior of a domain for steady state calculations. The only situation where the performance of the vertex-centered scheme is superior to that of the cell-centered scheme is over a distorted grid. In all other situations, it is more advantageous to use the cell-centered arrangement as it leads to a more straightforward implementation in a computer code.

5.8 Implicit Versus Explicit Numerical Methods

As described in Sect. 5.1, once the number of integration points and the type of linearization are defined, the cell-centered finite volume discretization method results in a set of equations with the values of the dependent variables at the cell centers as unknowns. The way these unknowns are organized and solved, classifies the adopted computational approach. Numerical solution schemes are often referred to as being either explicit or implicit.

An explicit numerical method is one in which the dependent variables are computed directly via already known values. In this case any discretization operator can be directly evaluated based on the actual variable values.

On the other hand, a numerical method is said to be implicit when the dependent variables are treated as unknowns and assembled to form a coupled set of equations which are then solved via special numerical tools using either a direct or an iterative solution algorithm.

The conservation equations dealt with in computational fluid dynamics are nonlinear and the implicit approach is most often preferred over the explicit method for solving them. Once the proper discretization and linearization are performed, the

last step is to solve the set or system of algebraic equations to get the solution. Solving the corresponding system with a direct equation solver, as discussed in a previous chapter, is not feasible. The iterative approach being more economical, is the most widely, if not the only one, used. In difference with a direct approach, starting from an initial guess an iterative method progresses toward solution by using results obtained at the end of an iteration as the initial guess for the following iteration. The sequence of events is said to be converging if the intermediate computed solution is approaching the final solution, otherwise it is said to be diverging. The process is general and is used whether solving for one time step in a transient problem or for a final solution in a steady state problem. In fact adopting an iterative approach to obtain a steady state solution is computationally cheaper than marching in time until steady state is reached. Additional details about matrix iterative solvers will be covered in Chap. 10.

5.9 The Mesh Support

Now that the basis of the numerics of the FVM and its properties have been presented, it is worth outlining the needed mesh support. In all preceding derivations it has been implicitly assumed that certain geometric and topological information is readily available. Even though the description of the finite volume mesh will be the subject of the next two chapters, based on the covered material so far, it has become possible to draw a simplified list of its characteristics.

The application of the FVM as a numerical discretization technique necessitates a mesh support that provides a range of information both geometric and topological, as described next.

For an element, the needed information includes: its index, its centroid, a list of bounding faces, and a list of neighboring elements. For a face, information is needed about its index, its centroid, its surface vector, a list of neighboring elements (2 for an interior face and one for a boundary face), in addition to a list of vertices that defines it. Also needed is information about the boundaries of the computational domain, i.e., the boundary faces that define each boundary patch.

Another issue to be resolved is the orientation of the normal vector to an element face. In the above derivations it was assumed that the normal to all faces of the element were pointing outward. The normal vectors to faces in a computational domain cannot be all pointing outward. Generally any element will have some of its faces with their normal vectors pointing outward while for the remaining faces they will be pointing inward. This indicates that a proper account should be made of the face sign.

Why and how the above is defined and used will be the focus of the next chapter.

5.10 Computational Pointers

Throughout this book uFVM, a Matlab®-based finite volume CFD educational code, and openFOAM® [17], an open source finite volume code capable of solving industrial type problems, will be used to illustrate implementation details and concretize various numerical procedures. Generally, comments about implementation techniques, adopted methods, and data structure in these two codes will be added, unless otherwise stated, in the Computational Pointers's section of every chapter to follow. Moreover, a number of uFVM test cases (testDiffusion, testAdvection, testFlow, testSource, etc.) are available for the reader and can be downloaded from the book website at the URL address mentioned earlier.

5.10.1 uFVM

The computer program uFVM is an unstructured three dimensional finite volume code that was developed for academic purposes. It is written in Matlab® and as such lends itself substantially to any type of dissection by the user. Moreover, the Matlab® environment allows users to unroll the various data structures adopted in the code at any time during the running of a case. Furthermore because it is intended for academic and teaching use, clarity of coding was high on the priority list while devising the implementation details of the various algorithms and schemes. Still all of its numerics and algorithms are similar in many ways to those used in industrial oriented CFD codes and thus uFVM is quite useful as a guide for anyone interested in developing a CFD code.

The discretized system of equations in uFVM is stored in a matrix format such that each row represents the discretized equation in one element, with the off-diagonal coefficients representing the mutual influence of neighboring elements. The computational representation of this matrix usually takes into account the fact that each row will contain only a small number of non-zero elements, specifically in each row there will be as many non-zero elements as neighbors to the element associated with the row.

For the example shown in Fig. 5.18, the mesh consists of 7 elements, and element 3 has four neighbors. Thus the discretized equation for element 3 will include 4 coefficients in addition to the diagonal coefficient. It is clear why for large meshes many of the non-diagonal elements are zero.

In both uFVM and as is shown later in OpenFOAM®, and indeed in all CFD codes, a sparse matrix is used to store the various coefficients of the resulting system of equations. In uFVM, the matrix **A** is stored in an array of arrays, where the first element of each row represents the diagonal element, while the remaining elements of the array store the non-diagonal coefficients. The right hand side of the system is stored in a separate array **B**. This is illustrated in Fig. 5.18.

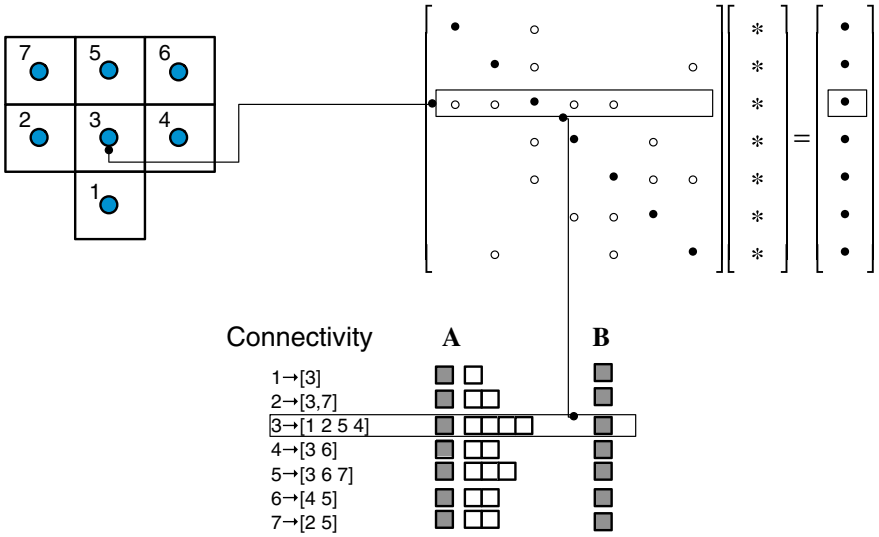


Fig. 5.18 Coefficients for an element equation with element connectivity

5.10.2 OpenFOAM®

OpenFOAM® (Open source Field Operation And Manipulation) is an object-oriented C++ framework that can be used to build a variety of computational solvers for problems in continuum mechanics with a focus on finite volume discretization. OpenFOAM® also includes several ready solvers, utilities, and applications that can be directly used. At the core of these libraries are a set of object classes that allow the programmer to manipulate meshes, geometries, and discretization techniques at a high level of coding. Tables 5.1, 5.2, 5.3 and 5.4 present a list of the main OpenFOAM® classes and their functions. These classes represent the basic bricks for the development of OpenFOAM® based applications and utilities. They enable programmers constructing a variety of algorithms while allowing for extensive code re-use.

Another characteristic of OpenFOAM® is its use of operator overloading that allows algorithms to be expressed in a natural way. For example, the discretization of the transport equation for a generic scalar ϕ given by

Table 5.1 Numerics and discretization

Objects	Type of data	OpenFOAM® Class
Interpolation	Differencing schemes	surfaceInterpolation<template>
Explicit discretization: differential operator	ddt, div, grad, curl	fvc::
Implicit discretization: differential operator	ddt, d2dt2, div, laplacian	fvm::

Table 5.2 Computational domain

Objects	Type of data	OpenFOAM® Class
Variables	Primitive variables	scalar, vector, tensor
Mesh components	Point, face, cell	point, face, cell
Finite volume mesh	Computational mesh	fvMesh, polyMesh
Time	Time database	Time

Table 5.3 Field operation

Objects	Type of data	OpenFOAM® Class
Field	List of values	Field<template>
Dimensions	Dimension set up	dimensionSet
Variable field	Field + mesh + boundaries + dimension	GeometricField<template>
Algebra	+, −, pow, = , sin, cos…	field operators

Table 5.4 Linear equation systems and linear solvers

Objects	Type of data	OpenFOAM® Class
Sparse matrix	Matrix coefficients and manipulation	lduMatrix, fvMatrix
Iterative solver	Iterative matrix solvers	lduMatrix::solver
Preconditioner	Matrix preconditioner	lduMatrix::preconditioner

$$\underbrace{\frac{\partial}{\partial t}(\phi)}_{\text{unsteady term}} + \underbrace{\nabla \cdot (\mathbf{v}\phi)}_{\text{convection term}} = \underbrace{\nabla \cdot \left(D^{\phi} \nabla \phi\right)}_{\text{diffusion term}} + \underbrace{P\phi - C}_{\text{source and sink term}} \tag{5.43}$$

is basically written in OpenFOAM® as (Listing 5.1)

```
(
    fvm::ddt(phi)
  + fvm::div(mDot, phi)
  - fvm::laplacian(Dphi, phi)
 ==
    fvm::Sp(P,phi)
   -fvc::(C)
);
```

Listing 5.1 Script for solving a simple transport equation using OpenFOAM®

The fvm::div operator for example, takes the convective flux as a coefficient field defined over the faces of the control elements and phi as the variable field defined over the cell centroids, and returns a system of equations including a LHS matrix and a RHS source that represent the discretization of the convection operator. These LHS matrices and RHS vectors are generated for each of the operators and then added or subtracted as needed to yield the final system of equations that

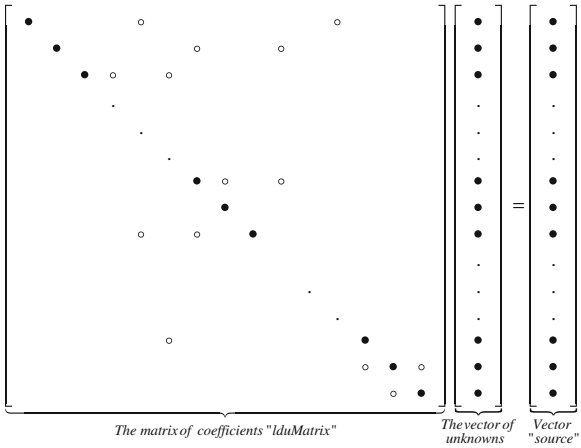


Fig. 5.19 The matrix "lduMatrix" and the vector "source" in which the coefficients and sources are stored

represent the discretized set of algebraic equations defined over each element of the computational domain.

The namespaces fvm:: and fvc:: allow for the evaluation of a variety of operators implicitly and explicitly, respectively.

The explicit operator fvc, named "finite volume calculus", returns an equivalent field based on the actual field values. For example the operator fvc::div(ϕ) returns an equivalent geometricField in which each cell contains the value of the divergence of the variable (ϕ).

The fvm implicit operator, instead, defines the implicit finite volume discretization in terms of matrices of coefficients. For example fvm::laplacian(ϕ) returns an fvMatrix in which all the coefficients are based on the finite volume discretization of the laplacian.

The role of the fvm:: and fvc:: operators is to construct the LHS and RHS of a system of equations representing the discretized form of Eq. (5.43) over each element in the mesh. The discretization process yields a system of equations that can be represented in the matrix form shown in Fig. 5.19.

Each row of the system represents the discretized equation in one element, with the off-diagonal coefficients representing the mutual influence of the neighboring elements.

In OpenFOAM® the matrix of coefficients is stored in a class named lduMatrix and the specialized fvMatrix. The chosen storage format of the coefficients is based on the ldu sparse format, in which all coefficients are stored in three main arrays: diagonal, upper, and lower. The diagonal coefficients are thus stored in one array accessed by diag() function with its dimension being equal to the number of elements in the computational domain. The upper and lower coefficients are stored each in an array as shown in Fig. 5.20. The source or the right hand side is stored under a specific vector called source with its dimension being also equal to the number of cells over the computational domain.

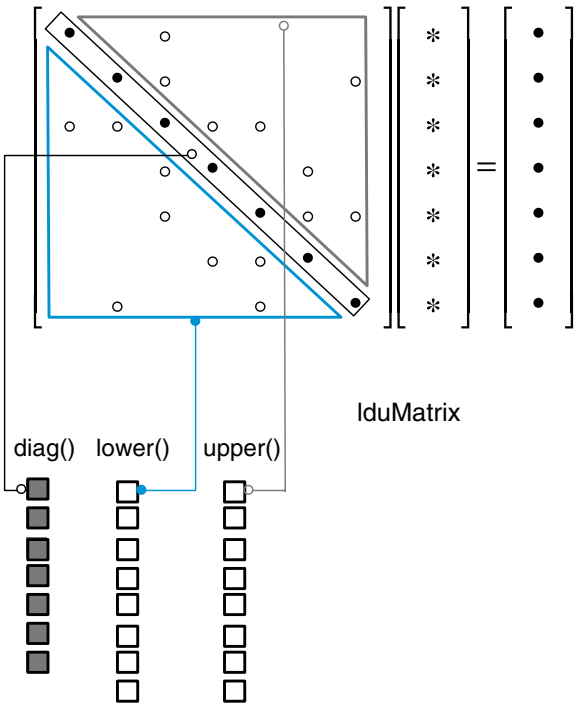


Fig. 5.20 The ldu sparse matrix storage (lduMatrix)

It is worth noting that despite defining the fvMatrix class as a "template" the vector specialization of the class does not imply a block coupled form of the matrix itself but just a vector of scalar equations that are solved in a standard segregated approach. This is confirmed by the code defined inside the fvMatrix class shown in Listing 5.2 with its member function, which returns the diagonal vector coefficients of the matrix, only returning as object a scalar field independently of the template type.

```
template<class Type>
class fvMatrix
:
    public refCount,
    public lduMatrix
{
…

//- Return the matrix scalar diagonal
  tmp<scalarField> D() const;
…

//- Return the central coefficient
tmp<volScalarField> A() const;
```

Listing 5.2 Excerpt of the code defined inside the fvMatrix class returning as object a scalar field

More details on the coefficients' storage techniques used in uFVM and OpenFOAM® will be presented in Chap. 7.

5.11 Closure

In this chapter a general overview of the FVM was presented. The discretization process was shown to involve two distinct steps, with step 1 resulting in a set of semi-discretized equations. The chapter also discussed some guiding principles for the discretization process, which guarantee that the resulting discretized equation will possess some desirable attributes. Before embarking onto a detailed description of the numerical techniques that will be used in step 2 of the discretization process, the focus of the next chapter will be on the discretization of the computational domain and some related issues.

5.12 Exercises

Exercise 1
A critical tool used with OpenFOAM® is Doxygen [18]. It is an open source software that generates automatic documentation from annotated C++ sources. Doxygen uses the UML (http://www.uml-diagrams.org/class-reference.html) graph formalism and is capable of visually displaying the relations between the various classes, inheritance diagrams, and collaboration diagrams, which are all generated automatically as easily browsed HTML documents. Two options are available to access Doxygen. The first is to generate the documents by direct compilation on the local machine (the main files are placed in $WM_PROJECT_DIR/doc/Doxygen). The second option is to access Doxygen on the web at the address http://www.openfoam.org/docs/cpp/. This latest option yields always up to date documentation. To start, either open the browser at the address http://www.openfoam.org/docs/cpp/ or open the file $WM_PROJECT_DIR/doc/Doxygen/html/index.html. Once opened, click on the *Classes* button that appears in the menu in the upper part of the screen. Then click on the *Class Index* button. Once clicked, a list ordered alphabetically of all classes defined in OpenFOAM® is displayed. The inheritance diagram of any class along its public member functions and derived classes are obtained by simply clicking on the class name appearing in the list. Doxygen is used to quickly browse all source files as well as checking all the properties of the classes and namespaces. Basically the most common information to look for are class inheritances and public member functions as well as public class constructors.

(a) Using the Doxygen documentation find the class definition of the following type of data:

Foam::face
Foam::cell
Foam::fvMesh
Foam::fvMatrix

(b) Find the parent classes for each of the classes of (a).
(c) Find the public constructors for each of the classes defined in (a).
(d) Find all the public member functions defined in each of the classes of (a).
(e) Find all the public member functions defined in the parent class of each of the classes of (a).

Exercise 2
Run testDiffusion (which is available as a test case with uFVM) for one iteration, then

(a) Use 'cfdGetCoefficients' to get the global matrix data structure and compare it to that shown in Fig. 5.18.
(b) Use cfdGetMesh to get a reference to the Mesh object and find the definition of a face, a cell and a vertex.

References

1. Blazek J (2005) Computational fluid dynamics: principles and applications. Elsevier, Amsterdam
2. Ferziger JH, Peric M (2002) Computational methods for fluid dynamics, 3rd edn. Springer, Berlin
3. Versteeg HK, Malalasekera W (2007) An introduction to computational fluid dynamics: the finite volume method. Prentice Hall, New Jersey
4. Courant R, Friedrichs KO, Lewy H (1928) Über die partiellen Differenzengleichungen der Mathematischen Physik. Math Ann 100:32–74 (English translation, with commentaries by Lax, P.B., Widlund, O.B., Parter, S.V., in IBM J. Res. Develop. 11 (1967))
5. Crank J, Nicolson P (1947) A practical method for numerical integration of solution of partial differential equations of heat-conduction type. Proc Cambridge Philos Soc 43:50–67
6. Clough RW (1960) The finite element method in plane stress analysis. Proceedings of second ASCE conference on electronic computation, Pittsburg, Pennsylvania, 8:345–378
7. Launder BE, Spalding DB (1972) Mathematical models of turbulence. Academic Press, Massachusetts
8. Patankar SV (1967) Heat and mass transfer in turbulent boundary layers, Ph.D. Thesis, Imperial College, London University, UK
9. Gosman AD, Pun WM, Runchal AK, Spalding DB, Wolfshtein M (1969) Heat and mass transfer in recirculating flows. Academic Press, London
10. Runchal AK (1969) Transport processes in steady two-dimensional separated flows. Ph.D. Thesis, Imperial College of Science and Technology, London, UK

11. Runchal AK, Wolfshtein M (1969) Numerical integration procedure for the steady-state Navier-Stokes equations. Mech Eng Sci II 5:445–453
12. Rhie CM, Chow WL (1983) A numerical study of the turbulent flow past an isolated airfoil with trailing edge separation. AIAA J 21:1525–1532
13. Lilek Z, Peric M (1995) A fourth-order finite volume method with collocated variable arrangement. Comput Fluids 24(3):239–252
14. Leonard BP (1988) Universal limiter for transient interpolation modeling of the advective transport equations: the ultimate conservative difference scheme. NASA Technical Memorandum 100916, ICOMP-88-11
15. Baliga BR, Patankar SVA (1983) Control volume finite-element method for two-dimensional fluid flow and heat transfer. Numer Heat Transf 6:245–261
16. Vaughan RV (2009) Basic control volume finite element methods for fluids and solids. IISc research monographs series, IISc Press
17. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
18. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 6
The Finite Volume Mesh

Abstract A key ingredient in the implementation of the finite volume method is setting up the geometrical support framework for the problem at hand. This process starts with mesh generation, which replaces the continuous domain by a discrete one formed of a contiguous set of non overlapping elements or cells delimited by a set of faces, and the defining of the physical boundaries through the marking of the boundary faces. It continues with the computation of relevant geometric information for the various components of the computational mesh, and is completed by capturing the topology of these components, i.e., how they are related and located one with respect to the other. Thus the result of the domain discretization step is not only the set of non-overlapping elements and other related geometric entities and the generated information about their geometric properties, but also the topological information about their arrangement and relations. It is this combined information that defines the finite volume mesh. The objective of this chapter is to clarify the topological and geometric requirements of the finite volume mesh.

6.1 Domain Discretization

The discretization of the physical domain, or mesh generation, produces a computational mesh (Fig. 6.1) on which the governing equations are subsequently solved. The methods and techniques used for domain discretization have changed drastically over the last few decades [1, 2], and, nowadays, have become mostly automated [3–6]. Before reviewing the types of elements commonly used in a computational mesh, the characteristics and attributes that the mesh system should possess in order to be employed in the context of the finite volume method are first described. These attributes will be presented in the context of computing the gradient of a variable ϕ on both a structured and an unstructured triangular mesh.

In general a geometric domain may be discretized using either a structured or an unstructured grid system. In a structured mesh, three dimensional elements are defined by their local indices (i, j, k). A structured grid system has many coding and

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_6

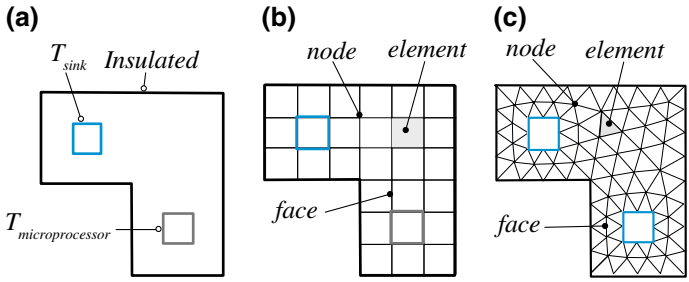


Fig. 6.1 **a** Domain of interest, **b** domain discretized using a uniform grid system, and **c** domain discretized using an unstructured grid system with triangular elements

performance advantages but suffers from a limited geometric flexibility. Additional flexibility in the generation of structured meshes can be achieved by employing multiple blocks to define the geometry, with a structured mesh generated for each block either independently from other blocks or jointly.

Another way to make the mesh generation more flexible is to avoid the use of structured grids with their implicit topological information, and to adopt an unstructured mesh with explicit topological information based on connectivity tables and geometric entity numbering.

Structured grids remained the staple of numerical simulation for a long time and it is only in the past two decades that unstructured grids became more popular [7]. Starting in early 1970s, interest in automatic mesh generation escalated as problem size increased and manual mesh generation became too time consuming [8]. The first methods were semi-automatic with an operator manually placing points in the computational domain and then, in a second step, using a computer to generate the mesh. Nowadays the whole process is fully automated with both points and elements generated automatically.

Most modern CFD codes have the ability to use unstructured grids in addition to a variety of hybrid multiblock grids [9]. OpenFOAM® [10] uses unstructured grids but can also use conforming and non-conforming multiblock grids [11]. The finite volume method will be presented here in the context of an unstructured grid system. However as the unstructured finite volume mesh requirements are defined, its characteristics will be compared to those of a structured grid system.

6.2 The Finite Volume Mesh

The discussion for the requirements of the finite volume mesh will be contextualized in terms of a simple problem, namely the computation of the gradient of an element field. The gradient will first be computed on a structured grid and then over an unstructured grid; the differences will help clarifying a number of issues.

6.2.1 Mesh Support for Gradient Computation

Many techniques can be used to compute the gradient of an element field, and these will form the subject of Chap. 9. The method adopted in this chapter is based on the Green-Gauss theorem [12, 13]. It is relatively straightforward and can be used for a variety of topologies and grids (structured/unstructured, orthogonal/nonorthogonal, etc.). The starting point is defining the average gradient over a finite volume element of centroid C and volume V_C as

$$\overline{\nabla\phi}_C = \frac{1}{V_C}\int_{V_C} \nabla\phi \, dV \tag{6.1}$$

Then, using the divergence theorem, the volume integral is transformed into a surface integral yielding

$$\overline{\nabla\phi}_C = \frac{1}{V_C}\int_{\partial V_C} \phi \, d\mathbf{S} \tag{6.2}$$

where $d\mathbf{S}$ is the **outward** pointing surface vector. In the presence of discrete faces, Eq. (6.2) can be written as

$$\overline{\nabla\phi}_C V_C = \sum_{\partial V_C}\int_{face} \phi \, d\mathbf{S} \tag{6.3}$$

Next the integral over a cell face is approximated using the mid-point integration rule to become equal to the interpolated value of the field at the face centroid multiplied by the face area, resulting in

$$\overline{\nabla\phi}_C = \frac{1}{V_C}\sum_{f=nb(C)} \overline{\phi_f}\mathbf{S}_f \tag{6.4}$$

By reviewing Eq. 6.4 and Fig. 6.2, it is clear that to compute the average of the gradient over the control element C, information about the face area and direction ($\mathbf{S}_f$) is required, as well as information about the neighboring elements and the ϕ values at the element centroids $\left(\phi_C, \phi_{F_k}\right)$. This information is needed to compute the value of ϕ at the interface (ϕ_f), which will have to be interpolated in some fashion.

A profile for the variation of the dependent variable ϕ between nodal values is assumed, which basically introduces an approximation in the evaluation of the gradient. In all cases the value of ϕ_f will have to be computed at each face centroid. Assuming a linear profile for the variation of ϕ between the elements C and F straddling the interface f, an approximate value for ϕ_f, denoted by $\overline{\phi}_f$, can be computed as

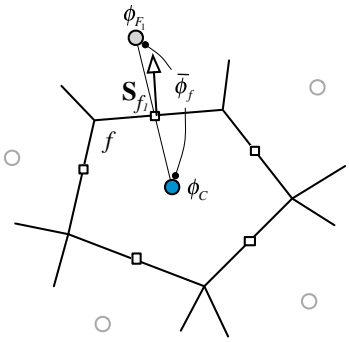


Fig. 6.2 Gradient computation

$$\overline{\phi}_f = g_F \phi_F + g_C \phi_C \tag{6.5}$$

One way to calculate the weight factors g_F and g_C is given by

$$g_F = \frac{V_C}{V_C + V_F} \qquad g_C = \frac{V_F}{V_C + V_F} = 1 - g_F \tag{6.6}$$

Other interpolation practices may be used some of which will be explained later in the chapter.

Example 1

Compute the gradient for the two fields given in Table 6.1 *over the two-dimensional cell shown in* Fig. 6.3 *using the surface vector values given in the table.*

Table 6.1 Geometric data for Example 1

	Sf	V	Field (1)	Field (2)
C		37.8	1	6
1	(−2.4, −3.24)		1	10
2	(2.4, −3.48)		1	9
3	(4.1, −6.7)		1	5
4	(2.2, 3.7)		1	3
5	(−2.64, 2.9)		1	4
6	(−3.66, 6.82)		1	8

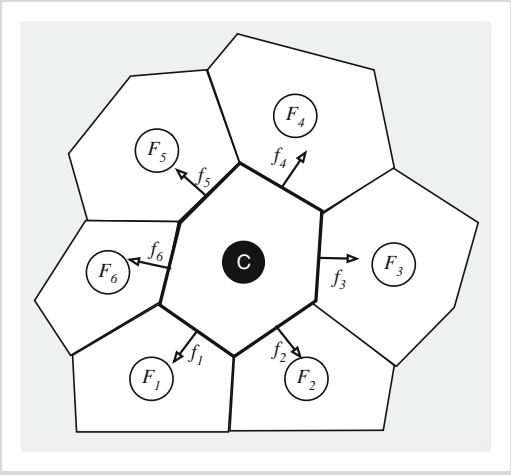


Fig. 6.3 A two dimensional cell for Example 1

Solution

For case 1, the field is constant with a value of 1, so the value of the gradient is expected to be 0.

$$\begin{aligned}
\overline{\nabla\phi}_C &= \frac{1}{V_C}\sum_{f=nb(C)} \phi_f \mathbf{S}_f \\
&= \frac{1}{V_C}\left(\phi_{f_1}\mathbf{S}_{f_1} + \phi_{f_2}\mathbf{S}_{f_2} + \phi_{f_3}\mathbf{S}_{f_3} + \phi_{f_4}\mathbf{S}_{f_4} + \phi_{f_5}\mathbf{S}_{f_5} + \phi_{f_6}\mathbf{S}_{f_6}\right) \\
\overline{\nabla\phi}_C &= \frac{1}{37.8}\begin{pmatrix} 1(-2.4\mathbf{i} - 3.24\mathbf{j}) + 1(2.4\mathbf{i} - 3.48\mathbf{j}) + 1(4.1\mathbf{i} - 6.7\mathbf{j}) \\ + 1(2.2\mathbf{i} + 3.7\mathbf{j}) + 1(-2.64\mathbf{i} + 2.9\mathbf{j}) + 1(-3.66\mathbf{i} + 6.82\mathbf{j}) \end{pmatrix} \\
&= (0\mathbf{i} + 0\mathbf{j})
\end{aligned}$$

This is actually a property of the surfaces of closed elements, provided they all are pointing outward (or inward) they always sum to zero.

For case 2 the gradient is computed as

$$\begin{aligned}
\overline{\nabla\phi}_C &= \frac{1}{V_C}\sum_{f=nb(C)} \phi_f \mathbf{S}_f \\
&= \frac{1}{V_C}\left(\phi_{f_1}\mathbf{S}_{f_1} + \phi_{f_2}\mathbf{S}_{f_2} + \phi_{f_3}\mathbf{S}_{f_3} + \phi_{f_4}\mathbf{S}_{f_4} + \phi_{f_5}\mathbf{S}_{f_5} + \phi_{f_6}\mathbf{S}_{f_6}\right) \\
&= \frac{1}{37.8}\begin{pmatrix} 10(-2.4\mathbf{i} - 3.24\mathbf{j}) + 9(2.4\mathbf{i} - 3.48\mathbf{j}) + 5(4.1\mathbf{i} - 6.7\mathbf{j}) \\ + 3(2.2\mathbf{i} + 3.7\mathbf{j}) + 4(-2.64\mathbf{i} + 2.9\mathbf{j}) + 8(-3.66\mathbf{i} + 6.82\mathbf{j}) \end{pmatrix} \\
&= -15.14\mathbf{i} - 19.96\mathbf{j}
\end{aligned}$$

6.3 Structured Grids

For a regular structured grid, every interior cell in the domain is connected to the same number of neighboring cells. These neighboring cells (Fig. 6.4) can be identified using the indices i, j, and (k) in the x, y, and (z) coordinate direction, respectively, and can be directly accessed by incrementing or decrementing the respective indices. This allows for lower memory usage since topological information is embedded in the mesh structure through the indexing system. This also leads to greater efficiency in coding, cache utilization, and vectorization. Structured grids were widely used in the development of the Finite Volume and Finite Difference methods.

In a structured grid, one can associate with each computational cell an ordered set of indices (i, j, k), where each index varies over a fixed range, independently of the values of the other indices, and where neighboring cells have associated indices that differ by plus or minus one. Thus, if there are Ni, Nj, and Nk elements in the i, j, and k index direction, respectively, then the total number of elements in the domain is $Ni * Nj * Nk$. In three-dimensional spaces, elements are hexagons with 6 faces and 8 vertices, with each interior element having 6 neighbors. In two-dimensions, elements are quadrilaterals with 4 faces and 4 vertices, with each interior element having 4 neighbors.

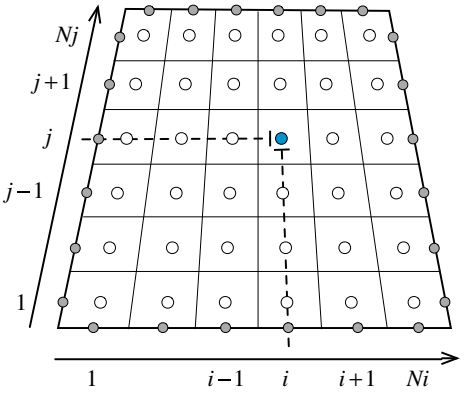


Fig. 6.4 Local indices and topology

6.3.1 Topological Information

Global indices are generally used for building the full system of equations over the computational domain, while local indices are employed to define the local stencil for an element, information that is useful during the discretization process. In structured grid systems local indices are used interchangeably as global indices

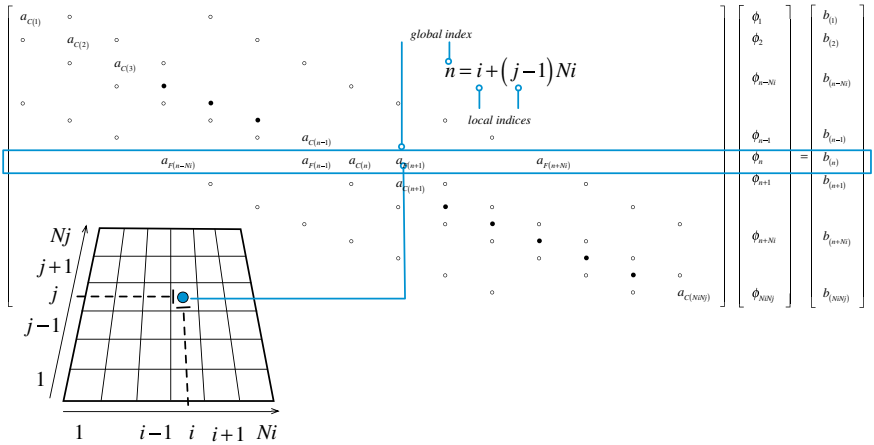


Fig. 6.5 Local versus global indices

because they can be readily translated to global indices and vice versa, as illustrated in Fig. 6.5. In two dimensional spaces the relation between local indices (i, j) and the global index (n) is given by

$$n = i + (j-1)Ni \qquad 1 \le i \le Ni \qquad 1 \le j \le Nj \tag{6.7}$$

and the corresponding global indices for the neighbors of cell (i, j) are obtained as

$$\begin{array}{lcllcl} (i,j) & \rightarrow & n & & & \\ (i+1,j) & \rightarrow & n+1 & (i-1,j) & \rightarrow & n-1 \\ (i,j+1) & \rightarrow & n+Ni & (i,j-1) & \rightarrow & n-Ni \end{array} \tag{6.8}$$

On the other hand, in three-dimensional spaces the relation is

$$n = i + (j-1)Ni + (k-1)Ni * Nj \qquad 1 \le i \le Ni \qquad 1 \le j \le Nj \qquad 1 \le k \le Nk \tag{6.9}$$

and the corresponding global indices for the neighbors of cell (i, j, k) are given by

$$\begin{array}{lcllcl} (i,j,k) & \rightarrow & n & & & \\ (i+1,j,k) & \rightarrow & n+1 & (i-1,j,k) & \rightarrow & n-1 \\ (i,j+1,k) & \rightarrow & n+Ni & (i,j-1,k) & \rightarrow & n-Ni \\ (i,j,k+1) & \rightarrow & n+Ni*Nj & (i,j,k-1) & \rightarrow & n-Ni*Nj \end{array} \tag{6.10}$$

This greatly simplifies the access to the coefficients and the solution of the system of equations, since the coefficients constructed over the local stencil of a cell are basically used in the general system of equations with no need for a translational

step between local and global indices as the mapping between them is readily available. This also applies to the geometric fields and the various conservation fields that are being resolved.

Example 2
In a 5 × 7 structured grid find the global indices of the neighbors of the element C defined by the local index (3, 4).

Solution
For the defined mesh $Ni = 5$ and $Nj = 7$, the global index of element $C(3, 4)$ is thus $3 + (4 - 1) * 5 = 18$, the neighbors of elements C in terms of their local and global indices are (2, 4) → 17, (4, 4) → 19, (3, 3) → 13, and (3, 5) → 23.

6.3.2 Geometric Information

As shown in Fig. 6.6, accessing the local geometric information around an element is quite simple. For element (i, j) the surrounding stored faces are $\mathbf{S}1(i, j)$, $\mathbf{S}2(i, j)$, $\mathbf{S}1(i + 1, j)$, and $\mathbf{S}2(i, j + 1)$. Since for any element the faces have to be pointing outward (see Fig. 6.6), then

$$\begin{aligned} \mathbf{S}_{i-1/2,j} &= -\mathbf{S}1(i,j) \\ \mathbf{S}_{i,j-1/2} &= -\mathbf{S}2(i,j) \end{aligned} \tag{6.11}$$

while for other faces, the following applies:

$$\begin{aligned} \mathbf{S}_{i+1/2,j} &= \mathbf{S}1(i+1,j) \\ \mathbf{S}_{i,j+1/2} &= \mathbf{S}2(i,j+1) \end{aligned} \tag{6.12}$$

The element field in a structured two or three dimensional mesh is also defined as an array of size $[Nx][Ny]$ or $[Nx][Ny][Nz]$, respectively. Thus accessing the element value and its neighbors is equally simple. The element field of a multi-dimensional system may also be defined using a one dimensional array, which in two

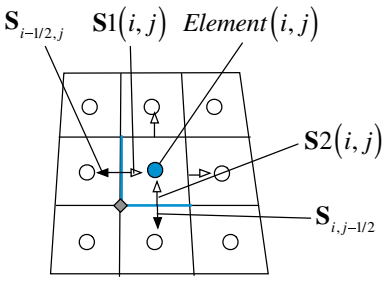


Fig. 6.6 Geometric information

dimensions will be of size $[Ni * Nj]$ and in three dimensions of size $[Ni * Nj * Nk]$. Representing a multi-dimensional field by a one-dimensional array, with values stored using the global indexing system, saves a lot of computer memory when a multi grid system is adopted.

6.3.3 Accessing the Element Field

Accessing the field in a structured grid is as simple as using the indices of the element. Therefore, in a two-dimensional space, $\phi(i,j)$ or $\phi_{i,j}$ is the value of field ϕ at element (i, j). As shown in Fig. 6.7a the values of ϕ at the neighboring cells to (i, j) are, respectively, $\phi_{i+1,j}$, $\phi_{i-1,j}$, $\phi_{i,j+1}$, and $\phi_{i,j-1}$.

As mentioned above computing the gradient using Eq. (6.4) requires calculating the value of ϕ at each face of the finite volume (for our pseudo elements the front and back faces have the same value and thus will not be included in the computation). Thus in addition to the value of ϕ at point (i, j), the values of ϕ at the neighboring points $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$, and $(i, j - 1)$ are also needed. In a structured grid this information is readily available and the value at the face is computed by simple interpolation between the values of ϕ at the centroids of the two volumes sharing the face. Using Eq. (6.5), the interpolated value at face $(i+1/2, j)$ in terms of local indices can be written as

$$\overline{\phi}_{i+1/2,j} = g_{i+1/2,j}\phi_{i+1,j} + \left(1 - g_{i+1/2,j}\right)\phi_{i,j} \tag{6.13}$$

Details of the linear interpolation will be presented later in the chapter. Referring to Fig. 6.7a, the gradient at element (i, j) can be computed using local indexing as

$$\begin{aligned}\overline{\nabla\phi_{i,j}} &= \frac{1}{V_{ij}}\left(\overline{\phi}_{i+1/2,j}\mathbf{S}_{i+1/2,j} + \overline{\phi}_{i-1/2,j}\mathbf{S}_{i-1/2,j} + \overline{\phi}_{i,j+1/2}\mathbf{S}_{i,j+1/2} + \overline{\phi}_{i,j-1/2}\mathbf{S}_{i,j-1/2}\right)\\ &= \frac{1}{V_{ij}}\left(\overline{\phi}_{i+1/2,j}\mathbf{S}1_{i+1,j} - \overline{\phi}_{i-1/2,j}\mathbf{S}1_{i,j} + \overline{\phi}_{i,j+1/2}\mathbf{S}2_{i,j+1} - \overline{\phi}_{i,j-1/2}\mathbf{S}2_{i,j}\right)\end{aligned} \tag{6.14}$$

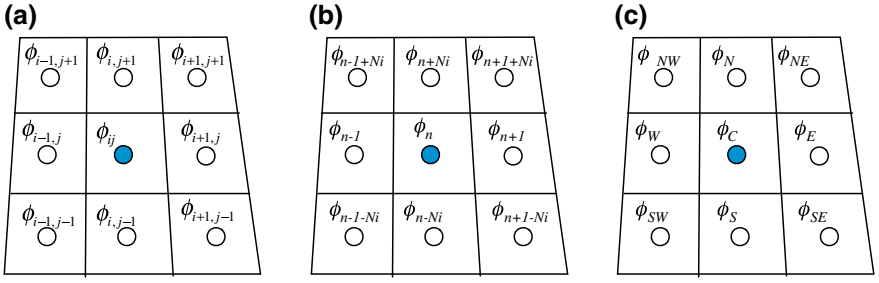


Fig. 6.7 Local versus discretization versus global indices. **a** Local indexing, **b** global indexing, and **c** discretization indexing

while using a global indexing system, Fig. 6.7b, it becomes

$$\overline{\nabla \phi_n} = \frac{1}{V_n}\left(\overline{\phi}_{n+1/2}\mathbf{S}_{n+1/2} + \overline{\phi}_{n-1/2}\mathbf{S}_{n-1/2} + \overline{\phi}_{n+Ni/2}\mathbf{S}_{n+Ni/2} + \overline{\phi}_{n-Ni/2}\mathbf{S}_{n-Ni/2}\right) \tag{6.15}$$

It should be mentioned that **S** is the outward vector normal to the surface at the control volume face. Except at the domain boundaries, control volume faces are shared by two elements. Therefore the outward direction for one element will represent the inward direction for the other element. Thus to avoid duplicating surface vectors at interfaces, only one vector is computed and stored at an interface; the one in the direction of increasing i or j. The embedded features in a structured grid system allows for the correct direction to be chosen with no need to store any additional information. For any element (i, j), the surface with an index greater than i or j is positive while the surface with an index lower than i or j is multiplied by a negative sign. This explain the negative signs in Eqs. (6.11) and (6.14).

6.3.3.1 Discretization Indexing

In addition to local and global indexing, another type known as discretization indexing is sometimes used, where the fields and geometric quantities are defined in terms of their position or neighboring values. Referring to Fig. 6.7c, the gradient at element (i, j) can be computed using discretization indexing as

$$\overline{\nabla \phi_C} = \frac{1}{V_C}\left(\overline{\phi}_e\mathbf{S}_e + \overline{\phi}_w\mathbf{S}_w + \overline{\phi}_n\mathbf{S}_n + \overline{\phi}_s\mathbf{S}_s\right) \tag{6.16}$$

Thus the algorithm for computing the gradient could be written as

```
>Loop over elements (i,j)
     > initialize element gradient to zero grad(i,j) = 0
     > loop over the element faces
          > compute the flux_f =  phi_f*S_f
          > add/subtract flux_f to the element gradient
          depending on the orientation of S_f (pointing
          out of/into element)
     >  divide the sum of the fluxes stored in the
     gradient by the volume of the element to yield the
     element gradient
```

6.4 Unstructured Grids

Unstructured grids offer more flexibility in meshing a domain both in terms of the element types that can be used and in terms of where the elements can be concentrated. This flexibility, however, comes at the cost of additional complexity.

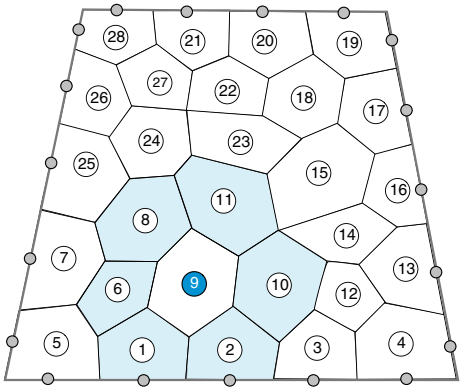


Fig. 6.8 Unstructured mesh global indexing

In an unstructured mesh system, elements are numbered sequentially, as are faces, nodes, and other geometric quantities. This means that there is no direct way to link various entities together based on their indices alone. Thus local connectivity has to be defined explicitly starting with determining the geometric quantities for a particular element. In Fig. 6.8, for example, the neighbors of element 9 cannot be directly derived from its index. Similarly the bounding faces of element 9, or for that matter their nodes, cannot be guessed or derived from its index in the same way this can be done in a structured grid.

Therefore detailed topological information about neighboring elements, faces, and nodes is needed to complement the global indexing of the grid.

6.4.1 *Topological Information (Connectivities)*

As shown in Fig. 6.9, topological information is developed by explicitly constructing the local, Fig. 6.9a, and global, Fig. 6.9b, indices that define the geometric component connectivities (element to element, element to faces, faces to elements, element to nodes, etc.). To this end, the data structure of elements, faces, and nodes now include information about neighboring connections in terms of local and global indices.

The gradient computation algorithm can now be expressed in terms of the discretization indices shown in Fig. 6.9a as

$$\overline{\nabla\phi_C} = \frac{1}{V_C}\left(\overline{\phi}_{f_1}\mathbf{S}_{f_1} + \overline{\phi}_{f_2}\mathbf{S}_{f_2} + \overline{\phi}_{f_3}\mathbf{S}_{f_3} + \overline{\phi}_{f_4}\mathbf{S}_{f_4} + \overline{\phi}_{f_5}\mathbf{S}_{f_5} + \overline{\phi}_{f_6}\mathbf{S}_{f_6}\right) \tag{6.17}$$

or in terms of the local numbers of the element faces shown in Fig. 6.9a as

$$\overline{\nabla\phi_{(0)}} = \frac{1}{V_{(0)}}\left(\overline{\phi}_{(1)}\mathbf{S}_{(1)} + \overline{\phi}_{(2)}\mathbf{S}_{(2)} + \overline{\phi}_{(3)}\mathbf{S}_{(3)} + \overline{\phi}_{(4)}\mathbf{S}_{(4)} + \overline{\phi}_{(5)}\mathbf{S}_{(5)} + \overline{\phi}_{(6)}\mathbf{S}_{(6)}\right) \tag{6.18}$$

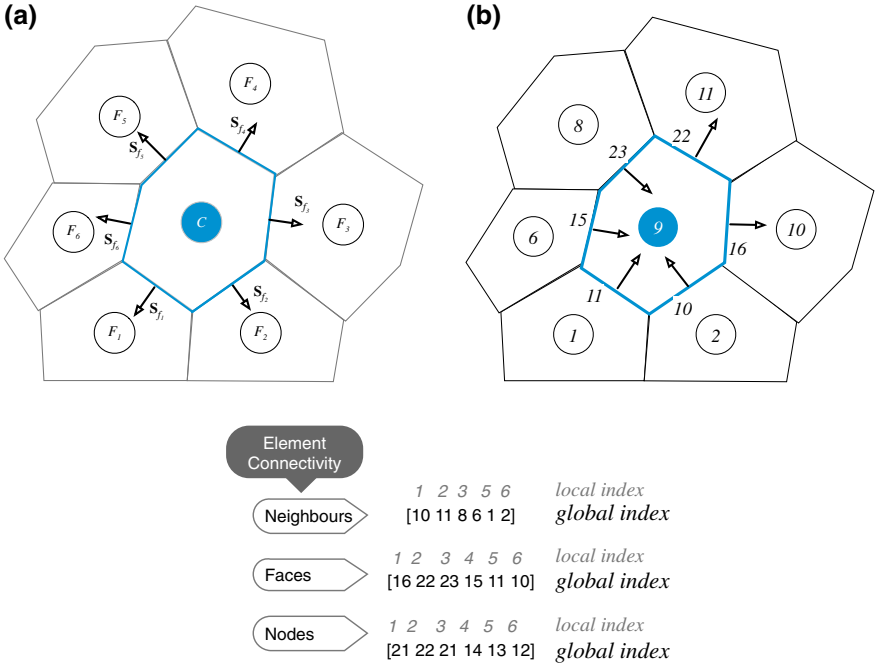


Fig. 6.9 Element connectivity and face orientation using **a** local indices and **b** global indices

where the "()" indicates the local index of the face. The gradient relation can be also written in terms of global indices representing the stored values at the element faces shown in Fig. 6.9b as

$$\overline{\nabla\phi_9} = \frac{1}{V_9}\left(\overline{\phi}_{16}\mathbf{S}_{16} + \overline{\phi}_{22}\mathbf{S}_{22} - \overline{\phi}_{23}\mathbf{S}_{23} - \overline{\phi}_{15}\mathbf{S}_{15} - \overline{\phi}_{11}\mathbf{S}_{11} - \overline{\phi}_{10}\mathbf{S}_{10}\right) \tag{6.19}$$

where the negative signs for terms of faces 23, 15, 11 and 10 are to be noted. While the local surface vector **S** is always assumed to be in the outward direction, this is not necessarily true, as only one normal vector is stored at any one face. A look at Fig. 6.9 shows that these specific stored surface vectors are actually pointing inward of element 9, thus the negative sign. Unlike structured grid systems where the correct direction can easily be obtained, in an unstructured grid the direction of the normal to the surface should somehow be stored. This will be detailed in the following section. In order to account for the vector direction, a sign function is used and the equation for the gradient is modified as

$$\overline{\nabla\phi_k} = \frac{1}{V_k}\left(-\sum_{n\leftarrow\langle f\sim nb(k)\rangle<k}\overline{\phi}_n\mathbf{S}_n + \sum_{n\leftarrow\langle f\sim nb(k)\rangle>k}\overline{\phi}_n\mathbf{S}_n\right) \tag{6.20}$$

For faces, information about the straddling elements is what defines the topology of the face. Furthermore the orientation of the face can be defined in a standard fashion by indexing the elements in a specified order. In this case the normal vector at the interface between two elements is oriented from element 1 to element 2, which are also denoted in OpenFOAM® by the *owner* and *neighbor* elements, respectively, as shown in Fig. 6.10.

Therefore if the interface is considered with element 2, then it should be multiplied by a negative sign. Thus considering the faces that bound element 9, the connectivity information is defined as shown in Fig. 6.11.

The computation of the gradient can be done for every element. However as the flux $\overline{\phi}_f \mathbf{S}_f$ at every interface is the same for the straddling elements with the

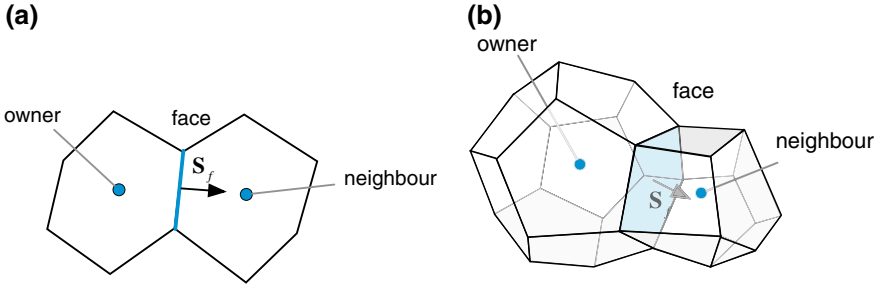


Fig. 6.10 Owners, neighbors, and faces for **a** 2D and **b** 3D elements

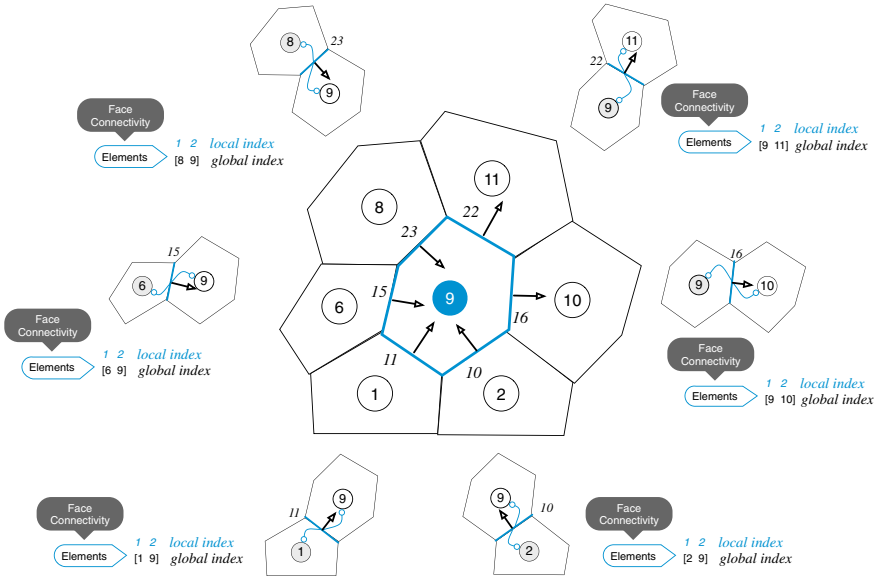


Fig. 6.11 An example of face, element, and node connectivities for unstructured grids

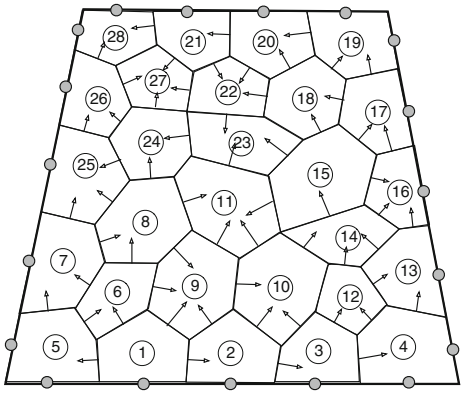


Fig. 6.12 An unstructured mesh system

difference being its sign, the computation of the gradient could proceed in a more efficient manner by computing the gradient over the entire domain (e.g., the entire domain shown in Fig. 6.12) rather than element by element. This is done by looping over all the faces in the computational domain and directly updating the value of the gradient for the elements straddling the interface by incrementing and decrementing the calculated flux from the gradient of element 1 and element 2, respectively.

Therefore the algorithm on an unstructured grid for calculating a gradient field becomes

Algorithm for computing the gradient on an unstructured grid system

```
1. Declare gradient array and initialize it to zero
2. Loop over interior faces
     > compute flux_f = phi_f*Sf
     > add flux_f to gradient of owner element and -
     flux_f to gradient of neighbor element
3.       Loop over boundary faces
     > compute flux_f = phi_f Sf
     > add flux_f to gradient of owner element
4.       Loop over elements
     > divide gradient by volume of element
```

This basically yields the gradient at each element in the computational domain as per Eq. (6.20). The same algorithm could be used with a structured grid to reduce the computational cost.

Example 3
Write the connectivity arrays for elements 1 *and* 5, *and faces* 1, 7, 11 *and* 23 *for the mesh shown in* Figs. 6.13 *and* 6.14.

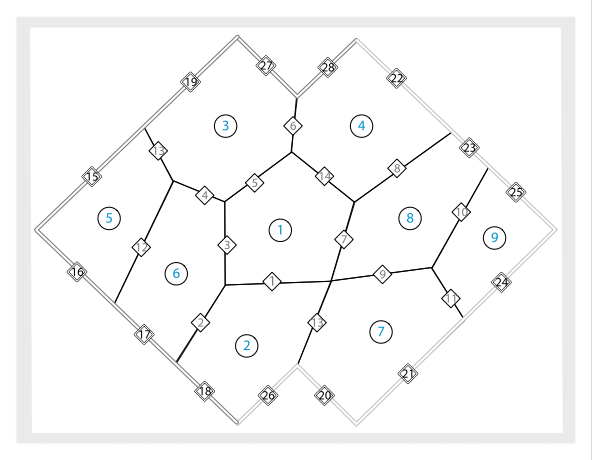


Fig. 6.13 An unstructured mesh for Example 2

Solution

For elements, the neighbors are stored in the increasing index number of shared faces, and interior faces are stored in increasing index number, followed by boundary faces again stored in increasing index number.

element 5
neighbors 6 3
faces 12 13 15 16

element 1
neighbors 2 6 3 8 4
faces 1 3 5 7 14

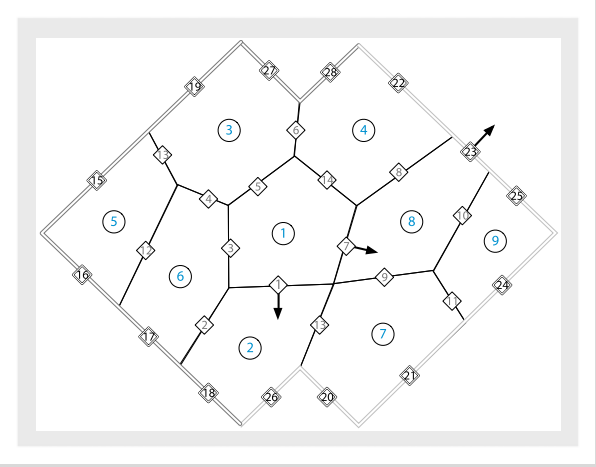


Fig. 6.14 Surface vector direction at faces *1*, *7*, and *23* for Example 2

For a face the owner is the element of lower index and the neighbor is the element of higher index. A Boundary face has only an owner. The direction of the surface vector associated with any face is oriented from owner to neighbor.

face 1
owner 1
neighbor 2

face 7
owner 1
neighbor 8

face 11
owner 7
neighbor 9

face 23 (a boundary face)
owner 8
neighbor -1 (a boundary face has no neighbor)

6.5 Geometric Quantities

In addition to topological data, the finite volume mesh incorporates information about its geometric entities, such as the volume of elements, the area of faces, the centroids of elements and faces, the alignment of faces with the vectors joining the centroids of the owner and neighbor elements (Fig. 6.15), etc. The calculations of some of these geometric quantities will be presented next. The type of elements that

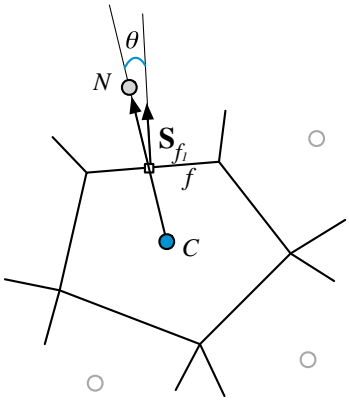


Fig. 6.15 Angle between surface vector and vector joining the centroids of the owner and neighbor elements

can be used in generating the mesh are first described, followed by the techniques employed for computing the geometric information.

6.5.1 Element Types

An element in the finite volume mesh is basically a polyhedron in a three-dimensional mesh (Fig. 6.16) or a polygon in a two-dimensional mesh (Fig. 6.17). The most widely used three-dimensional shapes, as displayed in Fig. 6.16, are tetrahedrons, hexahedrons, prisms, and in some cases general polyhedrons.

The type of faces for these three-dimensional elements, which also represent the type of two-dimensional elements (Fig. 6.17), vary greatly, with the ones that are the most widely used being quadrilaterals, triangles and pentagons, though general polygons have also been adopted in some applications.

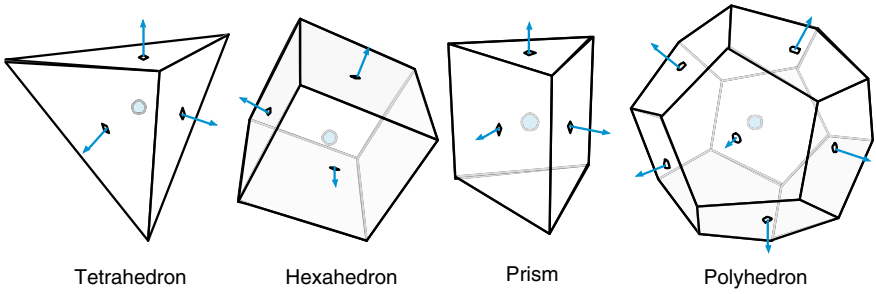


Fig. 6.16 Three-dimensional element types

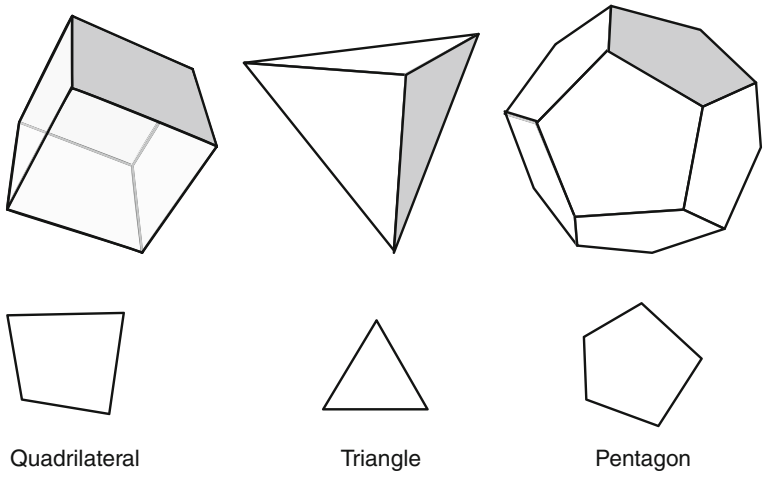


Fig. 6.17 Three-dimensional face types or two-dimensional element types

The computation of geometric factors for such elements and faces will now be detailed. It is worth noting that when working with a two-dimensional mesh the volume of the elements are considered to be the area of the two dimensional elements multiplied by a unit dimension in the off plane direction. Thus the techniques to compute the volume of elements of two-dimensional meshes are exactly those used in computing the surface area of faces of three-dimensional meshes. Other variables arising during discretization, which are solely dependent on geometric quantities, will be presented when needed.

6.5.2 Computing Surface Area and Centroid of Faces

The general shape of an element face in a three-dimensional finite volume mesh is a polygon, though triangular and quadrilateral faces are the most widely used. The computation of the surface vector and centroid follows the same procedure for all types of polygons. Basically a point is constructed within the polygon based on the average of all the points that define the polygon. This point is the geometric centre of the polygon $\mathbf{x}_G = (x_G, y_G, z_G)$, which coincides with the centroid of the polygon $\mathbf{x}_{CE} = (x_{CE}, y_{CE}, z_{CE})$ only for some very special shapes, which include triangles. Therefore the geometric centre of k points forming a polygon is computed as

$$\mathbf{x}_G = \frac{1}{k}\sum_{i=1}^{k} \mathbf{x}_i \tag{6.21}$$

Using the geometric centre (Fig. 6.18) a number of triangles are formed with the centre as the apex for each side of the polygon. For each of the triangles the centroid (since for triangles $\mathbf{x}_G$ and $\mathbf{x}_{CE}$ coincide) and area are readily computed. The sum of their areas will give the total area of the polygon. For computing the centroid the area-weighted centroid (or geometric centre) of each sub-triangle are summed over the polygon and divided by the polygon area, yielding the centroid of the polygon. Mathematically this is computed as

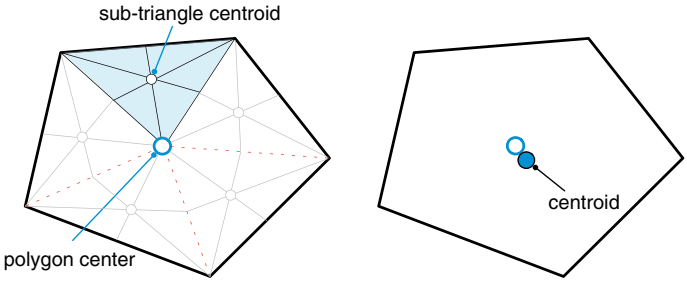


Fig. 6.18 The geometric centre and centroid of a polygon

$$S_f = \sum_{t \sim Sub-triangles(C)} S_t$$
$$(\mathbf{x}_{CE})_f = \frac{\sum_{t \sim Sub-triangles(C)} (\mathbf{x}_{CE})_t * S_t}{S_f} \tag{6.22}$$

6.5.2.1 Surface of a Triangle

The area of a triangle is computed using vector product. The magnitude of the vector product of two vectors represents the area of the parallelogram formed by the two vectors. Thus the area of the triangle is half the magnitude of the vector product of the two vectors. Denoting the position vectors of the three vertices 1, 2, and 3 of the triangle shown in Fig. 6.19 by $\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{r}_3$, respectively, the surface vector of the triangle can be computed as

$$\mathbf{S} = \frac{1}{2}(\mathbf{r}_2 - \mathbf{r}_1) \times (\mathbf{r}_3 - \mathbf{r}_1) = \frac{1}{2}\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix} = S_x\mathbf{i} + S_y\mathbf{j} + S_z\mathbf{k} \tag{6.23}$$

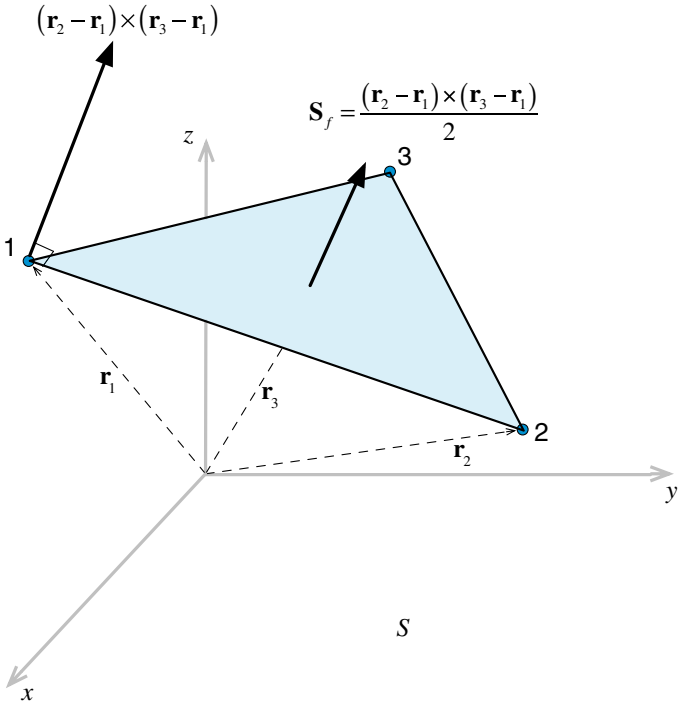


Fig. 6.19 Surface vector and area magnitude of a triangle in a three dimensional space

and the magnitude of the area is given by

$$S = \sqrt{S_x^2 + S_y^2 + S_z^2} \tag{6.24}$$

To know whether the surface vector is pointing outward, its dot product with the position vector that joins the centroid of the element *CE* to the centroid of the surface *ce* is computed. If the sign of the dot product is positive then the surface vector is pointing outward, otherwise it is pointing inward. The same approach may be used to discern the orientation of the surface vector in two dimensions.

For a two dimensional grid the surface area represents the volume of the control cell with a unit depth in the off-plane direction. Therefore the volume of a triangular cell in a two dimensional grid is calculated using

$$\begin{aligned} V &= \frac{1}{2}|(\mathbf{r}_2 - \mathbf{r}_1) \times (\mathbf{r}_3 - \mathbf{r}_1)| = \frac{1}{2}[(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)] \\ &= \frac{1}{2}[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)] \end{aligned} \tag{6.25}$$

Note that the signed volume (or area) will be positive if the vertices 1, 2 and 3 are oriented counterclockwise around the triangle, otherwise it will be negative. Taking the absolute value of the right hand side of Eq. (6.25) will always result in the correct volume value.

Example 4

Compute the centroid and area of the polygon shown in Fig. 6.20 *whose coordinates are displayed in* Table 6.2.

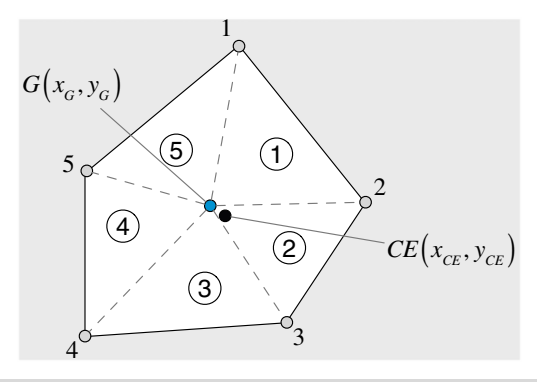


Fig. 6.20 A polygonal element

Table 6.2 Coordinates of the polygonal element for Example 3

Node	1	2	3	4	5
x	1	2.4	2	0.4	0
y	6.4	4.0	0.2	0	4.0

Solution

For this polygon $k = 5$. Thus its geometric centre $G(x_G, y_G)$ is located at

$$x_G = \frac{1}{5}(1 + 2.4 + 2 + 0.4 + 0) = 1.16$$
$$y_G = \frac{1}{5}(6.4 + 4 + 0.2 + 0 + 4) = 2.92$$

The polygon is decomposed into 5 triangles of apex $G(x_G, y_G)$. The centroid of triangle 1 is located at

$$x_{G1} = \frac{1}{3}(1.16 + 1 + 2.4) = 1.52$$
$$y_{G1} = \frac{1}{3}(2.92 + 6.4 + 4) = 4.44$$

In a similar way the centroids of other triangles are found and are presented in Table 6.3.

Table 6.3 Coordinates of the triangles' centroids

Triangle	1	2	3	4	5
x-centroid	1.52	1.85333	1.18666	0.52	0.72
y-centroid	4.44	2.37333	1.04	2.30666	4.44

The areas of the triangles are found using Eq. (6.25) and for triangle 1 is given by

$$S_1 = \frac{1}{2}[x_G(y_2 - y_1) + x_2(y_1 - y_G) + x_1(y_G - y_2)]$$
$$= \frac{1}{2}[1.16(4 - 6.4) + 2.4(6.4 - 2.92) + 1(2.92 - 4)] = 2.244$$

In a similar way the areas for the other triangles are found and are presented in Table 6.4.

Table 6.4 Areas of the various triangles

Triangle	1	2	3	4	5
Area	2.244	2.14	2.62	2.104	1.932

The area of the polygon is found to be

$$S_t = \sum_{i=1}^{5} S_i = 2.244 + 2.14 + 2.62 + 2.104 + 1.932 = 11.04$$

The coordinates of its centroid can be obtained as

$$x_C = \frac{1}{S_t} \sum_{i=1}^{5} S_i x_{Ci} = 1.174925$$

$$y_C = \frac{1}{S_t} \sum_{i=1}^{5} S_i y_{Ci} = 2.825940$$

The difference between the geometric centre and centroid is clear.

6.5.2.2 Volume and Centroid of Elements

The general procedure followed to compute the volume and centroid of a general polyhedron is conceptually simple. The process starts by computing the location of the geometric centre of the polyhedron element and decomposing it into a number of polygonal pyramids. As shown in Fig. 6.21, each polygonal pyramid is formed of the geometric centre as the apex and a polygonal face of the element as the base, with its side faces being triangles.

For a polygonal pyramid, the volume and centroid are readily computed. The volume is calculated as $(1/3) \times Base \times Height$. The base is basically one of the surfaces of the element, while the sub-element pyramid centroid, measured from the centroid of the base, is situated at $1/4$ of the line joining the centroid of the base to the apex of the pyramid. The volume of the polyhedron element is the sum of the volumes of the polygonal pyramids. As for the centroid it is computed as the

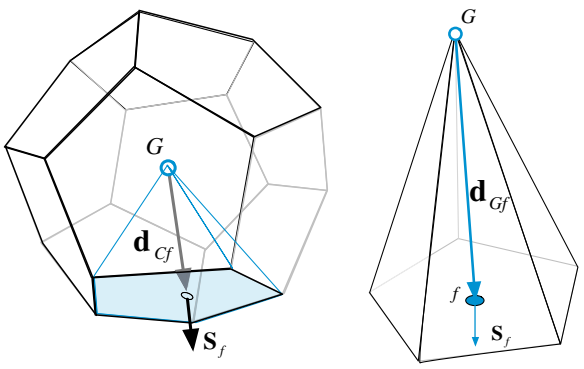


Fig. 6.21 A sub-element pyramid

volume-weighted average of the centroids of the pyramids. Mathematically this is computed from the following relations:

$$\begin{aligned}
\mathbf{x}_G &= \frac{1}{k}\sum_{i=1}^{k}\mathbf{x}_i \\
(\mathbf{x}_{CE})_{pyramid} &= 0.75(\mathbf{x}_{CE})_f + 0.25(\mathbf{x}_G)_{pyramid} \\
V_{pyramid} &= \frac{\mathbf{d}_{Gf}\cdot\mathbf{S}_f}{3} \\
V_C &= \sum_{\sim Sub-pyramids(C)} V_{pyramid} \\
(\mathbf{x}_{CE})_C = \mathbf{x}_C &= \frac{\sum_{\sim Sub-pyramids(C)} (\mathbf{x}_{CE})_{pyramid} V_{pyramid}}{V_C}
\end{aligned} \tag{6.26}$$

Since the centroid and volume of a polygonal pyramid are easily computed, this procedure allows accurate computations of both the volume and centroid of an element.

6.5.2.3 Face Weighting Factor

Consider the one dimensional finite volume mesh system shown in Fig. 6.22. The values of ϕ are known at the control volume centroids C and F, and are to be used to compute the value of ϕ at the interface f.

A simple linear interpolation will result in the following formula:

$$\phi_f = g_f\phi_F + (1 - g_f)\phi_C \tag{6.27}$$

where

$$g_f = \frac{d_{Cf}}{d_{Cf} + d_{fF}} \tag{6.28}$$

The simplicity of this formula does not extend into multi-dimensional situations as in two or three dimensions the circumstances become a bit more complicated. In this case there is not a unique option for the definition of the geometric weighting factors.

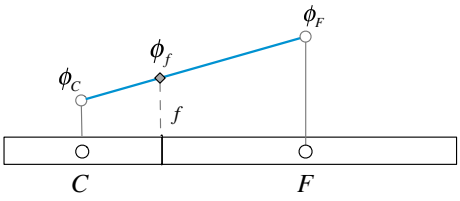


Fig. 6.22 One dimensional mesh system

One choice would be to base the weighting factor on the respective volumes, such that

$$g_f = \frac{V_C}{V_C + V_F} \tag{6.29}$$

This however yields wrong results in certain cases, such as in the configuration shown in Fig. 6.23.

Another difficulty arises when the points C, f, and F are not collinear as depicted in Fig. 6.24a.

A better alternative for such cases, as displayed in Fig. 6.24b, is to base the interpolation on the normal distances to the face, i.e., Cf' and Ff''. Thus the interpolation factor is computed as

$$g_f = \frac{\mathbf{d}_{Cf} \cdot \mathbf{e}_f}{\mathbf{d}_{Cf} \cdot \mathbf{e}_f + \mathbf{d}_{fF} \cdot \mathbf{e}_f} \tag{6.30}$$

where $\mathbf{e}_f$ is the surface unit vector given by

$$\mathbf{e}_f = \frac{\mathbf{S}_f}{\|\mathbf{S}_f\|} \tag{6.31}$$

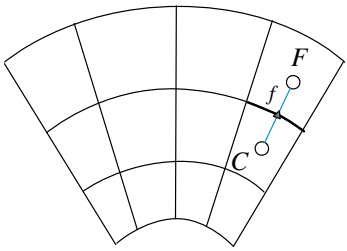


Fig. 6.23 Axisymmetric grid system

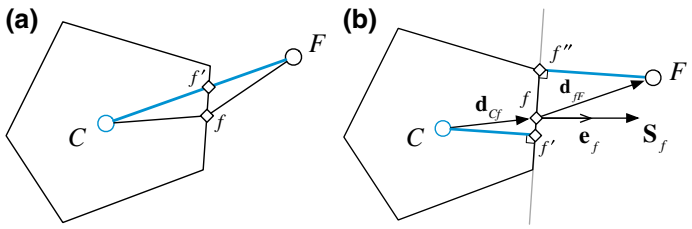


Fig. 6.24 Two dimensional control volume with the points C, f, and F being non collinear

Example 5

Compute the weighing factor using Eqs. (6.26) *and* (6.27) *for the two triangular elements shown in* Fig. 6.25 (Table 6.5).

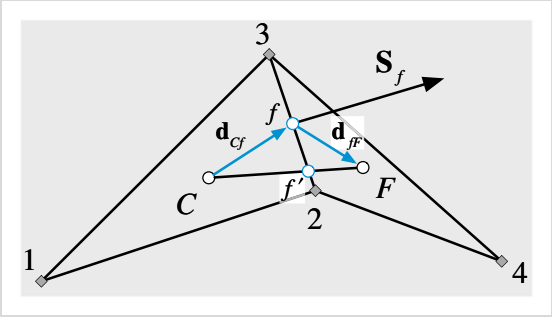


Fig. 6.25 Two neighboring polygonal elements

Table 6.5 Coordinates of the triangular elements for Example 4

Node	1	2	3	4
x	0	1.2	1	2
y	0	0.4	1	0.1

Solution

To calculate the interpolation factor using Eq. (6.26) the volumes of the two elements are needed and are computed as

$$\begin{aligned} V_C &= \frac{1}{2}[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)] \\ &= \frac{1}{2}[0 + 1.2(1 - 0) + 1(0 - 0.4)] = 0.4 \\ V_F &= \frac{1}{2}[x_2(y_4 - y_3) + x_4(y_3 - y_2) + x_3(y_2 - y_4)] \\ &= \frac{1}{2}[1.2(0.1 - 1) + 2(1 - 0.4) + 1(0.4 - 0.1)] = 0.42 \\ g_f &= \frac{V_C}{V_C + V_F} = \frac{0.4}{0.4 + 0.42} = 0.4878 \end{aligned}$$

The calculation of g_f based on Eq. (6.27) is more involved. The centroids of the two control volumes are required and are calculated as

$$x_C = \frac{1}{3}(0 + 1 + 1.2) = 0.7333 \qquad y_C = \frac{1}{3}(0 + 1 + 0.4) = 0.4666$$

$$x_F = \frac{1}{3}(1.2 + 1 + 2) = 1.4 \qquad y_F = \frac{1}{3}(0.4 + 1 + 0.1) = 0.5$$

The face centroid is also required and is found to be

$$x_f = \frac{1}{2}(1 + 1.2) = 1.1 \qquad y_f = \frac{1}{2}(1 + 0.4) = 0.7$$

The surface vector is calculated as

$$\mathbf{S}_f = (y_3 - y_2)\mathbf{i} - (x_3 - x_2)\mathbf{j} = 0.6\mathbf{i} + 0.2\mathbf{j}$$

The unit vector normal to the face becomes

$$\mathbf{e}_f = \frac{\mathbf{S}_f}{S_f} = \frac{0.6\mathbf{i} + 0.2\mathbf{j}}{\sqrt{0.6^2 + 0.2^2}} = 0.949\mathbf{i} + 0.316\mathbf{j}$$

The distance from the centroids of the control volumes to the face centroid are given by

$$\mathbf{d}_{Cf} = (x_f - x_C)\mathbf{i} + (y_f - y_C)\mathbf{j} = 0.3667\mathbf{i} + 0.2334\mathbf{j}$$
$$\mathbf{d}_{fF} = (x_F - x_f)\mathbf{i} + (y_F - y_f)\mathbf{j} = 0.3\mathbf{i} - 0.2\mathbf{j}$$

The interpolation factor using Eq. (6.27) is found as

$$g_f = \frac{\mathbf{d}_{Cf} \cdot \mathbf{e}_f}{\mathbf{d}_{Cf} \cdot \mathbf{e}_f + \mathbf{d}_{fF} \cdot \mathbf{e}_f} = \frac{(0.3667\mathbf{i} + 0.2334\mathbf{j}) \cdot (0.949\mathbf{i} + 0.316\mathbf{j})}{(0.6667\mathbf{i} + 0.0334\mathbf{j}) \cdot (0.949\mathbf{i} + 0.316\mathbf{j})} = 0.6556$$

the difference in values is obvious.

6.6 Computational Pointers

6.6.1 uFVM

In ufvm, the processing of all geometric and topological data is performed in one routine denoted by cfdProcessOpenFoamMesh, which is executed right after reading the OpenFOAM® mesh with cfdReadOpenFoamMesh. Reading the OpenFOAM® mesh in its native form, requires reading the various files defining an OpenFOAM® mesh, namely and in that order, *points*, *faces*, *owners*, *neighbours*, and *boundaries* files.

In cfdProcessOpenFoamMesh the face geometry is initially processed (centroid, area and surface vector), as shown in Listing 6.1.

```
%
% Process basic Face Geometry
%
numberOfFaces = theMesh.numberOfFaces;
for iFace=1:numberOfFaces
    iNodes = theMesh.faces(iFace).iNodes;
    numberOfiNodes = length(iNodes);
    %
    % Compute a rough centre of the face
    %
    centre = [0 0 0]';
    for iNode=iNodes
       centre = centre + theMesh.nodes(iNode).centroid;
    end
    centre = centre/numberOfiNodes;

    centroid = [0 0 0]';
    Sf = [0 0 0]';
    area = 0;
    %
    % using the center compute the area and centroid
    % of virtual triangles based on the centre and the
    % face nodes
    %
    for iTriangle=1:numberOfiNodes
        point1 = centre;
        point2 = theMesh.nodes(iNodes(iTriangle)).centroid;
        if(iTriangle<numberOfiNodes)
  point3 = theMesh.nodes(iNodes(iTriangle+1)).centroid;

        else
            point3 = theMesh.nodes(iNodes(1)).centroid;
        end
        local_centroid = (point1+point2+point3)/3;
        local_Sf  = 0.5*cross(point2-point1,point3-point1);
        local_area = cfdMagnitude(local_Sf);

       centroid = centroid + local_area*local_centroid;
        Sf = Sf + local_Sf;
        area = area + local_area;
    end
    centroid = centroid/area;

    %
    theMesh.faces(iFace).centroid = centroid;
    theMesh.faces(iFace).Sf = Sf;
    theMesh.faces(iFace).area = area;
end
```

Listing 6.1 Processing of basic face geometry

This is followed by processing the basic element geometry (Listing 6.2). The element volume is computed by subdividing it into non-overlapping pyramids, whose geometric characteristics can be easily computed.

```
%
%  compute volume and centroid of each element
%
numberOfElements = theMesh.numberOfElements;
for iElement=1:numberOfElements

    iFaces = theMesh.elements(iElement).iFaces;
    %
    % Compute a rough centre of the element
    %
    centre = [0 0 0]';
    for iFace=1:length(iFaces)
      centre = centre + theMesh.faces(iFace).centroid;
    end
    centroid = [0 0 0]';
    Sf = [0 0 0]';
    centre = centre/length(iFaces);
    % using the centre, compute the area and centroid
    % of virtual triangles based on the centre and the
    % face nodes
    %
    localVolumeCentroidSum = [0 0 0]';
    localVolumeSum = 0;
    for iFace=1:length(iFaces)
        localFace = theMesh.faces(iFaces(iFace));
        localFaceSign = theMesh.elements(iElement).faceSign(iFace);
        Sf = localFace.Sf*localFaceSign;
        Cf = localFace.centroid - centre;
        % calculate face-pyramid volume
        localVolume = Sf'*Cf/3;
        % Calculate face-pyramid centre

        localCentroid  =  0.75*localFace.centroid  +
0.25*centre;

        %Accumulate volume-weighted face-pyramid centre
         localVolumeCentroidSum = localVolumeCentroidSum +
localCentroid*localVolume;

         % Accumulate face-pyramid volume
        localVolumeSum = localVolumeSum + localVolume;
    end
    centroid = localVolumeCentroidSum/localVolumeSum;
    volume = localVolumeSum;
    %
    theMesh.elements(iElement).volume = volume;
    theMesh.elements(iElement).centroid = centroid;
end
```

Listing 6.2 Processing of basic element geometry

6.6.2 OpenFOAM®

As OpenFOAM® [10] uses an unstructured grid platform, all its geometric entities, such as elements, volumes, areas, and centroids, as well as face weighting factors, have to be evaluated and stored. This section provides an overview of the geometric relations needed in the evaluation of the geometric quantities used in OpenFOAM®.

6.6.2.1 Area and Centroid of Faces

In order to evaluate areas and centers for a generic polygon face, the face is decomposed into a series of triangular faces. The overall polygon face metrics are calculated by summing up the properties of each triangular portion. Thus Eqs. (6.21)–(6.23) have to be applied for each computational cell.

OpenFOAM® constructs centers of faces and calculates their areas in the file "$FOAM_SRC/OpenFOAM/meshes/primitiveMesh/primitiveMeshFaceCentresAndAreas.C" in which the following function is defined (Listing 6.3):

```
void Foam::primitiveMesh::makeFaceCentresAndAreas
(
    const pointField& p,
    vectorField& fCtrs,
    vectorField& fAreas
) const
…
```

Listing 6.3 Function used for defining centers and areas of faces

The function has three arguments with the first, defined as *const,* representing data read from files, while the second and third arguments designate returned lists of objects of dimensions equal to the number of faces in the domain, containing the centers (fCtrs) and areas (fAreas) of the polygonal faces, respectively.

The *pointField* data is a list of all mesh vertices with each vertex defined using three spatial coordinates. The second data needed is the face definition. In OpenFOAM® this is provided in faceList (Listing 6.4), which is a list of identities of the points defining the faces of the mesh.

```
const faceList& fs = faces();

forAll(fs, facei)
{
    const labelList& f = fs[facei];
    label nPoints = f.size();
…
```

Listing 6.4 List of the identities of points defining faces

Then a *for* loop is performed for each face. Inside the loop, the number of points describing the face and their identities are first read in order to access the

coordinates of the corresponding vertices. When a face is defined by only three vertices, then direct calculations of its centroid location and area are performed, as shown in Listing 6.5.

```
…
// If the face is a triangle, do a direct    calculation
for efficiency
// and to avoid round-off error-related problems
if (nPoints == 3)
{
     fCtrs[facei]  =  (1.0/3.0)*(p[f[0]]  +  p[f[1]]  +
p[f[2]]);
     fAreas[facei] = 0.5*((p[f[1]] - p[f[0]])^(p[f[2]] -
p[f[0]]));
}
else
{
…
```

Listing 6.5 Equations used in calculating the geometric center and area of a triangular face

In this case the face center *fCtrs[facei]* is evaluated directly using Eq. (6.21) with $k = 3$, while the face area is calculated using Eq. (6.23) where the symbol "^" represents vector product.

For a generic polygonal shape, the face is first decomposed into triangles. For that purpose, OpenFOAM® defines estimated centers for faces based on the average value of vertices defining the face, as displayed in Listing 6.6.

```
vector sumN = vector::zero;
scalar sumA = 0.0;
vector sumAc = vector::zero;

point fCentre = p[f[0]];
for (label pi = 1; pi < nPoints; pi++)
{
    fCentre += p[f[pi]];
}

fCentre /= nPoints;
```

Listing 6.6 Compute estimated centers for faces

Listing 6.7 indicates that a loop over all faces is performed for calculating the geometric centers and areas of the triangles into which these faces are decomposed.

```
for (label pi = 0; pi < nPoints; pi++)
{
    const point& nextPoint = p[f[(pi + 1) % nPoints]];

    vector c = p[f[pi]] + nextPoint + fCentre;
    vector n = (nextPoint - p[f[pi]])^(fCentre - p[f[pi]]);
    scalar a = mag(n);

    sumN += n;
    sumA += a;
    sumAc += a*c;
}
```

Listing 6.7 Decomposing the polygonal faces into triangles and calculating the geometric centers and areas of these triangles

This is done by finding the face center "c" of each triangle using Eq. (6.21) (the factor $1/3$ is applied later), the face normal vector "**n**" and its magnitude "a" that defines the triangle area as stated by Eq. (6.23) (again the factor $1/2$ will be applied later). All metrics of the decomposed triangles are then summed up and normalized in accordance with Eq. (6.22), where now the factors $1/3$ and $1/2$ are applied, as depicted in Listing 6.8.

```
// This is to deal with zero-area faces. Mark very small
faces
// to be detected in e.g., processorPolyPatch.
if (sumA < ROOTVSMALL)
{
    fCtrs[facei] = fCentre;
    fAreas[facei] = vector::zero;
}
else
{
    fCtrs[facei] = (1.0/3.0)*sumAc/sumA;
    fAreas[facei] = 0.5*sumN;
}
```

Listing 6.8 Calculating the centroids and areas of faces

In case of a degenerate face, i.e., when "sumA < a very small number", a rescue value is set for the face.

6.6.2.2 Volume and Centroid of Elements

After computing the normals, areas, and centers of faces, it is possible to evaluate the metrics of cells. Similar to faces, the basic idea for a polyhedral cell is to decompose it into a sum of tetrahedra elements. The operations that calculate

volumes and centroids of elements are then defined in the file "$FOAM_SRC/OpenFOAM/meshes/primitiveMesh/primitiveMeshCellCentresAndVols.C" using the function shown in Listing 6.9.

```
void Foam::primitiveMesh::makeCellCentresAndVols
(
    const vectorField& fCtrs,
    const vectorField& fAreas,
    vectorField& cellCtrs,
    scalarField& cellVols
) const
{
…
```

Listing 6.9 Function used to calculate volumes and centers of elements

This function uses four arguments where the first two represent, respectively, centers and areas of faces. The remaining two arguments return objects containing centers and volumes of cells. As shown in Listing 6.10, the "for" loops are defined twice. This redundancy is related to the LDU addressing used in OpenFOAM®, which will be introduced and described in the Chap. 7.

```
…
    // Clear the fields for accumulation
    cellCtrs = vector::zero;
    cellVols = 0.0;

    const labelList& own = faceOwner();
    const labelList& nei = faceNeighbour();

    vectorField cEst(nCells(), vector::zero);
    labelField nCellFaces(nCells(), 0);

    forAll(own, facei)
    {
        cEst[own[facei]] += fCtrs[facei];
        nCellFaces[own[facei]] += 1;
    }

    forAll(nei, facei)
    {
        cEst[nei[facei]] += fCtrs[facei];
        nCellFaces[nei[facei]] += 1;
    }

    forAll(cEst, celli)
    {
        cEst[celli] /= nCellFaces[celli];
    }
```

Listing 6.10 Compute estimated cell centers

In order to calculate the cells centroids and volumes, the first step is the evaluation of $\mathbf{x}_G$, the geometric centers of cells denoted in Listing 6.10 by cEst, using

the first relation in Eq. (6.25). Once the $\mathbf{x}_G$ values are obtained, calculations of the pyramids' volumes and cell centroids proceed by applying Eq. (6.25) using the code displayed in Listing 6.11.

```
    forAll(own, facei)
    {
        // Calculate 3*face-pyramid volume
        scalar pyr3Vol =
                    max(fAreas[facei] & (fCtrs[facei] -
cEst[own[facei]]), VSMALL);

        // Calculate face-pyramid centre
        vector pc = (3.0/4.0)*fCtrs[facei] +
(1.0/4.0)*cEst[own[facei]];

        // Accumulate volume-weighted face-pyramid centre
        cellCtrs[own[facei]] += pyr3Vol*pc;

        // Accumulate face-pyramid volume
        cellVols[own[facei]] += pyr3Vol;
    }

    forAll(nei, facei)
    {
        // Calculate 3*face-pyramid volume
        scalar pyr3Vol =
                  max(fAreas[facei] & (cEst[nei[facei]] -
fCtrs[facei]), VSMALL);

        // Calculate face-pyramid centre
        vector pc = (3.0/4.0)*fCtrs[facei] +
(1.0/4.0)*cEst[nei[facei]];

        // Accumulate volume-weighted face-pyramid centre
        cellCtrs[nei[facei]] += pyr3Vol*pc;

        // Accumulate face-pyramid volume
        cellVols[nei[facei]] += pyr3Vol;
    }
```

Listing 6.11 Calculating cell centers and volumes

In the above code, fCtrs corresponds to $\mathbf{x}_{CE}$ while "cEst—fCtrs" corresponds to the distance vector $\mathbf{d}_{Gf}$ displayed in Fig. 6.21. The final values are obtained, as shown in Listing 6.12, by dividing cell centroids by cell volumes and then dividing cell volumes by 3.

```
    cellCtrs /= cellVols;
    cellVols *= (1.0/3.0);
```

Listing 6.12 The final values of cell centroids and volumes

The mesh data structure for uFVM and OpenFOAM® will be described in detail in Chap. 7.

6.7 Closure

The geometric data defining a finite volume mesh were presented in this chapter. It was stressed that the finite volume mesh is not simply the set of non-overlapping elements and nodes. It also includes the set of all geometric quantities with information about their topologies. The collection of all this represents the infrastructure needed by the equation discretization method adopted in this book, namely the Finite Volume Method (FVM).

6.8 Exercises

Exercise 1
Compare the equations presented in the book to the ones used in OpenFOAM® for computing the interpolation weights at the faces, the owner-Neighbor element distances, and the non orthogonal coefficient. These can be found in OpenFOAM® using Doxygen [14] in the functions makeWeights(), makeDeltaCoeffs(), and makeNonOrthDeltaCoeffs().

Exercise 2
Write a program that reads an OpenFOAM® mesh and checks that for each element the sum of the surface vectors is zero.

Exercise 3
Start by reading a mesh into uFVM and then investigate the mesh structure (use `m = cfdGetMesh` to get access to the mesh data, and then investigate the structure of an element, a face, and a vertex).

References

1. Thompson JF, Warsi Z, Mastin C (1985) Numerical grid generation. Elsevier Science Publishers, New York
2. Cheng S, Dey T, Shewchuk JR (2012) Delaunay mesh generation. CRC Press, Boca Raton
3. Thompson JF, Soni BK, Weatherill NP (eds) (1999) Handbook of grid generation, Chapter 17. CRC Press, Boca Raton
4. George PL (1991) Automatic mesh generation. Wiley, New York
5. Frey P, George PL (2010) Mesh generation. Wiley, New York
6. Bern M, Plassmann P (2000) Mesh generation. Handbook of computational geometry. Elsevier Science, North Holland
7. Mavriplis DJ (1996) Mesh generation and adaptivity for complex geometries and flows. In: Peyret R (ed) Handbook of computational fluid mechanics. Academic Press, London
8. Thompson JF, Soni BK, Weatherill NP (1998) Handbook of grid generation. CRC Press, Boca Raton
9. Chappell JA, Shaw JA, Leatham M (1996) The generation of hybrid grids incorporating prismatic regions for viscous flow calculations. In: Soni BK, Thompson JF, Hauser J,

Eiseman, PR (eds) Numerical grid generation in computational field simulations. Mississippi State University, MS, pp 537–546
10. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
11. Mavriplis DJ (1997) Unstructured grid techniques. Ann Rev Fluid Mech 29:473–514
12. Lerner RG, Trigg GL (1994) Encyclopaedia of physics. VHC, New York
13. Byron F, Fuller R (1992) Mathematics of classical and quantum physics. Dover Publications, Mineola
14. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 7
The Finite Volume Mesh in OpenFOAM® and uFVM

Abstract The implementation of the finite volume mesh can follow many directions whether in the definition of the mesh fields, the storing of the variables, or even in determining the connectivity relations. This chapter aims at outlining the design decisions that shape the implementation of two CFD codes, uFVM an educational unstructured Finite Volume code and OpenFOAM® an industrial-strength open source code. The two codes are thus presented, initially in terms of their data structure and memory management schemes, and then in terms of how cases are setup. Finally the format of the system of equations generated by each of the two codes are detailed. The reader will notice that while uFVM shares many of the implementation details with OpenFOAM®, its simplicity allows for the use of simpler implementation techniques and data structure.

7.1 uFVM

The unstructured finite volume code uFVM was written to illustrate the various numerical techniques and algorithms, which collectively form a CFD program. Furthermore its numerics are in many ways similar to those in OpenFOAM® [1], making it a good vehicle to understand and present the internals of OpenFOAM®. The main data structures used in uFVM generally mirrors those in OpenFOAM® especially in terms of mesh fields and boundary conditions. Still differences between uFVM and OpenFOAM® will be used to underline various options available to CFD coders and thus to better present some implementation details.

7.1.1 An OpenFOAM® Test Case

The uFVM code is capable of reading an OpenFOAM® mesh included as part of any OpenFOAM® test case. An OpenFOAM® test case is a directory that generally

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_7

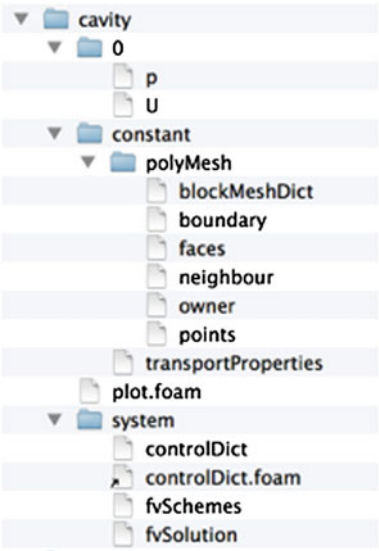


Fig. 7.1 The **cavity** OpenFOAM® case

contains at least three folders. Figure 7.1 shows the innards of the **cavity** test case folder, which consists of the following three sub-folders:

The '**time**' directories contain initialization and boundary condition information about the fields used in the test case. It is also the folder where results for various time steps are stored. The name of each sub-folder refers to the time at which simulation was performed. For example, in the **cavity** tutorial shown in Fig. 7.1, the velocity field **U** and pressure field **p** are initialized from files in the '**0**' folder **0/U** and **0/p**, respectively.

The **system** directory contains at least three files with information about the case setup, the schemes to be used, and the various solution parameters. These files are: (i) **controlDict** which is concerned with the general control parameters of the test case such as the simulation starting and ending times, the time step to be used, and required parameters for data output; (ii) **fvSchemes** in which the discretization schemes used in the simulation are defined; (iii) **fvSolution** that contains information related to the solution algorithms and relaxations used during simulation.

The **constant** directory contains information about relevant physical properties, e.g., **transportProperties**, and the data describing the grid system used within a folder denoted by **polyMesh**.

For the purpose of understanding the finite volume mesh structure in uFVM, attention will be focussed on the **polyMesh** folder in which the information needed to construct the finite volume mesh is defined.

7.1.2 The polyMesh Folder

The polyMesh subdirectory contains the following files:

points
The file **points** is a list of vectors denoting the cell vertices, with vertex **0** being the the first vector in the list, vertex **1** the second vector, etc. The format of the **points** file is shown in Listing 7.1.

```
#number of points
(
(#x #y #z)
…
)
```

Listing 7.1 Storing vertices as vectors of coordinates x, y and z

An example **points** file is shown in Listing 7.2.

```
1074
(
(32 16 0.9377383239)
(33.9429245 16.11834526 0.9377383239)
(35.84160614 16.46798134 0.9377383239)
(37.67648315 17.04080009 0.9377383239)
(39.42799377 17.82870483 0.9377383239)
(41.07658768 18.82359314 0.9377383239)
(…)
…
)
```

Listing 7.2 An example showing how vertices are stored

faces
The file **faces** represents a list of faces, with each face described by a list of indices to vertices in the **points** list, where again, the first entry in the list represents face **0**, the second entry represents face **1**, etc. The format of the **faces** file is shown in Listing 7.3.

```
#number of faces
(
#number of points for face 1 (#p1 #p2 #p3 …)
#number of points for face 2 (#p1 #p2 #p3 … )
…
)
```

Listing 7.3 Storing faces in the form of points of which the face is composed and their indices

Example of a **faces** file is shown in Listing 7.4.

```
3290
(
4(36 573 589 52)
4(41 578 634 97)
4(44 81 618 581)
4(30 82 619 567)
4(121 50 587 658)
4(39 120 657 576)
…
)
```

Listing 7.4 An example showing how faces are stored

owners

The file **owners** is a list in which the owner of faces are stored (Listing 7.5). The position of the owner in the list refers to the face it belongs to. Thus, the owner of face **0** is the index stored in the first entry, the owner of face **1** is the index stored in the second entry, etc. The number of owners is equal to the total number of faces (interior + boundary faces).

The number of Elements is equal to the largest index of the owners.

```
#number of owners
(
#owner of face1
#owner of face2
…
)
```

Listing 7.5 The format used to store owners

An example of an **owners** file is shown in Listing 7.6.

```
3290
(
0
1
2
3
4
5
6
…
)
```

Listing 7.6 An example showing how owners are stored

The total number of cells (**nCells**) in the domain can be found in the **owners** file header as shown in Listing 7.7.

```
note "nPoints:1074  nCells:918  nFaces:3290  nInternalFaces:1300";
```

Listing 7.7 Header of **owners** file

neighbours
A list of neighbor cell labels (Listing 7.8). The number of neighbors is basically equal to the number of interior faces.

```
#number of neighbour
(
#neighbour of face1
#neighbour of face2
…
)
```

Listing 7.8 The format used to store neighbors

An example of a **neighbours** file is shown in Listing 7.9.

```
1300
(
22
68
29
96
31
34
…
)
```

Listing 7.9 An example showing how neighbors are stored

boundary
File **boundary** lists the boundaries of the domain, with the faces of each boundary type referred to as a patch and assigned a name. The type of each boundary patch (type) is declared along with its number of faces (nFaces) and the starting face (startFace), which refers to the index of the first face in the list (Listing 7.10).

```
#boundary patch name
{
    type #patchtype;
    nFaces #number of face in patch set;
    startFace #starting face index for patch;
}
```

Listing 7.10 Format of a boundary patch

An example of a wall-type boundary patch is depicted in Listing 7.11.

```
wall-4
{
    type            wall;
    nFaces          100;
    startFace       1300;
}
```

Listing 7.11 An example of a wall-type boundary patch

7.1.3 The uFVM Mesh

In uFVM an OpenFOAM® mesh is read using the **cfdReadOpenFoamMesh** script. The script starts by reading the **points** file into the arrays of struct **nodes** where the (x, y, z) data is stored. Then file **faces** is read and the indices of the nodes are stored into the arrays of struct **faces**. Information about the face patches is then read from the file **boundary**. Finally the files **owners** and **neighbours** are read and the struct **elements** is composed. The loaded data is processed to compute additional geometric and topological information in script **cfdProcessOpenFoamMesh**. The details of a mesh generated over an elbow and read by uFVM is shown in Listing 7.12.

```
m = cfdReadOpenFoamMesh('elbow')
m =
                    nodes: [1x1074 struct]
            numberOfNodes: 1074
            caseDirectory: 'elbow'
            numberOfFaces: 3290
         numberOfElements: 918
                    faces: [1x3290 struct]
    numberOfInteriorFaces: 1300
               boundaries: [1x6 struct]
       numberOfBoundaries: 6
          numberOfPatches: 6
                 elements: [1x918 struct]
        numberOfBElements: 1990
           numberOfBFaces: 1990
```

Listing 7.12 Information uFVM can display about a mesh

As shown in Fig. 7.2 the mesh can be plotted using the cfdPlotMesh command.

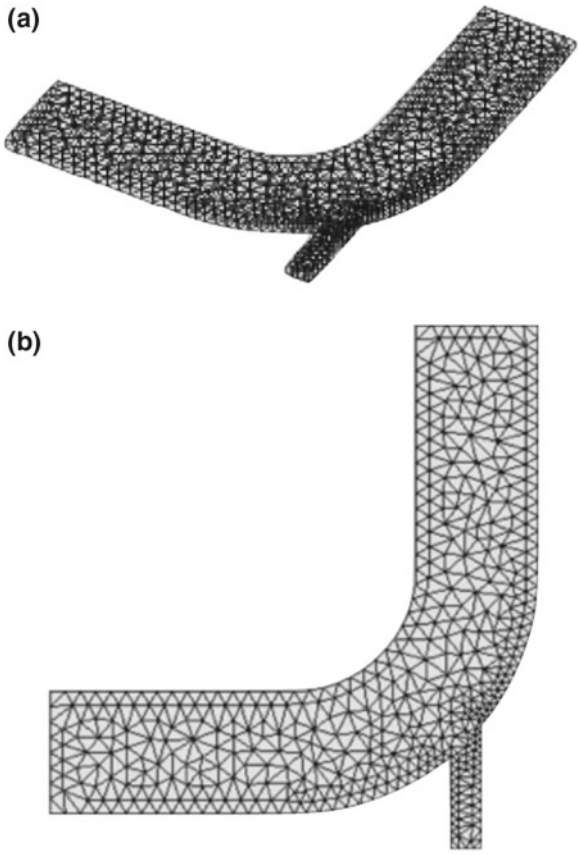


Fig. 7.2 **a** Three dimensional and **b** two dimensional views of the mesh over an elbow displayed using the cfdPlotMesh command

Example of information stored in struct **nodes** is presented in Listing 7.13 by displaying values for the node of index **1**.

```
n1= m.nodes(1)
n1 =
     centroid: [3x1 double]
        index: 1
       iFaces: [172 328 1355 1386 1677 1891 1893]
    iElements: [112 219 220]
```

Listing 7.13 An example of information related to a node

As shown, the centroid contains the coordinates of the node in question, while iFaces and iElements are lists of indices of the faces and elements, respectively, that are connected to the node.

Information stored in struct **faces** can be obtained in a similar way. For example, the attributes of the face indexed **3** are given by (Listing 7.14).

```
m.faces(3)
ans =
                 iNodes: [45 82 619 582]
                  index: 3
                 iOwner: 3
             iNeighbour: 30
               centroid: [3x1 double]
                     Sf: [3x1 double]
                   area: 5.3046
                     CN: [3x1 double]
                geoDiff: 4.5940
                      T: [3x1 double]
                     gf: 0.4226
               walldist: 0
    iOwnerNeighbourCoef: 1
    iNeighbourOwnerCoef: 1
```

Listing 7.14 An example of information related to a face

The above example shows that the struct **faces** contains the list of indices of the nodes defining the face, the indices of the owner and neighbor to the face, the centroid of the face, its surface vector, and its area. In addition, the distance vector **`T`** joining the centroids of the owner and neighbor elements, the geometric factor **`gf`**, the distance vector **`CN`** from the owner element centroid to the surface centroid, and the normal distance to the wall **`walldist`** of the owner element centroid are also stored. Other components will be described later as needed. Moreover, it should be noted that the neighbor for a boundary face is set to $-\mathbf{1}$.

As displayed in Listing 7.15 for the element with index **20**, which is a boundary element, the struct **elements** contains three lists of indices for neighboring elements (iNeighbours), faces (iFaces), and nodes (iNodes).

```
m.elements(20)
ans =
                  index: 20
           iNeighbours: [100 103]
                iFaces: [33 34 1317 1493 1494]
                iNodes: [168 79 616 705 617 80]
                volume: 3.2484
              faceSign: [1 1 1 1 1]
    numberOfNeighbours: 2
              centroid: [3x1 double]
```

Listing 7.15 An example of information related to an element

The order of the element indices and faces indices are synchronized such that elements are related to faces in the same order, and boundary faces are defined at the end of the list of faces. The struct **elements** also stores information about the element centroid and its volume. The **faceSign** list indicates whether the element is an owner (**+1**) or neighbor (**−1**) for the respective faces.

Elements can be identified on the mesh as in Fig. 7.3 using the following command (Listing 7.16):

```
cfdPlotElements([20 300])
```

Listing 7.16 Script needed for uFVM to display selected elements [here elements **(20)** and **(300)**] highlighted on the mesh

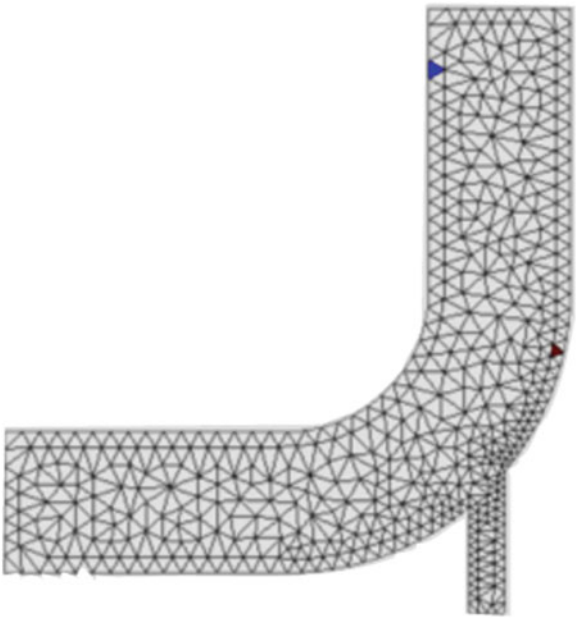


Fig. 7.3 A two dimensional view of the mesh over an elbow highlighting elements **(20)** and **(300)**

Information about element **(20)** were displayed above while the attributes of element **(300)**, which has two boundary faces, are as shown in Listing 7.17.

```
m.elements(300)
ans =
                 index: 300
           iNeighbours: [278 302 590]
                iFaces: [407 435 436 2053 2054]
                iNodes: [283 820 679 142 290 827]
                volume: 1.9083
              faceSign: [-1 1 1 1 1]
    numberOfNeighbours: 3
              centroid: [3x1 double]
```

Listing 7.17 An example of information related to element **(300)** with two boundary faces

Finally information about the various boundary patches, i.e., the list of the boundary faces is stored in struct **boundaries**. Each boundary array contains information about the starting index of the first boundary face, the number of boundary faces belonging to the patch, in addition to the physical type of the patch and its name. For the example considered, information about boundary patch **{1}** is shown in Listing 7.18.

```
>> m. boundaries(1)
ans =
          userName: 'wall-4'
             index: 1
              type: 'wall'
    numberOfBFaces: 100
         startFace: 1301
```

Listing 7.18 An example of information stored for a wall boundary

Moreover, information about the first boundary face, for example, is obtained as shown in Listing 7.19.

```
>> m.faces(1301)
ans =
                 iNodes: [38 53 590 575]
                  index: 1301
                 iOwner: 1
              iNeighbour: -1
                centroid: [3x1 double]
                      Sf: [3x1 double]
                    area: 3.7510
                      CN: [3x1 double]
                 geoDiff: 5.6264
                       T: [3x1 double]
                      gf: 1
               walldist: 0.6667
    iOwnerNeighbourCoef: []
    iNeighbourOwnerCoef: []
```

Listing 7.19 Information about a boundary face

To be noted is the index of the first boundary face, which is **1300+1** since in Matlab® arrays start at index **1** while in the C computer language arrays start at index **0**.

Thus to loop over the faces of a certain patch, the index of the starting face and the number of faces from the struct **boundary** associated with the said patch are needed. For example to loop over the faces of patch 2, the following script (Listing 7.20) can be used:

```
theMesh = cfdGetMesh;
iPatch = 2;
iBFaces = cfdGetFaceIndicesForBoundaryIndex(iPatch)
for iBFace=iBFaces
      theBFace = theMesh.faces(iBFace);
      disp(theBFace) %display theBFace internal fields
end
```

Listing 7.20 Looping over boundary patch faces

cfdGetFaceIndicesForBoundaryIndex is defined in Listing 7.21 as follows:

```
theIndices = cfdGetFaceIndicesForBoundaryIndex(theBoundaryIndex)
%
theBoundary = cfdGetBoundary(theBoundaryIndex);
theNumberOfBFaces = theBoundary.numberOfBFaces;
theStartFace = theBoundary.startFace;
theIndices = [theStartFace:theStartFace+theNumberOfBFaces-1];
%
end
```

Listing 7.21 Indices of boundary faces for a specific patch

7.1.4 uFVM Geometric Fields

In addition to all data stored in the struct **mesh**, information about the model to be solved and the values of the fields of interest should be stored to be accessed when needed. Three types of locale can be identified and fields of different types can be defined on them. Namely for locale (Elements, Faces, and Nodes) and for types (scalars, vectors, and tensors). The various fields are first defined based on their locale.

7.1.4.1 The Element Fields

An element field is constructed using the following script (Listing 7.22):

```
cfdSetupMeshField(theUserName,theLocale,theType,theTimeStep)
```

Listing 7.22 Script needed to construct an element field

where **theUserName** is the name of the field, **theLocale** is the geometric entity over which it is defined **(Elements**, **Faces**, **Nodes)**, **theType** defines the type of the array elements (**Scalar** or **Vector**), and finally the **TimeStep** indicates the relative time of the field (Step0, Step1, etc.). For example, the following script (Listing 7.23):

```
>> UField = cfdSetupMeshField('U:water','Elements','Vector','Step0')
UField =
    userName: 'U:water'
        name: 'U_fluid01'
        type: 'Vector'
      locale: 'Elements'
         phi: [2908x3 double]
```

Listing 7.23 Example of setting up a vector field

sets up a vector field defined over **Elements** at time step 0, i.e., at the current time step.

As shown in Fig. 7.4, the array has the size of the **NumberOfElements+ theNumberOfBoundaryFaces** since the element array will include the values defined for each element in the mesh in addition to each boundary face. The boundary face values represent boundary conditions for that field. These boundary values are grouped in terms of boundary patches.

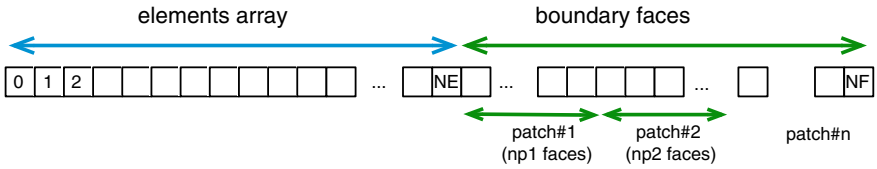


Fig. 7.4 Size of the field array

Generally they can be accessed as follows: For example to initialize the boundary values of the UField at patch **1** to a value **[1 0 0]**, the following should be written (Listing 7.24):

```
% get the mesh
theMesh = cfdGetMesh;
% get information about the boundary patch
theBoundary = theMesh.boundaries(iPatch);
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;
numberOfBFaces = theBoundary.numberOfBFaces;
% Starting face
iFaceStart = theBoundary.startFace;
% get information about starting and ending elements
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
% define the indices as an index array
iBElements = iElementStart:iElementEnd;
>> UField.phi(iBElements,:) =
cfdComputeFormulaAtLocale('[1;0;0]','BPatch1','Vector')
ans =
     1     0     0
     1     0     0
     1     0     0
…
       …
```

Listing 7.24 The script needed to initialize the boundary value of a field in a patch

where the statement in Listing 7.25 given by

```
cfdComputeFormulaAtLocale(theFormula,theLocale,theType)
```

Listing 7.25 Statement used to compute values over a locale

evaluates the values in **theFormula** over **theLocale** and returns and array of the appropriate length of type **theType** (**Scalar** or **Vector**).

7.1.4.2 The Face Fields

A face field is constructed with **theLocale** set to '**Faces**' as (Listing 7.26)

```
cfdSetupMeshField(theUserName,theLocale,theType,theTimeStep)
```

Listing 7.26 Statement used to construct a face field

Again, as shown in Fig. 7.5, the length of the array is equal to **numberOfFaces**, which combines the **numberOfInteriorFaces** plus the sum of all boundary faces.

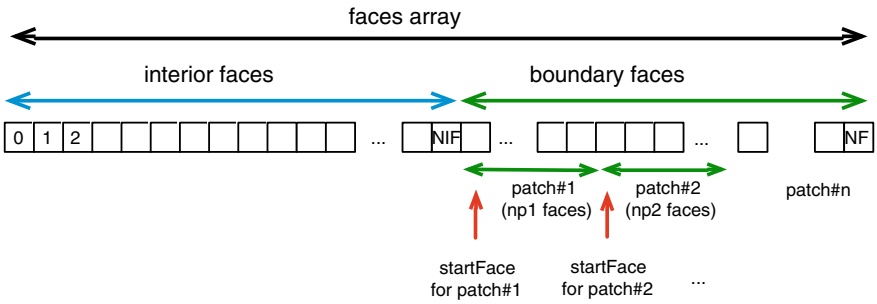


Fig. 7.5 Size of **faces** array

The boundary **faces** for any boundary patch can be accessed as (Listing 7.27)

```
theMesh = cfdGetMesh;
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;

theBoundary = theMesh.boundaries(iPatch);
numberOfBFaces = theBoundary.numberOfBFaces;

%
iFaceStart = theBoundary.startFace;
iFaceEnd = iFaceStart+numberOfBFaces-1;
iBFaces = iFaceStart:iFaceEnd;
%
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
iBElements = iElementStart:iElementEnd;

thBFaces = theMesh.faces(iBFaces)
```

Listing 7.27 Script to access the various arrays of struct **faces**

and the boundary elements for a scalar field can then be retrieved using Listing 7.28 as

```
phi_b = phi[iBElements]
```

Listing 7.28 Accessing the boundary elements of a scalar phi

7.1.4.3 The Node Field

The node field is a list where the indices of vertices of faces are stored. Each face is referred to by the indices of its vertices in the **points** list (Fig. 7.6). The position of

the face in the list refers to the face index. Therefore face **0** is the first entry in the list, face **1** is the second entry in the list, and so on.

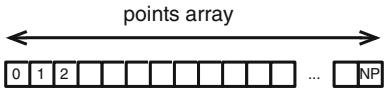


Fig. 7.6 A **points** array

7.1.5 Working with the uFVM Mesh

Looping over elements, interior faces, boundary faces, boundary elements, or even boundary patches are common operations that are performed during the discretization and solution cycles. It is worth reviewing the mechanisms that allow performing these operations readily.

7.1.5.1 Looping Over Elements

This is a simple loop to implement as the number of elements is already known and the elements are indexed from **1** to the **numberOfElements** (**0** to **numberOfElements-1** for OpenFOAM®). Also element fields are indexed in a similar fashion, so accessing them is as simple as accessing elements. Thus the loop is simply as shown in Listing 7.29.

```
for iElement=1:numberOfElements
      theElement = theMesh.elements(iElement)
      phi(iElement)  % this is field phi at centroid of element iElement
      ....
end
```

Listing 7.29 Script used to loop over elements

To access the boundary elements, the script shown in Listing 7.30 is used.

```
for boundary = 1:numberOfBoundaries
    theBoundary = theMesh.boundaries{1};
end
```

Listing 7.30 Script used to access boundary elements

7.1.5.2 Looping Over Faces

The faces array is constructed so that all interior faces run through indices **1** to **numberOfInteriorFaces** followed by the boundary faces that are also arranged according to the boundary patch to which they belong. Thus looping over the interior faces is straight forward and is written as in Listing 7.31.

```
for iFace=1:numberOfInteriorFaces
      theFace = theMesh.faces(iFace)
end
```

Listing 7.31 Script used to loop over interior faces

Looping over the boundary faces can be done as shown in Listing 7.32.

```
for iBFace= numberOfInteriorFaces+1:numberOfFaces
      theBFace = theMesh.faces(iBFace)


end
```

Listing 7.32 Script used to loop over boundary faces

If the interest is to loop over the boundary faces of a certain patch, then the loop runs from a **start** face defined in the boundary condition to **nFaces** also defined in the boundary condition. Thus a loop over the faces of boundary patch **n** would be written as depicted in Listing 7.33.

```
startFace = theMesh.boundaries(n).startFace
nFaces = theMesh.boundaries(n).nFaces
for iBFace = startFace: startFace+nFaces-1
```

Listing 7.33 Script used to loop over boundary faces of a certain patch

The subroutine **cfdGetFaceIndicesForBoundaryIndex** is used to directly get the vector **startFace: startFace+nFaces-1**.

7.1.6 Computing the Gauss Gradient

Computing the gradient of an element field in uFVM, necessitates the use of many of the above routines. Function **cfdComputeGradientGauss0**, displayed in Listing 7.34, shows the details of the Gauss Gradient implementation.

```
function phiGrad = cfdComputeGradientGauss0(phi,theMesh)
%=================================================
%  written by the CFD Group @ AUB, Fall 2014
%=================================================
%
if(nargin<2)
    theMesh = cfdGetMesh;
end
%-------------------------------------------------
% Initialize phiGrad Array
%-------------------------------------------------
phiGrad = zeros(theMesh.numberOfElements+theMesh.numberOfBFaces,3);
%-------------------------------------------------
% INTERIOR FACES contribution to gradient
%-------------------------------------------------
% get the list of interior faces indices and the list of boundary faces
indices
iFaces = 1:theMesh.numberOfInteriorFaces;
iBFaces = theMesh.numberOfInteriorFaces+1:theMesh.numberOfFaces;
% get the list of element indices and the list of boundary element indices
iElements = 1:theMesh.numberOfElements;
iBElements = theMesh.numberOfElements+1:theMesh.numberOfElements
+theMesh.numberOfBFaces;
% get the list of owners and neighbours for all interior faces
iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';
% get the surface vector of all interior faces
Sf = [theMesh.faces(ifaces).Sf]';

% get the geometric factor for all interior faces
gf = [theMesh.faces(iFaces).gf]';
% compute the linear interpolation of phi into all interior faces
phi_f = gf.*phi(iNeighbours,iComponent) + (1-gf).*phi(iOwners,iComponent);
% loop over faces and add contribution of phi face flux to the owner and
neighbour elements of the faces
for iFace=iFaces
    phiGrad(iOwners(iFace),:) = phiGrad(iOwners(iFace),:) +
                                phi_f(iFace)*Sf(iFace,:);
    phiGrad(iNeighbours(iFace),:) = phiGrad(iNeighbours(iFace),:)
                                phi_f(iFace)*Sf(iFace,:);
    end
%-------------------------------------------------
% BOUNDARY FACES contribution to gradient
%-------------------------------------------------% get the list of
elements owning a boundary face
iOwners_b = [theMesh.faces(ibFaces).iOwner]';
% get the boundary values of phi
phi_b = phi(iBElements,iComponent);
% get the surface vector of all boundary faces
Sb = [theMesh.faces(iBFaces).Sf]';
% loop over all boundary faces and add phi flux contribution to gradient
for k=1:theMesh.numberOfBFaces
    phiGrad(iOwners_b(k),:) = phiGrad(iOwners_b(k),:) + phi_b(k)*Sb(k,:);
end
```

Listing 7.34 Computing the Gauss gradient

```
%-----------------------------------------------------
% Get Average Gradient by dividing with element volume
%-----------------------------
% get volume of all elements in mesh
volumes = [theMesh.elements(iElements).volume]';
% compute the average gradient by dividing the sum of all phi fluxes for an
element by the volume of the element
for iElement =1:theMesh.numberOfElements
    phiGrad(iElement,:) = phiGrad(iElement,:)/volumes(iElement);
end
%-----------------------------------------------------
% Set boundary Gradient equal to associated element Gradient
%-----------------------------
phiGrad(iBElements,:) = phiGrad(iOwners_b,:);
```

Listing 7.34 (continued)

The loop over the boundary faces in Listing 7.34 did not take into account the patches to which the boundary faces belong, rather it just looped over all boundary faces. A version that would loop over the various patches and then over their respective boundary faces would be written as shown in Listing 7.35.

```
%-----------------------------------------------------
% BOUNDARY FACES contribution to gradient
%-----------------------------------------------------
% get the list number of boundary Patches
%
theNumberOfPatches = theMesh.numberOfBoundaries;
theEquation = cfdGetModel(theEquationName);
%
for iPatch=1:theNumberOfPatches
    %
    theBoundary = theMesh.boundaries(iPatch);
    numberOfBFaces = theBoundary.numberOfBFaces;
    %
    iFaceStart = theBoundary.startFace;
    iFaceEnd = iFaceStart+numberOfBFaces-1;
    iBFaces = iFaceStart:iFaceEnd;
    %
    iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
    iElementEnd = iElementStart+numberOfBFaces-1;
    iBElements = iElementStart:iElementEnd;

    iBOwners = [theMesh.faces(iBFaces).iOwner]?;

    phi_b = phi(iBElements,iComponent);
    Sb = [theMesh.faces(iBFaces).Sf]';
    % loop over all boundary faces and add phi flux contribution to gradient
    for k= 1: numberOfBFaces
        phiGrad(iBOwners(k),:) = phiGrad(iBOwners(k),:) + phi_b(k)*Sb(k,:);

    end
end
```

Listing 7.35 Boundary faces contribution to gradient implemented by looping over the various patches and then over their respective boundary faces

7.2 OpenFOAM®

OpenFOAM® [1] uses a finite volume cell-centered discretization of the domain and handles unstructured mesh data format based on the so called face-addressing storage. The aim of this data structure is to provide maximum flexibility in the definition of unstructured grids in order to allow for the use of a general polyhedron shape.

As shown in Fig. 7.7, a polyhedron is a three-dimensional solid consisting of a collection of plane polygons, joined at their edges. Tetrahedra (polyhedra with four triangular faces) and Hexahedra (polyhedra with six faces), for instance, are polyhedra where the faces are made by polygons of three and four edges, respectively. The possibility to describe any three-dimensional shape and use it as a finite volume element for the discretization of the equations gives multiple advantages and flexibility during mesh generation.

An efficient way to describe a mesh constructed using polyhedral elements is the face addressing. In face addressing the element shape is of no consequence to the equation discretization process, requiring the use of global addressing when forming the coefficients. (Details on the formation of coefficients from the local and global perspective will be detailed later on).

In OpenFOAM® the data for points, faces, and elements are stored in a number of lists (arrays), as shown in Fig. 7.8. The list of points contains the three dimensional spatial coordinates, defined as vectors, which correspond to the vertices of the actual mesh. The dimensional unit is the meter. Moreover each vertex has a label defined by the position in the list, and because of the use of the C++ computer language the label counting starts from zero.

The faces are defined by a list of vertex labels referring to points in which the ordering is such that each two adjacent points are connected by an edge. The face list is organized in a way that all internal faces appear first in the list followed by faces related to the first boundary, then faces related to the second boundary, and so on. Lists of boundary faces are also named *patches*. It is important to remember that internal faces belong to two cells while boundary faces belong to one cell only.

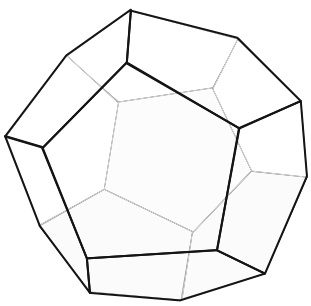


Fig. 7.7 A polyhedron element

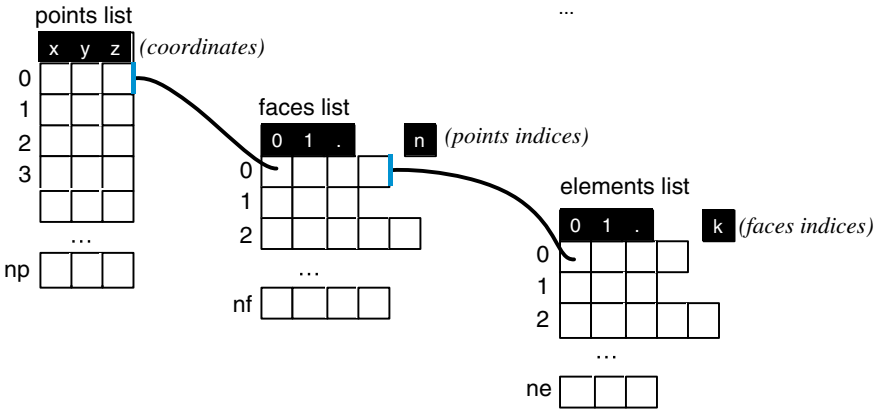


Fig. 7.8 Points, faces, and elements lists in OpenFOAM®

Finally the element or cell list is defined by a list of indices, where the position in that list is the cell index, the first index at any position is the number of faces for that element, and the face indices at a given position represent the faces for that cell.

OpenFOAM® can read computational grids from any mesh generator software capable of writing the mandatory files needed by OpenFOAM®. As explained earlier, the mesh is named **polyMesh** and has to be defined with a set of proper files. These files are placed in the directory "*constant/polyMesh*". Based on the previous description the names of the files whose contents are self-explanatory are given by

- **points**
- **faces**
- **owner**
- **neighbour**
- **boundary**

In OpenFOAM® the entire boundary of the domain is described as a list of patches in the file **boundary**. The file **boundary** is a list of all defined patches containing a proper entry named **dictionary** with each patch declared using the patch name, its type, its number of faces (**nFaces**), and the starting face (**startFace**). The general syntax used is as written in Listing 7.36.

```
PatchName
{
    type patch;
    nFaces 'number of faces';
    startFace 'starting face';
}
```

Listing 7.36 Script used to create a patch

An example of a **boundary** file with two boundary types is shown in Listing 7.37.

```
2
(
    inlet
    {
        type            patch;
        physicalType    automatic;
        nFaces          100;
        startFace       9850;
    }
    outlet
    {
        type            patch;
        physicalType    automatic;
        nFaces          100;
        startFace       9950;
    }
)
```

Listing 7.37 A **boundary** file with two boundary types

From a programming point of view it is worth giving a brief introduction of the C++ classes that handle the mesh and allow access to specific data.

The base class that handles the "low-level" structure is called **primitiveMesh**. It is a generic class that wraps the geometric information of the mesh without assuming any particular form of discretization. It is the basic class for low level information about the mesh.

Examples of some of the functions that are members of the **primitiveMesh** class are shown in Listing 7.38.

```
const labelListList &      cellCells() const
const labelListList &      pointCells() const
const cellList &           cells() const
const vectorField &        cellCentres() const
const vectorField &        faceCentres() const
const scalarField &        cellVolumes() const
const vectorField &        faceAreas() const
```

Listing 7.38 Some functions of the **primitiveMesh** class

As can be seen, this class allows obtaining specific data of the mesh (e.g., cell volume, cell centers, face centers, etc.). It is worth noting that the class itself does not recognize boundaries or domain interfaces. Boundaries and domain interfaces are defined in the **polyMesh** class, which is derived from the **primitiveMesh** class. In addition to possessing all the attributes of the derived class, the **polyMesh** class introduces the handling of boundary definition and information, as shown in Listing 7.39. Again this class does not require any particular discretization scheme.

```
const labelUList &  owner () const   //Internal face owner.
const labelUList &  neighbour () const   //Internal face neighbour.
const DimensionedField< scalar, volMesh > &    V0 () const
       Return old-time cell volumes.
const DimensionedField< scalar, volMesh > &    V00 () const
       Return old-old-time cell volumes.
 tmp< surfaceVectorField >       delta () const
       Return face deltas as surfaceVectorField
```

Listing 7.39 Information that can be obtained from the **polymesh** class

While **primitiveMesh** and **polyMesh** are the basic classes of the polyhedral mesh, the **fvMesh** class is derived from **polyMesh** and adds the particular data and functions needed for the finite-volume discretization. Addressing information as well as boundary information and specific mesh data are accessible for finite volume discretization.

OpenFOAM® decomposes the boundary mesh into patches stored in a list designated by **polyPatchList** under the class **polyBoundaryMesh**. Then, similar to interior mesh, a specialized class denoted by **fvBoundaryMesh** is derived from **polyBoundaryMesh** that inherits its functionalities and expands on it to include specific functions and data needed for finite-volume discretization. As for internal discretization, a similar hierarchy of classes composed of **primitivePatch**, **polyPatch**, and **fvPatch** is defined for boundary discretization. These classes are specific for the boundary mesh and contain the geometric information of each boundary. But it is the **fvPatch** that is used to implement the boundary conditions during the finite volume discretization. Figure 7.9 presents a schematic of the basic mesh description used in OpenFOAM®.

In a similar way the **fvMesh** is used to access all mesh functionalities. Thus, it is mainly with **fvMesh** and **fvPatch** that the discretization classes and functions interact.

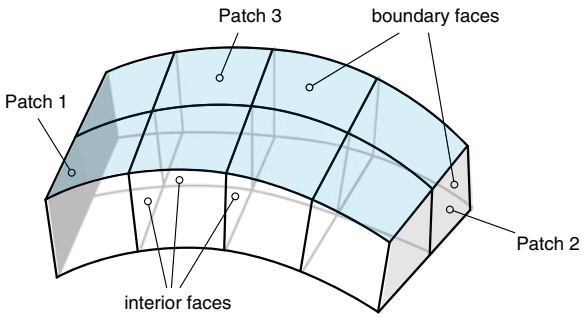


Fig. 7.9 Schematic of the basic mesh description used in OpenFOAM® [1]

Examples of codes for reading and accessing the main properties of a mesh within the OpenFOAM® framework are now presented.

To read a mesh, the **fvMesh** class that handles the finite volume mesh and discretization is needed along with a special constructor as shown in Listing 7.40.

```
#include "setRootCase.H"
#include "createTime.H"
Foam::Info
    << "Create mesh for time = "
    << runTime.timeName() << Foam::nl << Foam::endl;

Foam::fvMesh mesh
(
    Foam::IOobject
    (
        Foam::fvMesh::defaultRegion,
        runTime.timeName(),
        runTime,
        Foam::IOobject::MUST_READ
    )
);
```

Listing 7.40 Script to read a mesh in OpenFOAM®

The *include* statements in Listing 7.40 are required for initialization before reading the mesh files.

Once the **fvMesh** instantiation is constructed under the variable named *mesh*, the manipulation of the loaded mesh and collection of the necessary data for further programming can proceed, as shown in Listing 7.41.

```
    // read the element centroids
    volVectorField C = mesh.C();
    // element volumes
    volScalarField V = mesh.V();
    // surface centroids
    surfaceVectorField Cf = mesh.Cf();
```

Listing 7.41 Extracting data after reading a mesh in OpenFOAM®

To loop over the cell volumes, the dedicated class function for information on volumes should first be called (Listing 7.42). This is accomplished via

```
const DimensionedField< scalar, volMesh >&cellVolumes = mesh.V();
```

Listing 7.42 Getting information on volumes

and then a loop over the cell volumes can be performed as shown in Listing 7.43.

```
    forAll(cellVolumes, cellI)
    {
       scalar cellVolume = cellVolumes[cellI];
    }
```

Listing 7.43 Looping over all cell volumes

In OpenFOAM® the standard "for" loop is replaced by a more compact syntax using a macro definition named **forAll** (Listing 7.44) defined in the UList.H file as

```
#define forAll(list, i)  \
        for (Foam::label i=0; i<(list).size(); i++)
```

Listing 7.44 Script used to replace the standard "for" loop by the macro **forAll**

In a similar way access to other mesh information can be performed as shown in Listing 7.45.

```
    // internal access
    forAll(cellCenters.internalField(), cellI)
    {
       vector cellCenter = cellCenters[cellI];
    }
    // internal access
    forAll(faceNormals.internalField(), faceI)
    {
       vector faceNormal = faceNormals.internalField()[faceI];
    }
]
```

Listing 7.45 Script used to access mesh information

For boundary patches the access procedure is similar except that now each patch has its own class definition, and the full list of patches is defined in the **fvBoundaryMesh** class containing the **fvPatches** list. Therefore to access boundary data the following is used (Listing 7.46):

```
    const fvBoundaryMesh& boundaryMesh = mesh.boundary();
    forAll(boundaryMesh, patchI)
    {
        const fvPatch& patch = boundaryMesh[patchI];
        forAll(patch, faceI)
        {
            vector faceNormal = patch.Sf()[faceI];
            scalar faceArea = patch.magSf()[faceI];
            vector unitFaceNormal = patch.nf()()[faceI];
            vector faceCenter = patch.Cf()[faceI];
            label owner = patch.faceCells()[faceI];
        }
    }
```

Listing 7.46 Accessing data on boundary patches

7.2.1 Fields and Memory

Within the OpenFOAM® generic framework, lists, arrays, and in general containers of different types and sizes can be defined. For a given mesh and computational structure, it would be useful to define a specific class capable of combining fields, lists, and vectors directly with the mesh. A useful class that satisfies this requirement is the template class **GeometricField**<*Type*,…>. Each data defined using this class is strictly related to the mesh dimensions, both for the boundaries and interior domain (be it the number of elements, the number of interior faces or even the number of interior vertices).

In general the template class **GeometricField**<*Type*,…> stores the following data structure based on three main different characteristics of the mesh:

- **volField**<*Type*> A field defined at cell centers;
- **surfaceField**<*Type*> A field defined on cell faces;
- **pointField**<*Type*> A field defined on cell vertices.

The class also inherits the following properties:

- **Dimensions**: OpenFOAM® provides a useful feature in handling fields under the class **GeometricField**. OpenFOAM® associates with each field a dimension (meter, kg, seconds, etc.) characterizing the physical meaning of the variable. Based on that all operations carried out with **GeometricField** have to be performed with fields of the same dimension (e.g., velocity can be summed up with velocity not pressure) otherwise a runtime error transpires during execution. Moreover the dimension of any new field defined as a combination of existing fields in **GeometricField**, is automatically generated by OpenFOAM® through applying the same algebraic relation utilized to generate the new field on the dimensions of the used fields (e.g., dividing a mass field with a volume field results in a field having the dimension of density).
- **InternalField**: This is a repository of size equal to the size of the internal mesh properties (cell centers, vertices, or faces) in which internal information of a defined field is stored.
- **BoundaryField**: contains all information pertinent to a defined variable at the boundary. A list of patches is developed and each field is defined for the patches of the boundary with the name **GeometricBoundaryField**. Operations can be performed either on the full set of boundaries or on a specific patch using **fvPatchField**.
- **Mesh**: Being a class strictly related to the mesh, each **GeometricField** contains a reference to the corresponding mesh.
- **Time Values and Previous Values**: This class is required to handle the specific field during simulation. It stores information of the previous two time steps for second order accuracy in time.

In the following, examples on the use of the specific class **GeometricField** to access the main properties of a field associated with a mesh are provided.

In the first example it is required to define the two variables U and T, representing a velocity and a temperature field, respectively, at the cell center of the mesh. This will be done using the templates of the specialized class **GeometricField**, which supports scalars, vectors, and tensors data type. The script used is shown in Listing 7.47 in which one of the several constructors of the class **GeometricField** *is used* (other constructors can be found in the header file of the specific class *GeometricField.H*).

```
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("DTVol", dimensionSet(0,0,0,1,0,0,0), 300.0),
    "zeroGradient"
 );

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedVector("U",dimensionSet(0,1,-1,0,0,0,0),vector::zero),
    "zeroGradient"
);
```

Listing 7.47 Script used to define the two variables U and T in OpenFOAM®

As can be seen in the code, both fields are linked to the *mesh.* Both require specifying the following *four arguments*: (i) field name, (ii) field dimension, (iii) initialization, and (iv) boundary conditions. Specific boundary conditions have to be defined for the field and in this case a zero order extrapolation (i.e., "zeroGradient") is specified for the full set of boundaries and for both variables.

Similar constructors can be used for variables defined at faces of the mesh. For example, the script needed to define the mass flux (volume flow rate) field at cell faces is shown in Listing 7.48.

```
    surfaceScalarField mdot
    (
        IOobject
        (
            "mdot",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        linearInterpolate(U) & blockMesh.Sf()
    );
```

Listing 7.48 Script used to define the mass flux field at mesh faces

In this case the field is constructed with a scalar product between velocity (interpolated at the face) and face area vectors. This field represents the volume flow rate at control volume faces. It also represents the mass flow rate field for incompressible flows with a density value of one.

Once fields are defined, access to specific data in different parts of the mesh is possible as shown in the scripts below.

7.2.2 InternalField Data

Internal field data can be accessed (Listing 7.49) using

```
// internal access
forAll(T.internalField(), cellI)
{
   scalar cellT = T.internalField()[cellI];
}
// internal access
forAll(U.internalField(), cellI)
{
   vector cellU = U.internalField()[cellI];
}
```

Listing 7.49 Accessing internal field data

7.2.3 BoundaryField Data

Boundary field data can be accessed (Listing 7.50) using

```
const volVectorField::GeometricBoundaryField& UBoundaryList =
U.boundaryField();
// boundary access
forAll(UBoundaryList, patchI)
{
    const fvPatchField<vector>& fieldBoundary = UBoundaryList[patchI];
    forAll(fieldBoundary, faceI)
    {
        vector faceU = fieldBoundary[faceI];
    }
}
```

Listing 7.50 Accessing boundary field data

or in a more compact form via (Listing 7.51)

```
// boundary access
forAll(T.boundaryField(), patchI)
{
    forAll(T.boundaryField()[patchI], faceI)
    {
        scalar faceT = T.boundaryField()[patchI][faceI];
    }
}
```

Listing 7.51 A more compact script to access boundary field data

7.2.4 lduAddressing

OpenFOAM® uses exclusively **face addressing** in its discretization loops and coefficients storage. It also uses arbitrary polyhedral elements in its meshes. A polyhedron element can have any number of faces with a neighbor element associated with each interior face. The storage of coefficients in OpenFOAM® is based on the **face addressing scheme**. In this approach, coefficients are stored following the interior face ordering, with access to elements and their coefficients, based on the owner/neighbor indices associated with interior faces. As mentioned in the previous section, the element with the lower index is the **owner** while the **neighbor** is the element with the higher index. For boundary faces the owner is always the cell to which the face is attached while no neighbor is defined by setting the **neighbour** index to **−1**. The list of **owner** or **neighbor** indices thus define the order in which the element-to-element coefficients are assembled for the various integral operators.

The above described scheme is denoted by **lduAddressing** and is implemented in the lduMatrix class displayed in Fig. 7.10 that includes 5 arrays representing the diagonal, upper, and lower coefficients and the lower and upper indices of the face owner and neighbor, respectively. In this scheme, owners represent the lower triangular part of the matrix (lower addressing) while neighbors refer to the upper triangular part (upper addressing). It is worth mentioning that given a face for an owner element the lower and upper addressing provide respectively the column and row where the coefficient of the face flux is stored in the matrix while it is the opposite for a neighbor cell. For the domain shown on the upper left side of Fig. 7.10, the owner and neighbor of each face are displayed in the **lowerAddr()** and **upperAddr()** array, respectively. The face number is the position of the owner or neighbor in the array plus one (since numbering starts with 0). Considering internal face number 4, for example, its related information is stored in the fifth row (in C++ indexing starts from 0) of the **lower()**, **upper()**, **lowerAddr()**, and **upperAddr()** array, respectively. The stored data indicate that its owner is element number 2 **(lowerAddr()** array), its neighbor is element number 4 **(upperAddr()** array), the coefficient multiplying ϕ_4 in the algebraic equation for element number 2

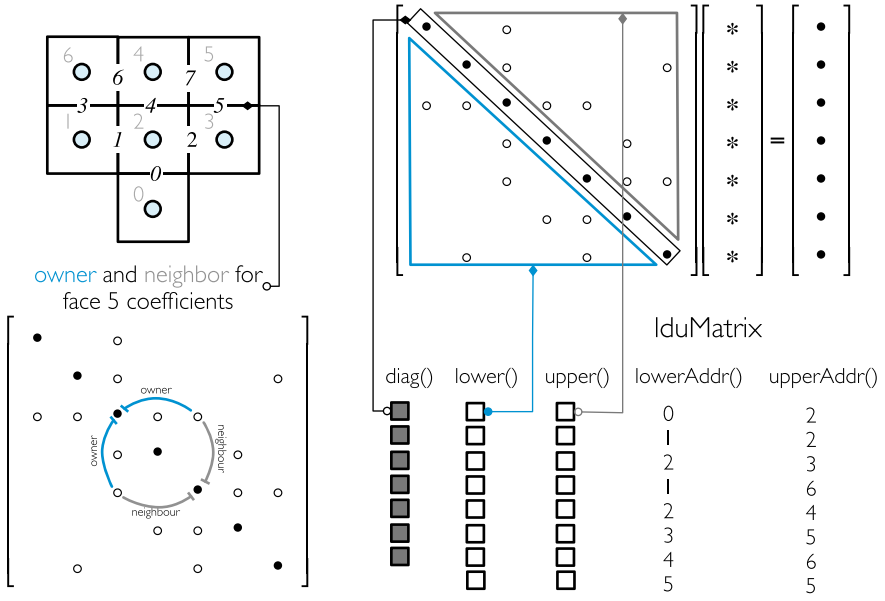


Fig. 7.10 The lduAddressing and lduMatrix

is stored in the fifth row of the array **upper()**, and the coefficient multiplying ϕ_2 in the algebraic equation for element number 4 is stored in the fifth row of the array **lower()**.

Thus the lduAdressing provides information about the addresses of the off diagonal coefficients in relation to the faces to which they are related. This means that while the computational efficiency for various operations on the matrix is high when they are mainly based on loops over all faces of the mesh, direct access to a specific row-column matrix element is difficult and inefficient. One example is the summation of the off-diagonal coefficients for each row given by

$$a_c = \sum_{n \sim nb(C)} a_n \tag{7.1}$$

In this case the use of face addressing does not allow direct looping over the off-diagonal elements of each row and performing such operation requires looping over all elements because it can only be done, as shown in Listing 7.52, following a face based approach.

```
for (label faceI=0; faceI<l.size(); faceI++)
{
    ac[l[faceI]] -= Lower[faceI];
    ac[u[faceI]] -= Upper[faceI];
}
```

Listing 7.52 Summation of the off-diagonal coefficients

In Listing 7.52 ac is the sum of the off-diagonals coefficients, *l* and *u* are the upper and lower addressing while *Lower* and *Upper* stores the corresponding coefficients. As can be noticed the summation is performed looping through all the faces and the summation of each row is not sequential, depending only on the owner-neighbor numeration of the mesh.

So in general the lduAddressing introduces a more complex handling of the matrix operations and it will be reflected also in the implementation of linear solvers but with the advantage of higher computational speed.

7.2.5 Computing the Gradient

The script used to compute the Green-Gauss gradient in OpenFOAM® is shown in Listing 7.53.

```
Foam::fv::gaussGrad<Type>::gradf
(
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf,
    const word& name
)
{
    typedef typename outerProduct<vector, Type>::type GradType;
    const fvMesh& mesh = ssf.mesh();
    tmp<GeometricField<GradType, fvPatchField, volMesh> > tgGrad
    (
        new GeometricField<GradType, fvPatchField, volMesh>
        (
            IOobject
            (
                name,
                ssf.instance(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensioned<GradType>
            (
                "0",
                ssf.dimensions()/dimLength,
                pTraits<GradType>::zero
            ),
            zeroGradientFvPatchField<GradType>::typeName
        )
    );
    GeometricField<GradType, fvPatchField, volMesh>& gGrad = tgGrad();
    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();
    const vectorField& Sf = mesh.Sf();
//
    Field<GradType>& igGrad = gGrad;
    const Field<Type>& issf = ssf;
    forAll(owner, facei)
    {
        GradType Sfssf = Sf[facei]*issf[facei];

        igGrad[owner[facei]] += Sfssf;
        igGrad[neighbour[facei]] -= Sfssf;

    }
    forAll(mesh.boundary(), patchi)
    {
        const labelUList& pFaceCells =
            mesh.boundary()[patchi].faceCells();

        const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
        const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];
        forAll(mesh.boundary()[patchi], facei)
        {
            igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
        }
    }
    igGrad /= mesh.V();
    gGrad.correctBoundaryConditions();
    return tgGrad;
}
```

Listing 7.53 Script used to compute a gradient field in OpenFOAM®

The small introduction reported above is neither intended as a replacement of OpenFOAM® user's manual [1] nor a replacement of a C++ manual. The purpose of the above presentation is to introduce the reader to the philosophy and general concepts that are useful for a quick understanding of the OpenFOAM® framework. Despite the generality of presentation, the described syntax introduced one of the main important data defined within OpenFOAM® that are necessary for writing a complete solver, which will be described in later chapters.

7.3 Mesh Conversion Tools

There are many tools capable of converting mesh files from a variety of format to the OpenFOAM® format. Some of these tools are as follows [1]:

- *ansysToFoam*: Converts an ANSYS input mesh file, exported from I-DEAS, to OpenFOAM® format.
- *cfx4ToFoam*: Converts a CFX 4 mesh to OpenFOAM® format.
- *datToFoam*: Reads in a datToFoam mesh file and outputs a points file. Used in conjunction with blockMesh.
- *fluent3DMeshToFoam*: Converts a Fluent mesh to OpenFOAM® format.
- *fluentMeshToFoam*: Converts a Fluent mesh to OpenFOAM® format including multiple region and region boundary handling.
- *foamMeshToFluent*: Writes out the OpenFOAM® mesh in Fluent mesh format.
- *foamToStarMesh*: Reads an OpenFOAM® mesh and writes a PROSTAR (v4) bnd/cel/vrt format.
- *foamToSurface*: Reads an OpenFOAM® mesh and writes the boundaries in a surface format.
- *gambitToFoam*: Converts a GAMBIT mesh to OpenFOAM® format.
- *gmshToFoam*: Reads.msh file as written by Gmsh.
- *ideasUnvToFoam*: I-Deas unv format mesh conversion.
- *kivaToFoam*: Converts a KIVA grid to OpenFOAM® format.
- *mshToFoam*: Converts.msh file generated by the Adventure system.
- *netgenNeutralToFoam*: Converts neutral file format as written by Netgen v4.4.
- *plot3dToFoam*: Plot3d mesh (ascii/formatted format) converter.
- *sammToFoam*: Converts a STAR-CD (v3) SAMM mesh to OpenFOAM® format.
- *star3ToFoam*: Converts a STAR-CD (v3) PROSTAR mesh into OpenFOAM® format.
- *star4ToFoam*: Converts a STAR-CD (v4) PROSTAR mesh into OpenFOAM® format.
- *tetgenToFoam*: Converts .ele and .node and .face files, written by tetgen.
- *writeMeshObj*: For mesh debugging: writes mesh as three separate OBJ files which can be viewed with e.g. javaview.

7.4 Closure

The chapter overviewed the realization of the FVM in a computer code by explaining several implementation attributes of the uFVM and OpenFOAM® CFD codes. The two codes were presented in terms of their data structure, memory management schemes, and case setup. The next chapter will detail the finite volume second discretization step as applied to the diffusion flux.

7.5 Exercises

Exercise 1
For the configuration shown in Fig. 7.11

(a) Write the owner and neighbor for each interior face and use it to write the owners and neighbors lists.
(b) Build the connectivity array for each element by looping over each of the interior faces using the owner-neighbor information; this is the coefficient connectivity used in uFVM.
(c) Build the coefficients ldu addressing connectivity used in OpenFOAM®.

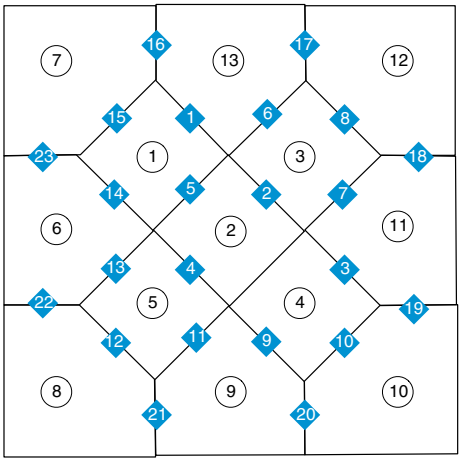


Fig. 7.11 The rectangular domain discretized with an unstructured grid used for Exercise 1

Exercise 2
Using the uFVM (MATLAB®) code, read the elbow mesh (available at the book website) and then do the following:

(a) Write a script to loop over all the elements of the domain and print out for each element its index, and the indices of its faces in the following format "element [i] → faces[k l m n…]"

(b) Declare and array phi as an element field (size = 1, Number of elements + Number of boundary faces), and initialize it using the following formula: $\phi(x, y, z) = 10xy + 5y^2$.
(c) Declare a new element array of type vector (size 3, Number of elements + Number of boundary faces) and use it to compute the gradient of phi, using the gauss theorem i.e.,

$$\nabla\phi = \frac{\sum_f \phi_f \mathbf{S}_f}{V}$$

which can be written as follows for the mesh data used:

$$\nabla\phi = \frac{\sum_{owner(f)} \overline{\phi_f}\mathbf{S}_f - \sum_{neighbour(f)} \overline{\phi_f}\mathbf{S}_f + \sum_{b=boundary(f)} \phi_b \mathbf{S}_b}{V}$$

Exercise 3
Use blockMesh to setup a uniform mesh similar to the one in Fig. 7.12 for $L = 1$. In OpenFOAM® and then in uFVM do the following:

(a) Write a program to read the mesh and loop over all boundary patches. Then for each patch print the centroid and normal vector of its faces.
(b) Modify the program to define a volumeScalarField T. Set the values for T at element centroids and at the boundaries to $10x^2y^2$, where x and y are the coordinates of the element centroids and for the case of the boundary the centroids of the boundary faces.
(c) Write a program to compute the gradient of T and compare its value with the analytical solution.

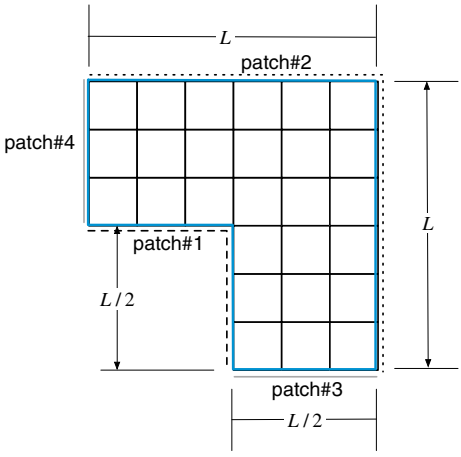


Fig. 7.12 A two dimensional domain generated with blockMesh

Exercise 4

(a) Find in OpenFOAM® using Doxygen [2], the class definition of the data types points, face, and cell.
(b) Loop over all interior faces (using forAll) and for each face write the owner, neighbor, and centroid.
(c) Loop over all elements and write for each element the value of the cubic root of its volume and its centroid.
(d) Loop over all boundary faces and write for each face the owner (parentCell) and centroid.
(e) Define a surfaceScalarField and set its value to be equal to the x component for interior faces and the y component for boundary faces.
(f) Find the member function of the surfaceScalarField (GeometricField<>) that returns the old time values.

References

1. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
2. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Part II
Discretization

Chapter 8
Spatial Discretization: The Diffusion Term

Abstract This chapter describes in detail the discretization of the diffusion term represented by the spatial Laplacian operator. It is investigated separately from the convection term, because convection and diffusion represent two distinct physical phenomena. Thus from a numerical point of view, they have to be handled differently, requiring distinct interpolation profiles with disparate considerations. The chapter begins with the discretization of the diffusion equation in the presence of a source term over a two-dimensional rectangular domain using a Cartesian grid system. The adopted interpolation profile for the variation of the dependent variable between grid points and the basic rules that should be satisfied by the coefficients of the discretized equation are discussed. The chapter proceeds with a discussion on the implementation of the Dirichlet, Von Neumann, mixed, and symmetry boundary conditions. The discretization over a non-Cartesian orthogonal grid is then introduced, followed by a detailed description of the discretization on non-orthogonal structured and unstructured grid systems. The treatment of the non-orthogonal cross-diffusion contribution, which necessitates computation of the gradient, is clarified. Then anisotropic diffusion is introduced and handled following the same methodology developed for isotropic diffusion. The under-relaxation procedure needed for highly non-linear problems is outlined. The chapter ends with computational pointers explaining the treatment of diffusion in both uFVM and OpenFOAM®.

8.1 Two-Dimensional Diffusion in a Rectangular Domain

A simple rectangular domain with a regular Cartesian grid, as shown in Fig. 8.1, is first considered. The aim is to discretize, on this domain, the steady-state diffusion equation given by

$$-\nabla \cdot \left(\Gamma^{\phi} \nabla \phi\right) = Q^{\phi} \tag{8.1}$$

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_8

where ϕ denotes a scalar variable (e.g., temperature, mass fraction of a chemical species, turbulence kinetic energy, etc.), Q^{ϕ} the generation of ϕ per unit volume within the domain, and Γ^{ϕ} the diffusion coefficient. The equation can be written more generally in term of a diffusion flux $\mathbf{J}^{\phi,D}$ as

$$\nabla \cdot \mathbf{J}^{\phi,D} = Q^{\phi} \tag{8.2}$$

where $\mathbf{J}^{\phi,D}$ is defined as

$$\mathbf{J}^{\phi,D} = -\Gamma^{\phi} \nabla \phi \tag{8.3}$$

Following the first stage discretization presented in Chap. 5, Eq. (8.1) can be formulated as

$$\sum_{f \sim nb(C)} \left(-\Gamma^{\phi} \nabla \phi\right)_f \cdot \mathbf{S}_f = Q_C^{\phi} V_C \tag{8.4}$$

The expanded form of the above equation can be written as

$$\left(-\Gamma^{\phi} \nabla \phi\right)_e \cdot \mathbf{S}_e + \left(-\Gamma^{\phi} \nabla \phi\right)_w \cdot \mathbf{S}_w + \left(-\Gamma^{\phi} \nabla \phi\right)_n \cdot \mathbf{S}_n + \left(-\Gamma^{\phi} \nabla \phi\right)_s \cdot \mathbf{S}_s = Q_C^{\phi} V_C \tag{8.5}$$

For the uniform Cartesian grid of Fig. 8.1, the surface vectors normal to the element faces are given by

$$\begin{aligned} \mathbf{S}_e &= +(\Delta y)_e \, \mathbf{i} = \|\mathbf{S}_e\| \mathbf{i} = S_e \mathbf{i} \quad \mathbf{S}_w = -(\Delta y)_w \, \mathbf{i} = -\|\mathbf{S}_w\| \mathbf{i} = -S_w \mathbf{i} \\ \mathbf{S}_n &= +(\Delta x)_n \, \mathbf{j} = \|\mathbf{S}_n\| \mathbf{j} = S_n \mathbf{j} \quad \mathbf{S}_s = -(\Delta x)_s \, \mathbf{j} = -\|\mathbf{S}_s\| \mathbf{j} = -S_s \mathbf{j} \end{aligned} \tag{8.6}$$

Thus for the east face the diffusion flux is found to be

$$\begin{aligned} J_e^{\phi,D} &= -\left(\Gamma^{\phi} \nabla \phi\right)_e \cdot \mathbf{S}_e \\ &= -\Gamma_e^{\phi} S_e \left(\frac{\partial \phi}{\partial x} \mathbf{i} + \frac{\partial \phi}{\partial y} \mathbf{j} \right)_e \cdot \mathbf{i} \\ &= -\Gamma_e^{\phi} (\Delta y)_e \left(\frac{\partial \phi}{\partial x} \right)_e \end{aligned} \tag{8.7}$$

Following the discrete conservation equation for one integration point discussed earlier, the discrete form of the diffusion flux can be written as

$$J_e^{\phi,D} = FluxT_e = FluxC_e \phi_C + FluxF_e \phi_F + FluxV_e \tag{8.8}$$

To determine the $FluxC_e$, $FluxF_e$, and $FluxV_e$ coefficients, a profile describing the variation of ϕ between the centroids of the two elements sharing the face, where the gradient has to be computed, is required. Assuming that ϕ varies linearly

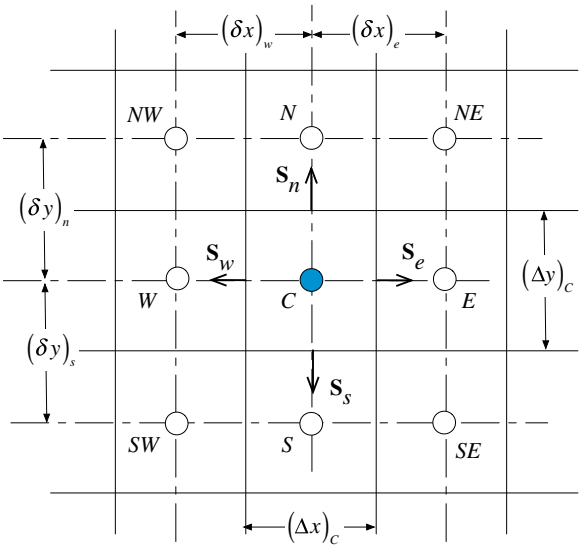


Fig. 8.1 A uniform Cartesian grid

between cell centroids (Fig. 8.2), the gradient at face e along the **i** direction may be written as

$$\left(\frac{\partial \phi}{\partial x}\right)_e = \frac{\phi_E - \phi_C}{(\delta x)_e} \tag{8.9}$$

Substituting into Eq. (8.8), the discretized form of the diffusion flux along face e is obtained as

$$\begin{aligned} FluxT_e &= -\Gamma_e^{\phi}(\Delta y)_e \frac{(\phi_E - \phi_C)}{\delta x_e} \\ &= \Gamma_e^{\phi} \frac{(\Delta y)_e}{\delta x_e}(\phi_C - \phi_E) \\ &= FluxC_e\phi_C + FluxF_e\phi_E + FluxV_e \end{aligned} \tag{8.10}$$

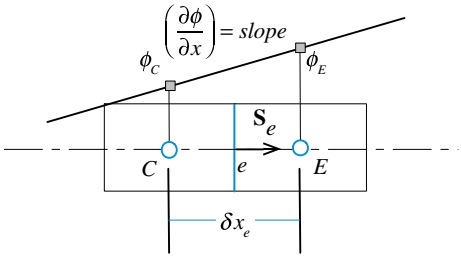


Fig. 8.2 Interpolation profile at face e and slope of gradient

Taking

$$gDiff_e = \frac{(\Delta y)_e}{\delta x_e} = \frac{\|\mathbf{S}_e\|}{\|\mathbf{d}_{CE}\|} = \frac{S_e}{d_{CE}} \tag{8.11}$$

where $\mathbf{d}_{CE}$ is the distance vector between the centroids of elements C and E, the coefficients become

$$\begin{aligned} FluxC_e &= \Gamma_e^{\phi} gDiff_e \\ FluxF_e &= -\Gamma_e^{\phi} gDiff_e \\ FluxV_e &= 0 \end{aligned} \tag{8.12}$$

A similar procedure applied to face w yields

$$\begin{aligned} FluxT_w &= -\left(\Gamma^{\phi}\nabla\phi\right)_w \cdot \mathbf{S}_w \\ &= -\Gamma_w^{\phi} S_w \left(\frac{\partial\phi}{\partial x}\mathbf{i} + \frac{\partial\phi}{\partial y}\mathbf{j}\right)_w \cdot (-\mathbf{i}) \\ &= \Gamma_w^{\phi} S_w \left(\frac{\partial\phi}{\partial x}\right)_w \\ &= \Gamma_w^{\phi} S_w \frac{(\phi_C - \phi_W)}{(\delta x)_w} \\ &= FluxC_w\phi_C + FluxF_w\phi_W + FluxV_w \end{aligned} \tag{8.13}$$

where now

$$\begin{aligned} FluxC_w &= \Gamma_w^{\phi} gDiff_w \\ FluxF_w &= -\Gamma_w^{\phi} gDiff_w \\ FluxV_w &= 0 \end{aligned} \tag{8.14}$$

and

$$gDiff_w = \frac{(\Delta y)_w}{\delta x_w} = \frac{\|\mathbf{S}_w\|}{\|\mathbf{d}_{CW}\|} = \frac{S_w}{d_{cw}}$$

Similar expressions may be written along faces n and s and are given by

$$\begin{aligned} FluxT_n &= FluxC_n\phi_C + FluxF_n\phi_N + FluxV_n \\ FluxT_s &= FluxC_s\phi_C + FluxF_s\phi_S + FluxV_s \end{aligned} \tag{8.15}$$

where

$$\begin{aligned} FluxC_n &= \Gamma_n^\phi gDiff_n \quad FluxF_n = -\Gamma_n^\phi gDiff_n \quad FluxV_n = 0 \\ FluxC_s &= \Gamma_s^\phi gDiff_s \quad FluxF_s = -\Gamma_s^\phi gDiff_s \quad FluxV_s = 0 \end{aligned} \tag{8.16}$$

and

$$\begin{aligned} gDiff_n &= \frac{\|\mathbf{S}_n\|}{\|\mathbf{d}_{CN}\|} = \frac{(\Delta x)_n}{\delta y_n} = \frac{S_n}{d_{CN}} \\ gDiff_s &= \frac{\|\mathbf{S}_s\|}{\|\mathbf{d}_{CS}\|} = \frac{(\Delta x)_s}{\delta y_s} = \frac{S_s}{d_{CS}} \end{aligned} \tag{8.17}$$

Substituting into Eq. (8.5), the algebraic form of the diffusion equation is obtained as

$$a_C\phi_C + a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S = b_C \tag{8.18}$$

where

$$\begin{aligned} a_E &= FluxF_e = -\Gamma_e^\phi gDiff_e \\ a_W &= FluxF_w = -\Gamma_w^\phi gDiff_w \\ a_N &= FluxF_n = -\Gamma_n^\phi gDiff_n \\ a_S &= FluxF_s = -\Gamma_s^\phi gDiff_s \\ a_C &= FluxC_e + FluxC_w + FluxC_n + FluxC_s \\ &= -(a_E + a_W + a_N + a_S) \\ b_C &= Q_C^\phi V_C - (FluxV_e + FluxV_w + FluxV_n + FluxV_s) \end{aligned} \tag{8.19}$$

or, more compactly, as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{8.20}$$

with

$$\begin{aligned} a_F &= FluxF_f = -\Gamma_f^\phi gDiff_f \\ a_C &= \sum_{f\sim nb(C)} FluxC_f \\ b_C &= Q_C^\phi V_C - \sum_{f\sim nb(C)} FluxV_f \end{aligned} \tag{8.21}$$

where the subscript F denotes the neighbors of element C (E, W, N, S), and the subscript f denotes the neighboring faces of element C (e, w, n, s).

8.2 Comments on the Discretized Equation

A proper discretization method should result in a discretized algebraic equation that reflects the characteristics of the original conservation equation. The properties of the discretization techniques were introduced in the previous chapter and two additional rules that the coefficients of the discretized equation have to satisfy are presented next.

8.2.1 The Zero Sum Rule

Looking back at the discretization process, the first major approximation made was the linear profile assumption for the variation of ϕ between the centroids of the elements straddling the element face. The reader might ask why to use a first order rather than a higher order profile. To answer this question, a one dimensional configuration with no source term is considered. Under these conditions the discretized equation reduces to

$$a_C\phi_C + a_E\phi_E + a_W\phi_W = 0 \tag{8.22}$$

where

$$a_E = -\Gamma_e^{\phi} gDiff_e \quad a_W = -\Gamma_w^{\phi} gDiff_w \quad a_C = -(a_E + a_W) \tag{8.23}$$

In the absence of any source or sink within this one dimensional domain, the transfer of ϕ occurs by diffusion only and is governed by Fourier's law (elliptic equation), i.e., in the direction of decreasing ϕ. As such, the value of ϕ_e or ϕ_w should lie between the values ϕ_C and ϕ_E or ϕ_C and ϕ_W, respectively, which is guaranteed by the linear profile. A second order profile (e.g., a parabolic profile) may result in a value at the face that is higher or lower than the values at the centroids of the cells straddling the face, which is unphysical. The same is true for other higher order profiles. If the discretization scheme is to guarantee physical results, then a linear profile should be used. Moreover, the adopted profile becomes less important as the size of the element decreases, since all approximations are expected to yield the same analytical solution in the limit when the size of the element approaches zero. Furthermore, in the absence of any source term, the multi-dimensional heat conduction equation reduces to,

$$-\nabla \cdot \left(\Gamma^{\phi}\nabla\phi\right) = 0 \tag{8.24}$$

This implies that ϕ and ϕ + *constant* are solutions to the conservation equation. A consistent discretization method should reflect this property through its discretized equation and the discretized equation should fulfill

$$\left.\begin{aligned} a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = 0 \\ a_C(\phi_C + constant) + \sum_{F\sim NB(C)} a_F(\phi_F + constant) = 0 \end{aligned}\right\} \Rightarrow a_C + \sum_{F\sim NB(C)} a_F = 0 \tag{8.25}$$

which is actually satisfied by the discretized equation. The above equation, which is valid in the presence or absence of a source/sink term as revealed by Eq. (8.19), may be written as

$$a_C = -\sum_{F\sim NB(C)} a_F \tag{8.26}$$

or as

$$\sum_{F\sim NB(C)} \frac{a_F}{a_C} = -1 \tag{8.27}$$

Thus ϕ_C can be viewed as the weighted sum of its neighbors, and in the absence of any source term it should always be bounded by these neighboring ϕ_F values. When a source term is present, i.e., when $S_C^\phi \neq 0$, ϕ_C does not need to be bounded in this manner, and can over/undershoot the neighboring values, but this is perfectly physical. The extent of over/under shoot is determined by the magnitude of S_C^ϕ with respect to the size of the coefficients at the neighboring nodes (a_F).

8.2.2 The Opposite Signs Rule

The above derivations demonstrated that the coefficients a_C and a_F are of opposite signs. This is of physical significance implying that as the value of ϕ_F is increased/decreased, the value of ϕ_C is expected to increase/decrease. This is basically related to boundedness suggesting that a sufficient condition for this property to be satisfied is for the neighboring and main coefficients to be of opposite signs. If not, then the boundedness property may not be enforced.

8.3 Boundary Conditions

It is well known that the analytical solution to any ordinary or partial differential equation is obtained up to some constants that are fixed by the applicable boundary conditions to the situation being studied. Therefore, using different boundary

conditions will result in different solutions even though the general equation remains the same. Numerical solutions follow the same constrain, necessitating correct and accurate implementation of boundary conditions as any slight change in these conditions introduced by the numerical approximation leads to a wrong solution of the problem under consideration.

Boundary conditions will be discussed as deemed relevant to the terms being discretized. For conduction/diffusion problems Dirichlet, Neumann, mixed, and symmetry boundary condition types are encountered, which are detailed next.

Boundary conditions are applied on boundary elements, which have one or more faces on the boundary. Discrete values of ϕ are stored both at centroids of boundary cells and at centroids of boundary faces.

Let C denotes the centroid of the boundary element shown in Fig. 8.3 with one boundary face of centroid b and of surface vector $\mathbf{S}_b$ pointing outward. As before, the discretization process over cell C yields

$$\sum_{f\sim nb(C)} \left(\mathbf{J}^{\phi,D}\cdot\mathbf{S}\right)_f = Q_C^{\phi} V_C \tag{8.28}$$

The fluxes on the interior faces are discretized as before, while the boundary flux is discretized with the aim of constructing a linearization with respect to ϕ_C, thus

$$\begin{aligned}\mathbf{J}_b^{\phi,D}\cdot\mathbf{S}_b &= FluxT_b \\ &= -\Gamma_b^{\phi}(\nabla\phi)_b\cdot\mathbf{S}_b \\ &= FluxC_b\phi_C + FluxV_b\end{aligned} \tag{8.29}$$

The specification of boundary conditions involves either specifying the unknown boundary value ϕ_b, or alternatively, the boundary flux $\mathbf{J}_b^{\phi,D}$. Using Eq. (8.18), the discretized equation at a boundary element for the different boundary condition types of diffusion problems are derived next.

8.3.1 Dirichlet Boundary Condition

A Dirichlet boundary condition is a type of boundary condition that specifies the value of ϕ at the boundary, i.e.,

$$\phi_b = \phi_{specified} \tag{8.30}$$

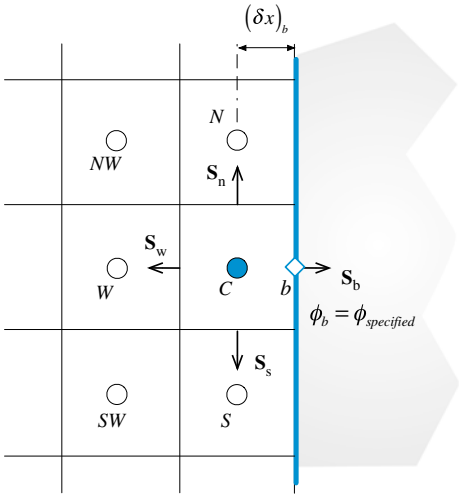


Fig. 8.3 Value specified boundary condition

For this case

$$\begin{aligned} FluxT_b &= -\Gamma_b^{\phi}(\nabla\phi)_b \cdot \mathbf{S}_b \\ &= -\Gamma_b^{\phi}\frac{\|\mathbf{S}_b\|}{\|\mathbf{d}_{Cb}\|}(\phi_b - \phi_C) \\ &= FluxC_b\phi_C + FluxV_b \end{aligned} \tag{8.31}$$

yielding

$$\begin{aligned} FluxC_b &= \Gamma_b^{\phi} gDiff_b = a_b \\ FluxV_b &= -\Gamma_b^{\phi} gDiff_b \phi_b = -a_b\phi_b \end{aligned} \tag{8.32}$$

with

$$gDiff_b = \frac{S_b}{d_{Cb}} \tag{8.33}$$

Thus for element C shown in Fig. 8.3 the a_E coefficient is zero reducing the discretized equation to

$$a_C\phi_C + a_W\phi_W + a_N\phi_N + a_S\phi_S = b_C \tag{8.34}$$

where

$$\begin{aligned} a_E &= 0 \\ a_W &= FluxF_w = -\Gamma_w^{\phi} gDiff_w \\ a_N &= FluxF_n = -\Gamma_n^{\phi} gDiff_n \\ a_S &= FluxF_s = -\Gamma_s^{\phi} gDiff_s \\ a_C &= FluxC_b + \sum_{f\sim nb(C)} FluxC_f = FluxC_b + (FluxC_w + FluxC_n + FluxC_s) \end{aligned} \tag{8.35}$$

and

$$b_C = Q_C^{\phi} V_C - \left(FluxV_b + \sum_{f \sim nb(C)} FluxV_f \right) \tag{8.36}$$

The following important observations can be made about the discretized boundary equation:

1. The coefficient a_b is larger than other neighbor coefficients because b is closer to C and consequently has a more important effect on ϕ_C.
2. The coefficient a_C is still the sum of all neighboring coefficients including a_b. This means that for the boundary element $\sum_{F \sim NB(C)} |a_F|/|a_C| < 1$ giving the second necessary condition to satisfy the Scarborough criterion, thus guaranteeing, at any one iteration, the convergence of the linear system of equations via an iterative solution method.
3. The $a_b\phi_b$ product (=$FluxV_b$) is now on the right hand side of the equation, i.e., part of b_C, because it contains no unknowns.

8.3.2 Von Neumann Boundary Condition

If the flux (or normal gradient to the face) of ϕ is specified at the boundary (Fig. 8.4), then the boundary condition is denoted by a Neumann boundary condition. In this case the specified flux is given by

$$-\left(\Gamma^{\phi} \nabla \phi\right)_b \cdot \mathbf{i} = q_b \tag{8.37}$$

which is, in effect, the flux $\mathbf{J}_b^{\phi,D}$, since

$$\begin{aligned} \mathbf{J}_b^{\phi,D} \cdot \mathbf{S}_b &= -\left(\Gamma^{\phi} \nabla \phi\right)_b \cdot \|\mathbf{S}_b\| \mathbf{i} = q_b \|\mathbf{S}_b\| \\ &= FluxC_b \phi_C + FluxV_b \end{aligned} \tag{8.38}$$

where now

$$\begin{aligned} FluxC_b &= 0 \\ FluxV_b &= q_b S_b \\ &= q_b (\Delta y)_C \end{aligned} \tag{8.39}$$

Here the flux components are assumed to be positive when they act in the same direction as the coordinate system used.

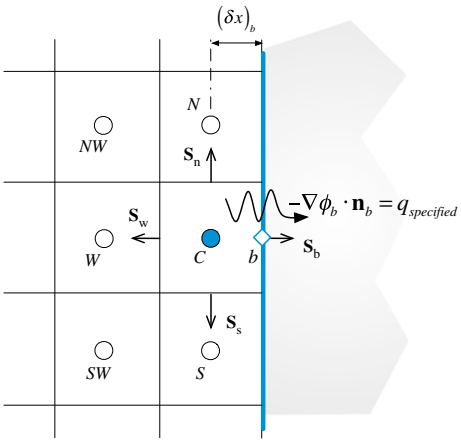


Fig. 8.4 Flux specified boundary condition

Thus q_b may directly be included in Eq. (8.18) to yield the following discrete equation for the boundary cell C:

$$a_C\phi_C + a_W\phi_W + a_N\phi_N + a_S\phi_S = b_C \tag{8.40}$$

where

$$\begin{aligned}
a_E &= 0 \\
a_W &= FluxF_w = -\Gamma_w^\phi gDiff_w \\
a_N &= FluxF_n = -\Gamma_n^\phi gDiff_n \\
a_S &= FluxF_s = -\Gamma_s^\phi gDiff_s \\
a_C &= \sum_{f\sim nb(C)} FluxC_f = -(a_W + a_N + a_S) \\
b_C &= Q_C^\phi V_C - \left(FluxV_b + \sum_{f\sim nb(C)} FluxV_f \right)
\end{aligned} \tag{8.41}$$

The following important points can be made about the above discretized equation:

1. A Von Neumann boundary condition does not result in a dominant a_C coefficient.
2. If both q_b and S_C^ϕ are zero, then ϕ_C will be bounded by its neighbors. Otherwise, ϕ_C can exceed (or fall below) the neighbor values of ϕ, which is admissible. If ϕ is temperature, for example, then q_b represents the heat flux applied at the boundary. Therefore if heat is added at the boundary, then the temperature in the region close to the boundary is expected to be higher than that in the interior.

3. Once ϕ_C is computed, the boundary value ϕ_b may be computed using

$$\phi_b = \frac{\Gamma_b^{\phi} gDiff_b \phi_C - q_b}{\Gamma_b^{\phi} gDiff_b} \tag{8.42}$$

4. Finally the Von Neumann condition can be considered as a natural boundary condition for the Finite Volume method since, for the case where the specified flux is zero, nothing needs to be done in terms of discretization for the face, while a specified value of zero (Dirichlet condition) will still require the discretization to be carried out.

8.3.3 Mixed Boundary Condition

The mixed boundary condition, schematically depicted in Fig. 8.5, refers to the situation where information at the boundary is given via a convection transfer coefficient (h_∞) and a surrounding value for $\phi(\phi_\infty)$ as

$$\mathbf{J}_b^{\phi,D} \cdot \mathbf{S}_b = -\left(\Gamma^{\phi} \nabla \phi\right)_b \cdot \mathbf{i} S_b = -h_\infty (\phi_\infty - \phi_b)(\Delta y)_C \tag{8.43}$$

which can be rewritten as

$$-\Gamma_b^{\phi} S_b \left(\frac{\phi_b - \phi_C}{\delta x_b}\right) = -h_\infty (\phi_\infty - \phi_b) S_b \tag{8.44}$$

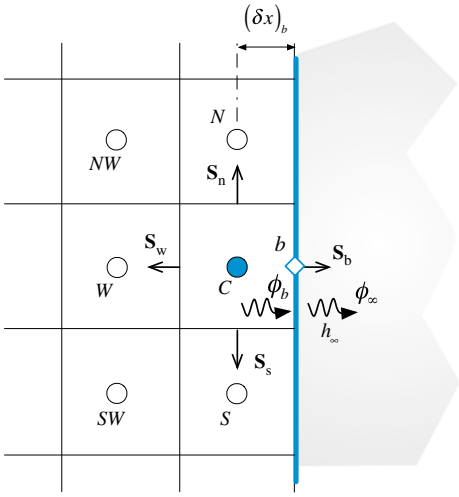


Fig. 8.5 Mixed type boundary condition

from which an equation for ϕ_b is obtained as

$$\phi_b = \frac{h_\infty \phi_\infty + \left(\Gamma_b^\phi / \delta x_b\right)\phi_C}{h_\infty + \left(\Gamma_b^\phi / \delta x_b\right)} \tag{8.45}$$

Substituting ϕ_b back in Eq. (8.43), the flux equation is transformed to

$$\begin{aligned} \mathbf{J}_b^{\phi,D} \cdot \mathbf{S}_b &= -\underbrace{\left[\frac{h_\infty\left(\Gamma_b^\phi/\delta x_b\right)}{h_\infty + \left(\Gamma_b^\phi/\delta x_b\right)} S_b\right]}_{R_{eq}} (\phi_\infty - \phi_C) \\ &= FluxC_b \phi_C + FluxV_b \end{aligned} \tag{8.46}$$

where now

$$\begin{aligned} FluxC_b &= R_{eq} \\ FluxV_b &= -R_{eq}\phi_\infty \end{aligned} \tag{8.47}$$

Using the flux term given by Eq. (8.47) in the discretized equation of the boundary element C, the modified equation is found to be

$$a_C \phi_C + a_W \phi_W + a_N \phi_N + a_S \phi_S = b_C \tag{8.48}$$

where

$$\begin{aligned} a_E &= 0 \\ a_W &= FluxF_w = -\Gamma_w^\phi gDiff_w \\ a_N &= FluxF_n = -\Gamma_n^\phi gDiff_n \\ a_S &= FluxF_s = -\Gamma_s^\phi gDiff_s \\ a_C &= FluxC_b + \sum_{f\sim nb(C)} FluxC_f = (FluxC_b + FluxC_w + FluxC_n + FluxC_s) \\ b_C &= Q_C^\phi V_C - \left(FluxV_b + \sum_{f=nb(C)} FluxV_f\right) \end{aligned} \tag{8.49}$$

8.3.4 Symmetry Boundary Condition

Along a symmetry boundary the normal flux to the boundary of a scalar variable ϕ is zero. Therefore a symmetry boundary condition is equivalent to a Neumann boundary condition with the value of the flux set to zero (i.e., $FluxC_b = FluxV_b = 0$).

Thus the modified equation along a symmetry boundary condition can be deduced from Eqs. (8.40) and (8.41) by setting q_b to zero and is given by

$$a_C\phi_C + a_W\phi_W + a_N\phi_N + a_S\phi_S = b_C \tag{8.50}$$

where

$$\begin{aligned} a_E &= 0 \\ a_W &= FluxF_w = -\Gamma_w^\phi gDiff_w \\ a_N &= FluxF_n = -\Gamma_n^\phi gDiff_n \\ a_S &= FluxF_s = -\Gamma_s^\phi gDiff_s \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = -(a_W + a_N + a_S) \\ b_C &= Q_C^\phi V_C - \sum_{f\sim nb(C)} FluxV_f \end{aligned} \tag{8.51}$$

8.4 The Interface Diffusivity

In the above discretized equation, the diffusion coefficients Γ_e^ϕ, Γ_w^ϕ, Γ_n^ϕ, and Γ_s^ϕ have been used to represent the value of Γ^ϕ at the e, w, n and s faces of the element, respectively. When the diffusion coefficient Γ^ϕ varies with position, its value will be known at the cell centroids E, W, … and so on. Then a prescription for evaluating the interface value is needed in terms of values at these grid points. The following discussion is, of course, not relevant to situations of uniform diffusion coefficient.

The discussion may become clearer if the energy equation is considered, in which case the diffusion coefficient represents the conductivity of the material used and ϕ denotes the temperature. Non-uniform conductivity occurs in non-homogeneous materials and/or when the thermal conductivity is temperature dependent. In the treatment of the general differential equation for ϕ, the diffusion coefficient Γ^ϕ will be handled in the same way. Significant variations of Γ^ϕ are frequently encountered for example, in turbulent flow, where Γ^ϕ may stand for the turbulent viscosity or the turbulent conductivity. Thus, a proper formulation for non-uniform Γ^ϕ is highly desirable.

A simple approach for calculating the interface conductivity is the linear profile assumption for the variation of Γ^ϕ, between point C and any of its neighbors. Thus, along the east face the value of Γ^ϕ, is obtained as

$$\Gamma_e^\phi = (1 - g_e)\Gamma_C^\phi + g_e\Gamma_E^\phi \tag{8.52}$$

where the interpolation factor g_e is a ratio in terms of distances between centroids given by

$$g_e = \frac{d_{Ce}}{d_{Ce} + d_{eE}} \tag{8.53}$$

Hence for a Cartesian grid if the interface is midway between the grid points, g_e would be 0.5, and Γ_e^ϕ would be the arithmetic mean of Γ_C^ϕ and Γ_E^ϕ. Similar coefficients can be defined for other faces as

$$\begin{aligned} g_w &= \frac{d_{Cw}}{d_{Cw} + d_{wW}} \\ g_n &= \frac{d_{Cn}}{d_{Cn} + d_{nN}} \\ g_s &= \frac{d_{Cs}}{d_{Cs} + d_{sS}} \end{aligned} \tag{8.54}$$

Therefore it is sufficient to calculate the coefficients of each surface of an element only once. Moreover, the use of distances instead of volumes will lead to the same interpolation factors as the grid here is Cartesian.

This basic approach leads to rather incorrect implications in some cases and cannot accurately handles, for example, abrupt changes of conductivity that may occur in composite materials. Fortunately, a much better alternative of comparable simplicity is available. In developing this alternative, it is recognized that the local value of conductivity at an interface is not of primary concern. Rather, the main objective is to obtain a good representation of the diffusion flux $\mathbf{J}^{\phi,D}$ at the interface [1].

For the one dimensional problem shown in Fig. 8.6, it is assumed that the element C is composed of a material having a thermal conductivity Γ_C^ϕ, while element E is made of a material of thermal conductivity Γ_E^ϕ. For the non-homogeneous slab between points C and E, a steady one-dimensional analysis (without sources) results in (the flux on either side of the interface e is supposed to be the same)

$$\mathbf{J}_e^{\phi,D} \cdot \mathbf{S}_e = \frac{\phi_C - \phi_e}{\frac{(\delta x)_{Ce}}{\Gamma_C^\phi}} = \frac{\phi_e - \phi_E}{\frac{(\delta x)_{eE}}{\Gamma_E^\phi}} = \frac{\phi_C - \phi_E}{\frac{(\delta x)_{Ce}}{\Gamma_C^\phi} + \frac{(\delta x)_{eE}}{\Gamma_E^\phi}} = \frac{\phi_C - \phi_E}{\frac{(\delta x)_{CE}}{\Gamma_e^\phi}} \tag{8.55}$$

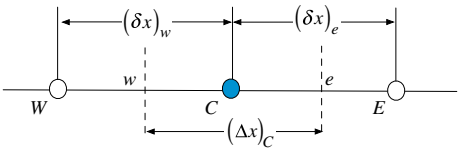


Fig. 8.6 Interpolation of properties at element faces

Hence the effective conductivity for the slab is found to be

$$\frac{(\delta x)_{CE}}{\Gamma_e^\phi} = \frac{(\delta x)_{Ce}}{\Gamma_C^\phi} + \frac{(\delta x)_{eE}}{\Gamma_E^\phi} \Rightarrow \frac{1}{\Gamma_e^\phi} = \left(\frac{1 - g_e}{\Gamma_E^\phi} + \frac{g_e}{\Gamma_C^\phi} \right) \tag{8.56}$$

When the interface is halfway between C and E ($g_e = 0.5$), Eq. (8.56) reduces to

$$\Gamma_e^\phi = \frac{2\Gamma_C^\phi \Gamma_E^\phi}{\Gamma_C^\phi + \Gamma_E^\phi} \tag{8.57}$$

which is the ***harmonic mean*** of Γ_C^ϕ and Γ_E^ϕ, rather than the ***arithmetic*** mean.

It is important to note that the harmonic mean interpolation for discontinuous diffusion coefficients is exact only for one-dimensional diffusion. Nevertheless, its application for multi-dimensional situation has an important advantage. With this type of interpolation, nothing special need to be done when treating conjugate interfaces. Solid and fluid cells are simply treated as part of the same domain with different diffusion coefficients stored at the cell centroids. By calculating the face diffusivity as the harmonic-mean of the values at the centroids sharing the face, the diffusion flux at the conjugate interface is correctly computed.

Example 1

The heat conduction in the two-dimensional rectangular domain composed of two materials shown in Fig. 8.7 *is governed by the following differential equation*:

$$\nabla \cdot (k \nabla T) = 0$$

where T represents temperature. For the thermal conductivities (k) and boundary conditions displayed in the figure:

(a) *Derive the algebraic equations for all the elements shown in the figure.*
(b) *Using the Gauss-Seidel iterative method discussed in Chap. 4, solve the system of equations obtained and compute the cell values of T.*
(c) *Compute the values of T at the bottom, right, and top boundaries.*
(d) *Compute the net heat transfer through the top, bottom, and left boundaries and check that energy conservation is satisfied.*

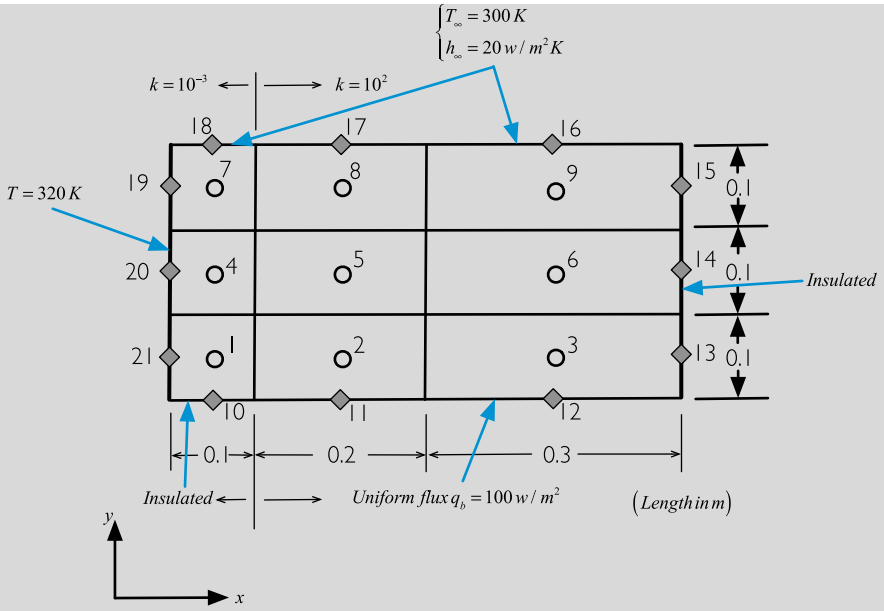


Fig. 8.7 Conduction heat transfer in a two dimensional rectangular domain

Solution

The first step is to determine the geometric quantities that are needed in the solution. The coordinates of the various nodes are found to be

$$x_{21} = x_{20} = x_{19} = 0 \qquad y_{10} = y_{11} = y_{12} = 0$$
$$x_{10} = x_1 = x_4 = x_7 = x_{18} = 0.05 \qquad y_{21} = y_1 = y_2 = y_3 = y_{13} = 0.05$$
$$x_{11} = x_2 = x_5 = x_8 = x_{17} = 0.2 \qquad y_{20} = y_4 = y_5 = y_6 = y_{14} = 0.15$$
$$x_{12} = x_3 = x_6 = x_9 = x_{16} = 0.45 \qquad y_{19} = y_7 = y_8 = y_9 = y_{15} = 0.25$$
$$x_{13} = x_{14} = x_{15} = 0.6 \qquad y_{18} = y_{17} = y_{16} = 0.3$$

The distance between nodes are computed as

$$\delta x_{21-1} = \delta x_{20-4} = \delta x_{19-7} = 0.05 \qquad \delta y_{10-1} = \delta y_{11-2} = \delta y_{12-3} = 0.05$$
$$\delta x_{1-2} = \delta x_{4-5} = \delta x_{7-8} = 0.15 \qquad \delta y_{1-4} = \delta y_{2-5} = \delta y_{3-6} = 0.1$$
$$\delta x_{2-3} = \delta x_{5-6} = \delta x_{8-9} = 0.25 \qquad \delta y_{4-7} = \delta y_{5-8} = \delta y_{6-9} = 0.1$$
$$\delta x_{3-13} = \delta x_{6-14} = \delta x_{9-15} = 0.15 \qquad \delta y_{7-18} = \delta y_{8-17} = \delta y_{9-16} = 0.05$$

The size of the elements are given by

$$\begin{array}{ll} \Delta x_1 = \Delta x_4 = \Delta x_7 = 0.1 & \Delta y_1 = \Delta y_2 = \Delta y_3 = 0.1 \\ \Delta x_2 = \Delta x_5 = \Delta x_8 = 0.2 & \Delta y_4 = \Delta y_5 = \Delta y_6 = 0.1 \\ \Delta x_3 = \Delta x_6 = \Delta x_9 = 0.3 & \Delta y_7 = \Delta y_8 = \Delta y_9 = 0.1 \end{array}$$

The computed volumes of the various cells are obtained as

$$V_1 = V_4 = V_7 = 0.01 \quad V_2 = V_5 = V_8 = 0.02 \quad V_3 = V_6 = V_9 = 0.03$$

The interpolation factors needed to find values at element faces are calculated as

$$\begin{aligned} (g_e)_1 = (g_e)_4 = (g_e)_7 &= \frac{V_1}{V_1 + V_2} = \frac{0.01}{0.01 + 0.02} = 0.333 \\ (g_e)_2 = (g_e)_5 = (g_e)_8 &= \frac{V_2}{V_2 + V_3} = \frac{0.02}{0.02 + 0.03} = 0.4 \\ (g_n)_1 = (g_n)_2 = (g_n)_3 &= \frac{V_1}{V_1 + V_4} = \frac{0.01}{0.01 + 0.01} = 0.5 \\ (g_n)_4 = (g_n)_5 = (g_n)_6 &= \frac{V_{10}}{V_{10} + V_{15}} = \frac{0.01}{0.01 + 0.01} = 0.5 \end{aligned}$$

The thermal conductivity values over the domain are given by

$$k_1 = k_4 = k_7 = k_{10} = k_{21} = k_{20} = k_{19} = k_{18} = 10^{-3}$$
$$k_2 = k_3 = k_5 = k_6 = k_8 = k_9 = k_{11} = k_{12} = k_{13} = k_{14} = k_{15} = k_{16} = k_{17} = 10^2$$

while values at the element faces are found to be

$$k_{1-2} = \frac{k_1 k_2}{\left[1 - (g_e)_1\right] k_1 + (g_e)_1 k_2} = \frac{10^{-3} \times 10^2}{(1 - 0.333) \times 10^{-3} + 0.333 \times 10^{-2}} \approx 3 \times 10^{-3}$$
$$k_{1-2} = k_{4-5} = k_{7-8} = 3 \times 10^{-3} \quad k_{1-4} = k_{4-7} = 10^{-3} \quad k_{2-5} = k_{3-6} = k_{5-8} = k_{6-9} = 10^2$$

Using the above values, the discretized algebraic equations for all elements are derived next.

Element #1

The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_1}{\delta x_{1-2}} = \frac{0.1}{0.15} = 0.667 \quad gDiff_w = \frac{\Delta y_1}{\delta x_{21-1}} = \frac{0.1}{0.05} = 2$$
$$gDiff_n = \frac{\Delta x_1}{\delta y_{1-4}} = \frac{0.1}{0.1} = 1$$

The interface conductivities are

$$k_e = k_{1-2} = 3 \times 10^{-3} \quad k_n = k_{1-4} = 10^{-3} \quad k_w = k_{21} = 10^{-3}$$

The general form of the equation is written as

$$a_C T_1 + a_E T_2 + a_N T_4 = b_C$$

where

$$a_E = FluxF_e = -k_e gDiff_e = -3 \times 10^{-3} \times 0.667 = -0.002$$
$$a_N = FluxF_n = -k_n gDiff_n = -10^{-3} \times 1 = -0.001$$

The west and south coefficients do not appear in the equation as their influence is integrated through the boundary conditions as

$$FluxC_w = k_w gDiff_w = 10^{-3} \times 2 = 0.002$$
$$FluxV_s = 0$$
$$FluxV_w = -k_w gDiff_w T_{21} = -10^{-3} \times 2 \times 320 = -0.64$$

The main coefficient and source term can now be calculated and are given by

$$a_C = FLuxC_e + FLuxC_n + FLuxC_w = 0.002 + 0.001 + 0.002 = 0.005$$
$$b_C = -FluxV_S - FluxV_w = 0 + 0.64 = 0.64$$

Substituting, the discretized algebraic equation is obtained as

$$0.005T_1 - 0.002T_2 - 0.001T_4 = 0.64$$

Element #2
The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_2}{\delta x_{2-3}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_w = \frac{\Delta y_2}{\delta x_{1-2}} = \frac{0.1}{0.15} = 0.667$$
$$gDiff_n = \frac{\Delta x_2}{\delta y_{2-5}} = \frac{0.2}{0.1} = 2$$

The interface conductivities are

$$k_e = k_{2-3} = 10^2 \quad k_n = k_{2-5} = 10^2 \quad k_w = k_{1-2} = 3 \times 10^{-3}$$

The general form of the equation is written as

$$a_C T_2 + a_E T_3 + a_W T_1 + a_N T_5 = b_C$$

where

$$a_E = FluxF_e = -k_e gDiff_e = -10^2 \times 0.4 = -40$$
$$a_W = FluxF_w = -k_w gDiff_w = -3 \times 10^{-3} \times 0.667 = -0.002$$
$$a_N = FluxF_n = -k_n gDiff_n = -10^2 \times 2 = -200$$

The south coefficient does not appear in the equation as its influence is integrated through the boundary conditions as

$$FluxV_s = q_b \mathbf{j} \cdot \mathbf{S}_1 = q_b \mathbf{j} \cdot (-\Delta x_2 \mathbf{j}) = -100 * 0.2 = -20$$

The main coefficient and source term can now be calculated and are given by

$$a_C = FLuxC_e + FLuxC_n + FLuxC_w = 40 + 0.002 + 200 = 240.002$$
$$b_C = -FluxV_s = 20$$

Substituting, the discretized algebraic equation is obtained as

$$240.002T_2 - 40T_3 - 0.002T_1 - 200T_5 = 20$$

Element #3
The needed diffusion terms are calculated as

$$gDiff_w = \frac{\Delta y_3}{\delta x_{2-3}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_n = \frac{\Delta x_3}{\delta y_{3-6}} = \frac{0.3}{0.1} = 3$$

The interface conductivities are

$$k_w = k_{2-3} = 10^2 \quad k_n = k_{3-6} = 10^2$$

The general form of the equation is written as

$$a_C T_3 + a_W T_2 + a_N T_6 = b_C$$

where

$$a_W = FluxF_w = -k_w gDiff_w = -10^2 \times 0.4 = -40$$
$$a_N = FluxF_n = -k_n gDiff_n = -10^2 \times 3 = -300$$

The east and south coefficients do not appear in the equation as their influence is integrated through the boundary conditions as

$$FluxV_e = 0$$
$$FluxV_s = q_b\mathbf{j} \cdot \mathbf{S}_1 = q_b\mathbf{j} \cdot (-\Delta x_3\mathbf{j}) = -100 * 0.3 = -30$$

The main coefficient and source term can now be calculated and are given by

$$a_C = FLuxC_w + FLuxC_n = 40 + 300 = 340$$
$$b_C = -FluxV_S - FluxV_e = 30 + 0 = 30$$

Substituting, the discretized algebraic equation is obtained as

$$340T_3 - 40T_2 - 300T_6 = 30$$

Element #4
The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_4}{\delta x_{4-5}} = \frac{0.1}{0.15} = 0.667 \quad gDiff_w = \frac{\Delta y_4}{\delta x_{20-4}} = \frac{0.1}{0.05} = 2$$
$$gDiff_n = \frac{\Delta x_4}{\delta y_{4-7}} = \frac{0.1}{0.1} = 1 \qquad gDiff_s = \frac{\Delta x_4}{\delta y_{1-4}} = \frac{0.1}{0.1} = 1$$

The interface conductivities are

$$k_e = k_{4-5} = 3 \times 10^{-3} \quad k_w = k_{20} = 10^{-3} \quad k_n = k_{4-7} = 10^{-3} \quad k_s = k_{1-4} = 10^{-3}$$

The general form of the equation is written as

$$a_C T_4 + a_E T_5 + a_N T_7 + a_S T_1 = b_C$$

where

$$a_E = FluxF_e = -k_e gDiff_e = -3 \times 10^{-3} \times 0.667 = -0.002$$
$$a_N = FluxF_n = -k_n gDiff_n = -10^{-3} \times 1 = -0.001$$
$$a_S = FluxF_s = -k_s gDiff_s = -10^{-3} \times 1 = -0.001$$

The west coefficient does not appear in the equation as its influence is integrated through the boundary condition as

$$FluxC_w = k_w gDiff_w = 10^{-3} \times 2 = 0.002$$
$$FluxV_w = -k_w gDiff_w T_{20} = -10^{-3} \times 2 \times 320 = -0.64$$

The main coefficient and source term can now be calculated and are given by

$$\begin{aligned} a_C &= FLuxC_e + FLuxC_w + FLuxC_n + FLuxC_s \\ &= 0.002 + 0.002 + 0.001 + 0.001 = 0.006 \\ b_C &= -FluxV_w = 0.64 \end{aligned}$$

Substituting, the discretized algebraic equation is obtained as

$$0.006T_4 - 0.002T_5 - 0.001T_7 - 0.001T_1 = 0.64$$

Element #5
The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_5}{\delta x_{5-6}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_w = \frac{\Delta y_5}{\delta x_{4-5}} = \frac{0.1}{0.15} = 0.667$$
$$gDiff_n = \frac{\Delta x_5}{\delta y_{5-8}} = \frac{0.2}{0.1} = 2 \qquad gDiff_s = \frac{\Delta x_5}{\delta y_{2-5}} = \frac{0.2}{0.1} = 2$$

The interface conductivities are

$$k_e = k_{5-6} = 10^2 \quad k_w = k_{4-5} = 3 \times 10^{-3} \quad k_n = k_{5-8} = 10^2 \quad k_s = k_{2-5} = 10^2$$

The general form of the equation is written as

$$a_C T_5 + a_E T_6 + a_W T_4 + a_N T_8 + a_S T_2 = b_C$$

where

$$a_E = FluxF_e = -k_e gDiff_e = -10^2 \times 0.4 = -40$$
$$a_W = FluxF_w = -k_w gDiff_w = -3 \times 10^{-3} \times 0.667 = -0.002$$
$$a_N = FluxF_n = -k_n gDiff_n = -10^2 \times 2 = -200$$
$$a_S = FluxF_s = -k_s gDiff_s = -10^2 \times 2 = -200$$

All coefficients appear in the equation as this is an internal element. The main coefficient and source term are calculated as

$$\begin{aligned} a_C &= FLuxC_e + FLuxC_w + FLuxC_n + FLuxC_s \\ &= 40 + 0.002 + 200 + 200 = 440.002 \\ b_C &= 0 \end{aligned}$$

Substituting, the discretized algebraic equation is obtained as

$$440.002T_5 - 40T_6 - 0.002T_4 - 200T_8 - 200T_2 = 0$$

Element #6

The needed diffusion terms are calculated as

$$gDiff_w = \frac{\Delta y_6}{\delta x_{5-6}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_n = \frac{\Delta x_6}{\delta y_{6-9}} = \frac{0.3}{0.1} = 3 \quad gDiff_s = \frac{\Delta x_6}{\delta y_{3-6}} = \frac{0.3}{0.1} = 3$$

The interface conductivities are

$$k_w = k_{5-6} = 10^2 \quad k_n = k_{6-9} = 10^2 \quad k_s = k_{3-6} = 10^2$$

The general form of the equation is written as

$$a_C T_6 + a_W T_5 + a_N T_9 + a_S T_3 = b_C$$

where

$$\begin{aligned} a_W &= FluxF_w = -k_w gDiff_w = -10^2 \times 0.4 = -40 \\ a_N &= FluxF_n = -k_n gDiff_n = -10^2 \times 3 = -300 \\ a_S &= FluxF_s = -k_s gDiff_s = -10^2 \times 3 = -300 \end{aligned}$$

The east coefficient does not appear in the equation as its represents a zero flux boundary condition. Moreover the main coefficient and source term are given by

$$\begin{aligned} a_C &= FLuxC_w + FLuxC_n + FLuxC_s \\ &= 40 + 300 + 300 = 640 \\ b_C &= FLuxV_e = 0 \end{aligned}$$

Substituting, the discretized algebraic equation is obtained as

$$640T_6 - 40T_5 - 300T_9 - 300T_3 = 0$$

Element #7
The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_7}{\delta x_{7-8}} = \frac{0.1}{0.15} = 0.667 \quad gDiff_w = \frac{\Delta y_7}{\delta x_{19-7}} = \frac{0.1}{0.05} = 2 \quad gDiff_s = \frac{\Delta x_7}{\delta y_{4-7}} = \frac{0.1}{0.1} = 1$$

The interface conductivities are

$$k_e = k_{7-8} = 3 \times 10^{-3} \quad k_w = k_{19} = 10^{-3} \quad k_n = k_{18} = 10^{-3} \quad k_s = k_{4-7} = 10^{-3}$$

The general form of the equation is written as

$$a_C T_7 + a_E T_8 + a_S T_4 = b_C$$

where

$$a_E = FluxF_e = -k_e gDiff_e = -3 \times 10^{-3} \times 0.667 = -0.002$$
$$a_S = FluxF_s = -k_s gDiff_s = -10^{-3} \times 1 = -0.001$$

The west and north coefficients do not appear in the equation as their influence is integrated through the boundary conditions as

$$R_{eq} = \frac{h_\infty (k_{18}/\delta y_{7-18})}{h_\infty + (k_{18}/\delta y_{7-18})} \Delta x_7 = \frac{20(10^{-3}/0.05)}{20 + 10^{-3}/0.05} 0.1 = 0.001998$$

$$FluxC_w = k_w gDiff_w = 10^{-3} \times 2 = 0.002 \quad FluxV_w = -k_w gDiff_w T_{19} = -0.64$$
$$FluxC_n = R_{eq} = 0.001998 \quad FluxV_n = -\mathrm{R}_{eq} T_\infty = -0.5994$$

The main coefficient and source term can now be calculated and are given by

$$a_C = FLuxC_e + FLuxC_w + FLuxC_n + FLuxC_s$$
$$= 0.002 + 0.002 + 0.001998 + 0.001 = 0.006998$$
$$b_C = -FLuxV_w - FLuxV_n = 0.64 + 0.5994 = 1.2394$$

Substituting, the discretized algebraic equation is obtained as

$$0.006998 T_7 - 0.002 T_8 - 0.001 T_4 = 1.2394$$

Element #8
The needed diffusion terms are calculated as

$$gDiff_e = \frac{\Delta y_8}{\delta x_{8-9}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_w = \frac{\Delta y_8}{\delta x_{7-8}} = \frac{0.1}{0.15} = 0.667 \quad gDiff_s = \frac{\Delta x_8}{\delta y_{5-8}} = \frac{0.2}{0.1} = 2$$

The interface conductivities are

$$k_e = k_{8-9} = 10^2 \quad k_w = k_{7-8} = 3 \times 10^{-3} \quad k_n = k_{17} = 10^2 \quad k_s = k_{5-8} = 10^2$$

The general form of the equation is written as

$$a_C T_8 + a_E T_9 + a_W T_7 + a_S T_5 = b_C$$

where

$$\begin{aligned} a_E &= FluxF_e = -k_e gDiff_e = -10^2 \times 0.4 = -40 \\ a_W &= FluxF_w = -k_w gDiff_w = -3 \times 10^{-3} \times 0.667 = -0.002 \\ a_S &= FluxF_s = -k_s gDiff_s = -10^2 \times 2 = -200 \end{aligned}$$

The north coefficient does not appear in the equation as its influence is integrated through the boundary condition as

$$R_{eq} = \frac{h_\infty (k_{17}/\delta y_{8-17})}{h_\infty + (k_{17}/\delta y_{8-17})} \Delta x_8 = \frac{20(10^2/0.05)}{20 + 10^2/0.05} 0.2 = 3.9604$$

$$FluxC_n = R_{eq} = 3.9604 \quad FluxV_n = -R_{eq} T_\infty = -1188.12$$

The main coefficient and source term can now be calculated and are given by

$$\begin{aligned} a_C &= FLuxC_e + FLuxC_w + FLuxC_n + FLuxC_s \\ &= 40 + 0.002 + 3.9604 + 200 = 243.9624 \\ b_C &= -FluxV_n = 1188.12 \end{aligned}$$

Substituting, the discretized algebraic equation is obtained as

$$243.9624 T_8 - 40 T_9 - 0.002 T_7 - 200 T_5 = 1188.12$$

Element #9
The needed diffusion terms are calculated as

$$gDiff_w = \frac{\Delta y_9}{\delta x_{8-9}} = \frac{0.1}{0.25} = 0.4 \quad gDiff_s = \frac{\Delta x_9}{\delta y_{6-9}} = \frac{0.3}{0.1} = 3$$

The interface conductivities are

$$k_w = k_{8-9} = 10^2 \quad k_n = k_{16} = 10^2 \quad k_s = k_{6-9} = 10^2$$

The general form of the equation is written as

$$a_C T_9 + a_W T_8 + a_S T_6 = b_C$$

where

$$a_W = FluxF_w = -k_w gDiff_w = -10^2 \times 0.4 = -40$$
$$a_S = FluxF_s = -k_s gDiff_s = -10^2 \times 3 = -300$$

The east and north coefficients do not appear in the equation as their influence is integrated through the boundary conditions as

$$R_{eq} = \frac{h_\infty (k_{16}/\delta y_{9-16})}{h_\infty + (k_{16}/\delta y_{9-16})} \Delta x_9 = \frac{20(10^2/0.05)}{20 + 10^2/0.05} 0.3 = 5.9406$$

$$FluxC_n = R_{eq} = 5.9406$$
$$FluxV_n = -R_{eq} T_\infty = -1782.18$$
$$FluxC_e = FluxV_e = 0$$

The main coefficient and source term can now be calculated and are given by

$$a_C = FLuxC_w + FLuxC_n + FLuxC_s = 40 + 5.9406 + 300 = 345.9406$$
$$b_C = -FLuxV_n = 1782.18$$

Substituting, the discretized algebraic equation is obtained as

$$345.9406T_9 - 40T_8 - 300T_6 = 1782.18$$

Summary of equations
The discretized algebraic equations are given by

$$0.005T_1 - 0.002T_2 - 0.001T_4 = 0.64$$

$$240.002T_2 - 40T_3 - 0.002T_1 - 200T_5 = 20$$

$$340T_3 - 40T_2 - 300T_6 = 30$$

$$0.006T_4 - 0.002T_5 - 0.001T_7 - 0.001T_1 = 0.64$$

$$440.002T_5 - 40T_6 - 0.002T_4 - 200T_8 - 200T_2 = 0$$

$$640T_6 - 40T_5 - 300T_9 - 300T_3 = 0$$

$$0.006998T_7 - 0.002T_8 - 0.001T_4 = 1.2394$$

$$243.9624T_8 - 40T_9 - 0.002T_7 - 200T_5 = 1188.12$$

$$345.9406T_9 - 40T_8 - 300T_6 = 1782.18$$

Solution of Equations

To solve the above system of equations via the Gauss-Seidel method, the equations are rearranged into

$$T_1 = (0.002T_2 + 0.001T_4 + 0.64)/0.005$$

$$T_2 = (40T_3 + 0.002T_1 + 200T_5 + 20)/240.002$$

$$T_3 = (40T_2 + 300T_6 + 30)/340$$

$$T_4 = (0.002T_5 + 0.001T_7 + 0.001T_1 + 0.64)/0.006$$

$$T_5 = (40T_6 + 0.002T_4 + 200T_8 + 200T_2)/440.002$$

$$T_6 = (40T_5 + 300T_9 + 300T_3)/640$$

$$T_7 = (0.002T_8 + 0.001T_4 + 1.2394)/0.006998$$

$$T_8 = (40T_9 + 0.002T_7 + 200T_5 + 1188.12)/243.9624$$

$$T_9 = (40T_8 + 300T_6 + 1782.18)/345.9406$$

The solution is found iteratively with the latest available values used during the solution process. Starting with a uniform initial guess of 300 K, Table 8.1 shows the results during the first two iterations along with the final converged solution.

Table 8.1 Summary of results obtained using the Gauss-Seidel iterative method

Iter	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
0	300	300	300	300	300	300	300	300	300
1	308.000	300.083	300.098	308.000	300.037	300.048	306.859	300.031	300.045
2	309.633	300.131	300.146	309.428	300.078	300.094	307.072	300.071	300.090
⋮	…	…	…	…	…	…	…	…	…
Solution	312.490	305.254	305.254	311.944	305.154	305.154	308.867	305.054	305.054

Values of T along the bottom boundary

These can be calculated from the specified boundary condition, in this case a specified flux, as

$$-k_b \frac{\partial T}{\partial y} = q_b \Rightarrow -k_b \frac{T_C - T_b}{y_C - y_b} = q_b \Rightarrow T_b = T_C + \frac{q_b}{k_b}(y_C - y_b)$$

Using the above equation, the temperatures are calculated as

$$T_{10} = T_1 = 312.490$$
$$T_{11} = T_2 + \frac{q_b}{k_{11}}(y_2 - y_{11}) = 305.254 + \frac{100}{100}(0.05 - 0) = 305.304$$
$$T_{12} = T_3 + \frac{q_b}{k_{12}}(y_3 - y_{12}) = 305.254 + \frac{100}{100}(0.05 - 0) = 305.304$$

Values of T along the right boundary

The above equation with the heat flux set to zero can be used to calculate the temperature along the zero flux boundary, leading to

$$-k_b \frac{\partial T}{\partial y} = 0 \Rightarrow T_b = T_C$$

The boundary temperatures are therefore given by

$$T_{13} = T_3 = 305.254 \quad T_{14} = T_6 = 305.154 \quad T_{15} = T_9 = 305.054$$

Values of T along the top boundary

Using Eq. (8.45) the temperature values along the top boundary are computed as

$$T_{18} = \frac{h_b T_\infty + (k_{18}/\delta y_{7-18}) T_7}{h_b + (k_{18}/\delta y_{7-18})} = \frac{20 \times 300 + (10^{-3}/0.05)308.867}{20 + (10^{-3}/0.05)} = 300.009$$
$$T_{17} = \frac{h_b T_\infty + (k_{17}/\delta y_{8-17}) T_8}{h_b + (k_{17}/\delta y_{8-17})} = \frac{20 \times 300 + (10^{2}/0.05)305.054}{20 + (10^{2}/0.05)} = 305.004$$
$$T_{16} = \frac{h_b T_\infty + (k_{16}/\delta y_{9-16}) T_9}{h_b + (k_{16}/\delta y_{9-16})} = \frac{20 \times 300 + (10^{2}/0.05)305.054}{20 + (10^{2}/0.05)} = 305.004$$

Total heat transfer along the left boundary
The total heat transfer along the west boundary is given by

$$\begin{aligned} Q_{Left} &= q_{21}\Delta y_{21} + q_{20}\Delta y_{20} + q_{19}\Delta y_{19} \\ &= k_{21}\frac{T_1 - T_{21}}{\delta x_{21-1}}\Delta y_{21} + k_{20}\frac{T_4 - T_{20}}{\delta x_{20-4}}\Delta y_{20} + k_{19}\frac{T_7 - T_{19}}{\delta x_{19-7}}\Delta y_{19} \\ &= \frac{10^{-3}\times 0.1}{0.05}[312.49 + 311.944 + 308.867 - 3\times 320] \\ &= -0.053398\,\mathrm{W} \end{aligned}$$

Total heat transfer along the top boundary
Along the top boundary, the total heat transfer is computed as

$$\begin{aligned} Q_{top} &= q_{18}\Delta x_{18} + q_{17}\Delta x_{17} + q_{16}\Delta x_{16} \\ &= h_\infty[\Delta x_{18}(T_{18} - T_\infty) + \Delta x_{17}(T_{17} - T_\infty) + \Delta x_{16}(T_{16} - T_\infty)] \\ &= 20[0.1(300.009 - 300) + 0.2(305.004 - 300) + 0.3(305.004 - 300)] \\ &= 50.058\,\mathrm{W} \end{aligned}$$

Total heat transfer along the bottom boundary
Knowing the heat flux, the total amount of heat transfer is found as

$$Q_{Bottom} = -q_b\Delta y = -100\times 0.5 = -50\,\mathrm{W}$$

Check for energy conservation
For conservation, the sum of heat entering and leaving the domain should be equal to zero. The sum is computed as

$$Q_{Top} + Q_{Left} + Q_{Bottom} = 50.058 - 0.053398 - 50 = 0.004602 \approx 0$$

The slight deviation from zero is due to the use of a limited number of digits during computations.

8.5 Non-Cartesian Orthogonal Grids

Let us now consider the case of an orthogonal grid that is not oriented along the x and y axes. As shown in Fig. 8.8, such a grid can be obtained by rotating the Cartesian grid of Fig. 8.1 by some angle.

The discretized equation for this grid should be exactly the same as the one obtained for the Cartesian grid. Moreover for similar boundary conditions, the same solution should be obtained.

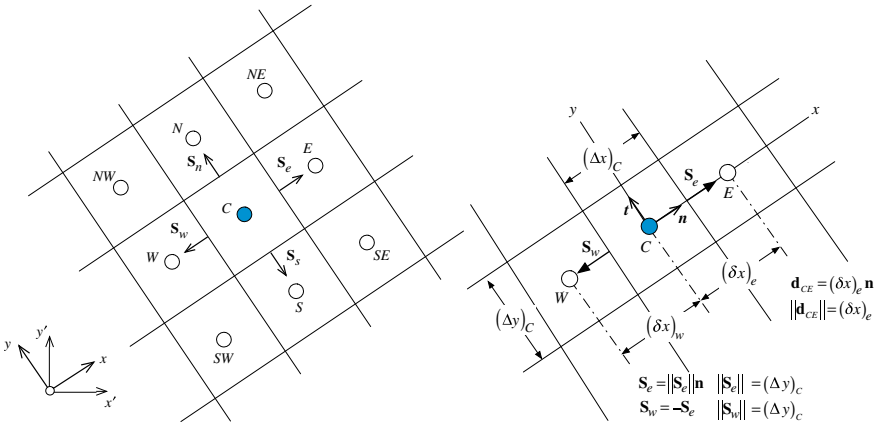


Fig. 8.8 Example of non-Cartesian orthogonal grids

Consider again the steady state conduction Eq. (8.1)

$$\nabla \cdot \mathbf{J}^{\phi,D} = Q^{\phi} \tag{8.58}$$

As before its discretized form is given by

$$\sum_{f \sim nb(C)} -\left(\Gamma^{\phi} \nabla \phi\right)_f \cdot \mathbf{S}_f = Q_C^{\phi} V_C \tag{8.59}$$

Considering the discretization along face e, the following is obtained:

$$\mathbf{J}_e^{\phi,D} \cdot \mathbf{S}_e = -\Gamma_e^{\phi} (\nabla \phi \cdot \mathbf{n})_e S_e = -\Gamma_e^{\phi} \left(\frac{\partial \phi}{\partial n}\right)_e S_e \tag{8.60}$$

where now

$$(\nabla \phi \cdot \mathbf{n})_e = \left(\frac{\partial \phi}{\partial n}\right)_e \tag{8.61}$$

is the gradient of ϕ at face e along the $\mathbf{n}$ direction. Assuming again a linear profile for ϕ along the $\mathbf{n}$ coordinate axis, the gradient can be written as

$$\left(\frac{\partial \phi}{\partial n}\right)_e = \frac{\phi_E - \phi_C}{d_{CE}} \tag{8.62}$$

The discretization of other terms proceeds as for a Cartesian grid, leading to the same final discretization equation.

8.6 Non-orthogonal Unstructured Grid

8.6.1 Non-orthogonality

In the above configurations, the fluxes were normal to the face. In general, structured curvilinear grids and unstructured grids are non-orthogonal [2–5].Therefore the surface vector $\mathbf{S}_f$ and the vector **CF** joining the centroids of the elements straddling the interface are not collinear (see Fig. 8.9). In this case the gradient normal to the surface cannot be written as a function of ϕ_F and ϕ_C, as it has a component in the direction perpendicular to **CF**.

Thus while on orthogonal grids the gradient in the direction normal to the interface yields

$$(\nabla\phi \cdot \mathbf{n})_f = \left(\frac{\partial\phi}{\partial n}\right)_f = \frac{\phi_F - \phi_C}{\|\mathbf{r}_F - \mathbf{r}_C\|} = \frac{\phi_F - \phi_C}{d_{CF}} \tag{8.63}$$

because **CF** and **n** (the unit vector normal to the surface) are aligned, on non-orthogonal grids [6, 7], the gradient direction that yields an expression involving ϕ_F and ϕ_C will have to be along the line joining the two points C and F.

If **e** represents the unit vector along the direction defined by the line connecting nodes C and F then

$$\mathbf{e} = \frac{\mathbf{r}_F - \mathbf{r}_C}{\|\mathbf{r}_F - \mathbf{r}_C\|} = \frac{\mathbf{d}_{CF}}{d_{CF}} \tag{8.64}$$

Therefore, the gradient in the **e** direction can be written as

$$(\nabla\phi \cdot \mathbf{e})_f = \left(\frac{\partial\phi}{\partial e}\right)_f = \frac{\phi_F - \phi_C}{\|\mathbf{r}_\mathbf{F} - \mathbf{r}_C\|} = \frac{\phi_F - \phi_C}{d_{CF}} \tag{8.65}$$

Thus to achieve the linearization of the flux in non-orthogonal grids, the surface vector $\mathbf{S}_f$ should be written as the sum of two vectors $\mathbf{E}_f$ and $\mathbf{T}_f$, i.e.,

$$\mathbf{S}_f = \mathbf{E}_f + \mathbf{T}_f \tag{8.66}$$

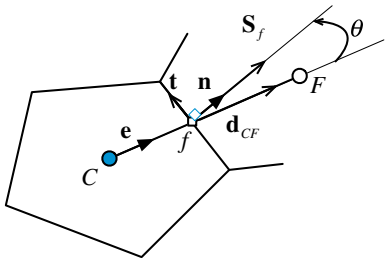


Fig. 8.9 An element in a non-orthogonal mesh system

with $\mathbf{E}_f$ being in the **CF** direction to enable writing part of the diffusion flux as a function of the nodal values ϕ_F and ϕ_C such that

$$\begin{aligned}(\nabla\phi)_f \cdot \mathbf{S}_f &= \underbrace{(\nabla\phi)_f \cdot \overbrace{\mathbf{E}_f}^{E_f\mathbf{e}}}_{orthogonal-like\ contribution} + \underbrace{(\nabla\phi)_f \cdot \mathbf{T}}_{non-orthogonal\ like\ contribution} \\ &= E_f \left(\frac{\partial\phi}{\partial e}\right)_f + (\nabla\phi)_f \cdot \mathbf{T}_f \\ &= E_f \frac{\phi_F - \phi_C}{d_{CF}} + (\nabla\phi)_f \cdot \mathbf{T}_f\end{aligned} \tag{8.67}$$

The first term on the right hand side of Eq. (8.67) represents a contribution similar to the contribution on orthogonal grids, i.e., involving ϕ_F and ϕ_C, while the second term on the right hand side is called cross-diffusion or non-orthogonal diffusion [8] and is due to the non-orthogonality of the grid. Different options for the decomposition of $\mathbf{S}_f$ are available and are discussed next.

8.6.2 Minimum Correction Approach

As shown in Fig. 8.10, the decomposition of $\mathbf{S}_f$ is done in such a way as to keep the non-orthogonal correction in Eq. (8.67) as small as possible, by making $\mathbf{E}_f$ and $\mathbf{T}_f$ orthogonal. As the non-orthogonality increases, the contribution to the diffusion flux from ϕ_F and ϕ_C decreases. In this case the vector $\mathbf{E}_f$ is computed as

$$\mathbf{E}_f = (\mathbf{e} \cdot \mathbf{S}_f)\mathbf{e} = (S_f \cos\theta)\mathbf{e} \tag{8.68}$$

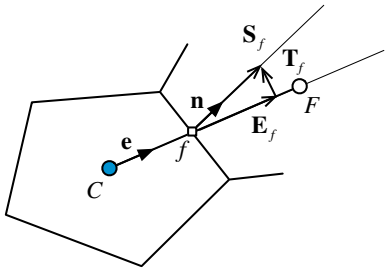


Fig. 8.10 Decomposing $\mathbf{S}_f$ via the minimum correction approach

8.6.3 Orthogonal Correction Approach

This approach, schematically depicted in Fig. 8.11, keeps the contribution of the term involving ϕ_F and ϕ_C the same as on an orthogonal mesh irrespective of the degree of grid non-orthogonality. To achieve this, $\mathbf{E}_f$ is defined as

$$\mathbf{E}_f = S_f \mathbf{e} \tag{8.69}$$

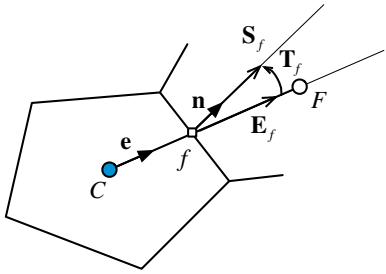


Fig. 8.11 Decomposing $\mathbf{S}_f$ via the orthogonal correction approach

8.6.4 Over-Relaxed Approach

In this approach, the importance of the term involving ϕ_F and ϕ_C is forced to increase as grid non-orthogonality increases. As shown in Fig. 8.12, this is achieved by selecting $\mathbf{T}_f$ to be normal to $\mathbf{S}_f$. Mathematically $\mathbf{E}_f$ is computed as

$$\mathbf{E}_f = \left(\frac{S_f}{\cos\theta}\right)\mathbf{e} = \left(\frac{S_f^2}{S_f \cos\theta}\right)\mathbf{e} = \frac{\mathbf{S}_f \cdot \mathbf{S}_f}{\mathbf{e} \cdot \mathbf{S}_f}\mathbf{e} \tag{8.70}$$

To summarize, the diffusion flux at an element face of a non-orthogonal grid cannot be written solely in terms of the values at the nodes straddling the face.

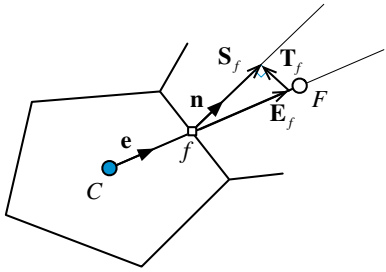


Fig. 8.12 Decomposing $\mathbf{S}_f$ via the over-relaxed approach

A term that accounts for non-orthogonality has to be added. This term is denoted in the literature by "cross diffusion" and is computed as

$$(\nabla\phi)_f \cdot \mathbf{T}_f = (\nabla\phi)_f \cdot \left(\mathbf{S}_f - \mathbf{E}_f\right) = \begin{cases} (\nabla\phi)_f \cdot (\mathbf{n} - \cos\theta\mathbf{e})S_f & \textit{minimum correction} \\ (\nabla\phi)_f \cdot (\mathbf{n} - \mathbf{e})S_f & \textit{normal correction} \\ (\nabla\phi)_f \cdot \left(\mathbf{n} - \dfrac{1}{\cos\theta}\mathbf{e}\right)S_f & \textit{over-relaxed} \end{cases} \tag{8.71}$$

For orthogonal meshes $\mathbf{n}$ and $\mathbf{e}$ are collinear, the angle θ shown in Fig. 8.9 is zero, and the cross-diffusion term is zero. When cross diffusion is not zero, and since it cannot be written as a function of ϕ_F and ϕ_C, it is added as a source term in the element algebraic equation.

All approaches described above are correct and satisfy Eq. (8.59). The difference between these methods is in their accuracy and stability on non-orthogonal meshes. The over-relaxed approach has been found to be the most stable even when the grid is highly non-orthogonal. Nevertheless, the final general form of the discretized diffusion term is the same for all three approximations.

8.6.5 Treatment of the Cross-Diffusion Term

The cross diffusion term cannot be expressed in terms of nodal values. Due to this fact, it is treated in a deferred correction manner by computing its value using the current gradient field and adding it as a source term on the right hand side of the algebraic equation. Gradients are computed at the main grid points, as described next, and values at the interfaces are obtained by interpolation.

8.6.6 Gradient Computation

In discretizing the diffusion term over a one-dimensional or orthogonal multi-dimensional computational domain, it was shown that the gradient of ϕ can be explicitly written as a function of the cell nodal ϕ values. In non-orthogonal domains however, the computation of the diffusion flux was found to be more complex. The non-orthogonal component of the gradient could not be linearized and written as a function of nodal values, but rather had to be moved to the right hand side and evaluated explicitly. This means that the gradient has to be evaluated in order to incorporate its non-orthogonal contribution in the discretization equation. One widely used approach for computing the gradient at a cell is the

Green-Gauss or gradient theorem, which states that for any closed volume V, surrounded by a surface ∂V the following holds:

$$\int_V \nabla\phi \, dV = \oint_{\partial V} \phi \, d\mathbf{S} \tag{8.72}$$

where $d\mathbf{S}$ is the outward pointing incremental surface vector. In order to obtain a discrete version of this equation, the mean value theorem is applied according to which the integral on the left hand side of Eq. (8.72) and the average gradient over the volume $\boldsymbol{V}$ are related by

$$\overline{\nabla\phi} V = \int_V \nabla\phi \, dV \tag{8.73}$$

Combining Eqs. (8.72) and (8.73), the average gradient over element C shown in Fig. 8.9 is found to be

$$\overline{\nabla\phi}_C = \frac{1}{V_C} \oint_{\partial V_C} \phi_f \mathbf{S}_f \tag{8.74}$$

Next the integral over a cell face is approximated by the face centroid value times the face area. Thus $\overline{\nabla\phi}_C$, or simply $\nabla\phi_C$, is computed as

$$\nabla\phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_f \mathbf{S}_f \tag{8.75}$$

The gradient at the face of an element can then be obtained as the weighted average of the gradients at the centroids straddling the interface and is given by

$$\nabla\phi_f = g_C \nabla\phi_C + g_F \nabla\phi_F \tag{8.76}$$

where g_C and g_F are geometric interpolation factors related to the position of the element face f with respect to the nodes C and F.

8.6.7 Algebraic Equation for Non-orthogonal Meshes

Splitting the surface vector $\mathbf{S}_f$ into the two $\mathbf{E}_f$ and $\mathbf{T}_f$ vectors and substituting its equivalent expression into the semi-discretized equation of the diffusion fluxes, yield

$$\begin{aligned}\sum_{f\sim nb(C)}\left(\mathbf{J}_f^{\phi,D}\cdot\mathbf{S}_f\right) &= \sum_{f\sim nb(C)}\left(-\left(\Gamma^{\phi}\nabla\phi\right)_f\cdot\left(\mathbf{E}_f+\mathbf{T}_f\right)\right)\\ &= \underbrace{\sum_{f\sim nb(C)}\left(-\left(\Gamma^{\phi}\nabla\phi\right)_f\cdot\mathbf{E}_f\right)}_{Orthogonal\ Linearizeable\ Part}+\underbrace{\sum_{f\sim nb(C)}\left(-\left(\Gamma^{\phi}\nabla^{\phi}\right)_f\cdot\mathbf{T}_f\right)}_{Non-Orthogonal\ Non-Linearizeable\ Part}\\ &= \sum_{f\sim nb(C)}\left(-\Gamma_f^{\phi}E_f\frac{(\phi_F-\phi_C)}{d_{CF}}\right)+\sum_{f\sim nb(C)}\left(-\left(\Gamma^{\phi}\nabla\phi\right)_f\cdot\mathbf{T}_f\right)\\ &= \sum_{f\sim nb(C)}\Gamma_f^{\phi}gDiff_f(\phi_C-\phi_F)+\sum_{f\sim nb(C)}\left(-\left(\Gamma^{\phi}\nabla\phi\right)_f\cdot\mathbf{T}_f\right)\\ &= \left(\sum_{f\sim nb(C)}FluxC_f\right)\phi_C+\sum_{f\sim nb(C)}\left(FluxF_f\phi_F\right)+\sum_{f\sim nb(C)}\left(FluxV_f\right)\end{aligned} \tag{8.77}$$

where again $gDiff_f$ is a geometric diffusion coefficient defined as

$$gDiff_f=\frac{E_f}{d_{CF}} \tag{8.78}$$

Using the above form of the diffusion fluxes and expanding, the final form of the discretized diffusion equation over unstructured/structured non-orthogonal grid is obtained as

$$a_C\phi_C+\sum_{F\sim NB(C)}a_F\phi_F=b_C \tag{8.79}$$

where

$$\begin{aligned}a_F &= FluxF_f=-\Gamma_f^{\phi}gDiff_f\\ a_C &= \sum_{f\sim nb(C)}FluxC_f=-\sum_{f\sim nb(C)}FluxF_f=\sum_{f\sim nb(C)}\Gamma_f^{\phi}gDiff_f\\ b_C &= Q_C^{\phi}V_C-\sum_{f\sim nb(C)}\left(FluxV_f\right)=Q_C^{\phi}V_C+\sum_{f\sim nb(C)}\left(\left(\Gamma^{\phi}\nabla\phi\right)_f\cdot\mathbf{T}_f\right)\end{aligned} \tag{8.80}$$

Note the change in sign of the non-orthogonal term on the right hand side of the equation.

Example 2

For the polygonal element C and its neighbor F shown in Fig. 8.13, the solution at any point satisfies $\phi = x^2 + y^2 + x^2y^2$. *If the volume of cell C is* $V_C = 8.625$ *calculate the following:*

1. *The gradient of* ϕ *at* (i.e., $\nabla\phi_C$) *both numerically and analytically.*
2. *The analytical value of* $\nabla\phi_F$.
3. *Interpolate between the numerical value of* $\nabla\phi_C$ *and the analytical value of* $\nabla\phi_F$ *to find an approximate value for* $\nabla\phi_{f_1}$. *Compare with the analytical value of* $\nabla\phi_{f_1}$.
4. *Express* $\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1}$ *in terms of* ϕ_C *and* ϕ_F *using*
   (a) *The minimum correction approach*
   (b) *The orthogonal correction approach*
   (c) *The over-relaxed approach*

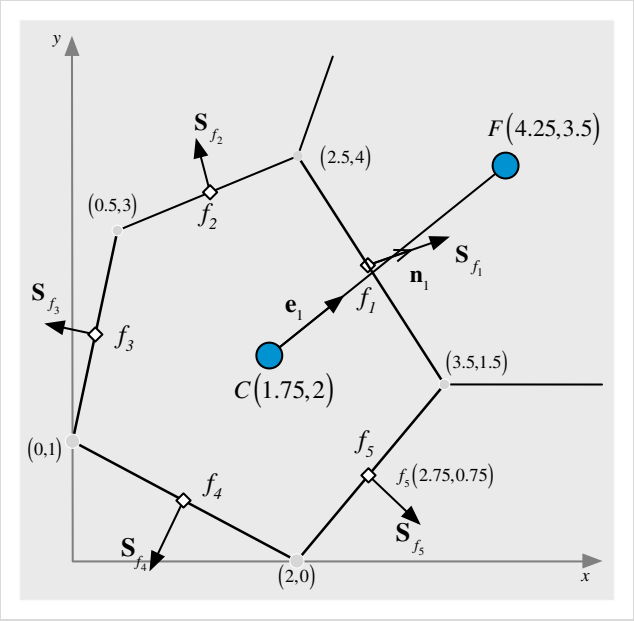


Fig. 8.13 A polygonal element with its geometric entities

Solution

1. Knowing that $\phi = x^2 + y^2 + x^2y^2$ at any point in the domain, the value of the gradient at any point can be calculated as

$$\nabla\phi = \frac{\partial\phi}{\partial x}\mathbf{i} + \frac{\partial\phi}{\partial y}\mathbf{j} = \left(2x + 2xy^2\right)\mathbf{i} + \left(2y + 2yx^2\right)\mathbf{j}$$

Therefore the analytical value of $\nabla\phi_C$ is given by

$$\nabla\phi_C = 17.5\mathbf{i} + 16.25\mathbf{j}$$

The numerical value of $\nabla\phi_C$ can be computed using

$$\nabla\phi_C = \frac{1}{V_C}\sum_{f\sim nb(C)} \phi_f \mathbf{S}_f$$

Therefore the values of ϕ_f and $\mathbf{S}_f$ are required. Using the given analytical expression, the values of ϕ at the various locations are found as

$$\phi_C = 1.75^2 + 2^2 + 1.75^2 \times 2^2 = 19.3125$$

$$\phi_F = 4.25^2 + 3.5^2 + 4.25^2 \times 3.5^2 = 251.578125$$

$$\phi_{f_1} = 3^2 + 2.75^2 + 3^2 \times 2.75^2 = 84.625$$

$$\phi_{f_2} = 1.5^2 + 3.5^2 + 1.5^2 \times 3.5^2 = 42.0625$$

$$\phi_{f_3} = 0.25^2 + 2^2 + 0.25^2 \times 2^2 = 4.3125$$

$$\phi_{f_4} = 1^2 + 0.5^2 + 1^2 \times 0.5^2 = 1.5$$

$$\phi_{f_5} = 2.75^2 + 0.75^2 + 2.75^2 \times 0.75^2 = 12.37890625$$

The surface vector is given by

$$\mathrm{S}_f = \pm(\Delta y\mathbf{i} - \Delta x\mathbf{j})$$

where the correct sign is chosen such that the vector is pointing outward. This is done by computing the surface vector as $\mathbf{S}_f = \Delta y\mathbf{i} - \Delta x\mathbf{j}$ and then performing the dot product of $\mathbf{S}_f$ with the distance vector $\mathbf{d}_{CF}$ (pointing from C to F). If the product is positive then the direction of $\mathbf{S}_f$ is correct. If the product is negative then the sign of $\mathbf{S}_f$ should be reversed, as shown below.

The distance vectors are computed as

$$\mathbf{d}_{Cf_1} = (3 - 1.75)\mathbf{i} + (2.75 - 2)\mathbf{j} = 1.25\mathbf{i} + 0.75\mathbf{j}$$

$$\mathbf{d}_{Cf_2} = (1.5 - 1.75)\mathbf{i} + (3.5 - 2)\mathbf{j} = -0.25\mathbf{i} + 1.5\mathbf{j}$$

$$\mathbf{d}_{Cf_3} = (0.25 - 1.75)\mathbf{i} + (2 - 2)\mathbf{j} = -1.5\mathbf{i}$$

$$\mathbf{d}_{Cf_4} = (1 - 1.75)\mathbf{i} + (0.5 - 2)\mathbf{j} = -0.75\mathbf{i} - 1.5\mathbf{j}$$

$$\mathbf{d}_{Cf_5} = (2.75 - 1.75)\mathbf{i} + (0.75 - 2)\mathbf{j} = \mathbf{i} - 1.25\mathbf{j}$$

while the tentative surface vectors are given by

$$\mathbf{S}_{f_1} = (4 - 1.5)\mathbf{i} - (2.5 - 3.5)\mathbf{j} = 2.5\mathbf{i} + \mathbf{j}$$

$$\mathbf{S}_{f_2} = (4 - 3)\mathbf{i} - (2.5 - 0.5)\mathbf{j} = \mathbf{i} - 2\mathbf{j}$$

$$\mathbf{S}_{f_3} = (3 - 1)\mathbf{i} - (0.5 - 0)\mathbf{j} = 2\mathbf{i} - 0.5\mathbf{j}$$

$$\mathbf{S}_{f_4} = (1 - 0)\mathbf{i} - (0 - 2)\mathbf{j} = \mathbf{i} + 2\mathbf{j}$$

$$\mathbf{S}_{f_5} = (1.5 - 0)\mathbf{i} - (3.5 - 2)\mathbf{j} = 1.5\mathbf{i} - 1.5\mathbf{j}$$

Performing the dot products, the obtained values are

$$\mathbf{S}_{f_1} \cdot \mathbf{d}_{Cf_1} = (2.5\mathbf{i} + \mathbf{j}) \cdot (1.25\mathbf{i} + 0.75\mathbf{j}) = 3.875 > 0$$

$$\mathbf{S}_{f_2} \cdot \mathbf{d}_{Cf_2} = (\mathbf{i} - 2\mathbf{j}) \cdot (-0.25\mathbf{i} + 1.5\mathbf{j}) = -3.25 < 0$$

$$\mathbf{S}_{f_3} \cdot \mathbf{d}_{Cf_3} = (2\mathbf{i} - 0.5\mathbf{j}) \cdot (-1.5\mathbf{i}) = -3 < 0$$

$$\mathbf{S}_{f_4} \cdot \mathbf{d}_{Cf_4} = (\mathbf{i} + 2\mathbf{j}) \cdot (-0.75\mathbf{i} - 1.5\mathbf{j}) = -3.75 < 0$$

$$\mathbf{S}_{f_5} \cdot \mathbf{d}_{Cf_5} = (1.5\mathbf{i} - 1.5\mathbf{j}) \cdot (\mathbf{i} - 1.25\mathbf{j}) = 3.375 > 0$$

Therefore the correct values of the surface vectors should be

$$\mathbf{S}_{f_1} = 2.5\mathbf{i} + \mathbf{j} \quad \mathbf{S}_{f_2} = -\mathbf{i} + 2\mathbf{j} \quad \mathbf{S}_{f_3} = -2\mathbf{i} + 0.5\mathbf{j} \quad \mathbf{S}_{f_4} = -\mathbf{i} - 2\mathbf{j} \quad \mathbf{S}_{f_5} = 1.5\mathbf{i} - 1.5\mathbf{j}$$

Thus, the numerical value of the gradient at C is obtained as

$$\begin{aligned}\nabla\phi_C &= \frac{1}{8.625}\begin{bmatrix} 84.625(2.5\mathbf{i} + \mathbf{j}) + 42.0625(-\mathbf{i} + 2\mathbf{j}) + 4.3125(-2\mathbf{i} + 0.5\mathbf{j}) \\ +1.5(-\mathbf{i} - 2\mathbf{j}) + 12.37890625(1.5\mathbf{i} - 1.5\mathbf{j}) \end{bmatrix} \\ &= 20.63111\mathbf{i} + 17.31454\mathbf{j}\end{aligned}$$

which is close to the analytical value.

2. The analytical value of $\nabla\phi_F$ is easily calculated as

$$\nabla\phi_F = 112.625\mathbf{i} + 133.4375\mathbf{j}$$

3. The interpolated value of $\nabla\phi_{f_1}$ is obtained using

$$\nabla\phi_{f_1} = g_{f_1}\nabla\phi_F + (1 - g_{f_1})\nabla\phi_C$$

where

$$g_{f_1} = \frac{d_{Cf_1}}{d_{Cf_1} + d_{f_1F}}$$

Performing the calculations, the interpolation factor is found to be given by

$$d_{Cf_1} = \sqrt{(x_{f_1} - x_C)^2 + (y_{f_1} - y_C)^2} = \sqrt{(3 - 1.75)^2 + (2.75 - 2)^2} = 1.4577$$
$$d_{fF} = \sqrt{(x_F - x_{f_1})^2 + (y_F - y_{f_1})^2} = \sqrt{(4.25 - 3)^2 + (3.5 - 2.75)^2} = 1.4577$$
$$g_{f_1} = \frac{1.4577}{1.4577 + 1.4577} = 0.5$$

leading to the following value for $\nabla\phi_{f_1}$:

$$\nabla\phi_{f_1} = 66.628055\mathbf{i} + 75.37602\mathbf{j}$$

The analytical value of $\nabla\phi_{f_1}$ is easily obtained as

$$\nabla\phi_{f_1} = 51.375\mathbf{i} + 55\mathbf{j}$$

Again values are close.

4. The general form of $\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1}$ is given by

$$-\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1} = \underbrace{E_{f_1}\frac{(\phi_C - \phi_F)}{d_{CF}}}_{\substack{orthogonal-like \\ contribution}} + \underbrace{-\nabla\phi_{f_1} \cdot \mathbf{T}_{f_1}}_{\substack{non-orthogonal\ like \\ contribution}}$$

with

$$\mathbf{d}_{CF} = (4.25 - 1.75)\mathbf{i} + (3.25 - 2)\mathbf{j} = 2.5\mathbf{i} + 1.5\mathbf{j} \Rightarrow d_{CF} = \sqrt{2.5^2 + 1.5^2} = 2.9155$$

The unit vector $\mathbf{e}_1$ in the direction of $\mathbf{d}_{CF}$ is calculated using

$$\mathbf{e}_1 = \frac{\mathbf{d}_{CF}}{d_{CF}} = 0.8575\mathbf{i} + 0.5145\mathbf{j}$$

(a) The minimum correction approach
Using this approach, the expression for $\mathbf{E}_{f_1}$ is computed as

$$\mathbf{E}_{f_1} = \left(\mathbf{e}_1 \cdot \mathbf{S}_{f_1}\right)\mathbf{e}_1 = 2.279\mathbf{i} + 1.368\mathbf{j} \Rightarrow E_{f_1} = 2.658$$

The normal component is computed as

$$\mathbf{T}_{f_1} = \mathbf{S}_{f_1} - \mathbf{E}_{f_1} = 0.221\mathbf{i} - 0.368\mathbf{j} \Rightarrow \nabla\phi_{f_1} \cdot \mathbf{T}_{f_1} = -13.014$$

The diffusion flux at the face becomes

$$-\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1} = 0.9122(\phi_C - \phi_F) + 13.014$$

(b) The orthogonal correction approach
In this approach, the expression for $\mathbf{E}_{f_1}$ is computed as

$$\mathbf{E}_{f_1} = S_{f_1}\mathbf{e}_1 = 2.309\mathbf{i} + 1.385\mathbf{j} \Rightarrow E_{f_1} = 2.693$$

The normal component is found to be

$$\mathbf{T}_{f_1} = 0.191\mathbf{i} - 0.385\mathbf{j} \Rightarrow \nabla\phi_{f_1} \cdot \mathbf{T}_{f_1} = -16.294$$

The diffusion flux at the face becomes

$$-\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1} = 0.924(\phi_C - \phi_F) + 16.294$$

(c) The over-relaxed approach
The expression for $\mathbf{E}_{f_1}$ is computed as

$$\mathbf{E}_{f_1} = \frac{\mathbf{S}_{f_1} \cdot \mathbf{S}_{f_1}}{\mathbf{e}_1 \cdot \mathbf{S}_{f_1}}\mathbf{e}_1 = 2.339\mathbf{i} + 1.403\mathbf{j} \Rightarrow E_{f_1} = 2.728$$

The normal component is found to be

$$\mathbf{T}_{f_1} = 0.161\mathbf{i} - 0.403\mathbf{j} \Rightarrow \nabla\phi_{f_1} \cdot \mathbf{T}_{f_1} = -19.649$$

The diffusion flux at the face becomes

$$\begin{aligned} -\nabla\phi_{f_1} \cdot \mathbf{S}_{f_1} &= FluxC_{f_1}\phi_C + FluxF_{f_1}\phi_F + FluxV_{f_1} \\ &= 0.936(\phi_C - \phi_F) + 19.649 \end{aligned}$$

8.6.8 Boundary Conditions for Non-orthogonal Grids

The treatment of boundary conditions for non-orthogonal grids is similar to that for orthogonal grids with some minor differences related to the non-orthogonal diffusion contribution. This is outlined next.

8.6.8.1 Dirichlet Boundary Condition

For the case of a Dirichlet boundary condition, i.e., when ϕ is specified by the user at the boundary as shown in Fig. 8.14, the boundary discretization proceeds as in orthogonal-grids. However, there is a need now to account for the cross-diffusion, which arises on boundary faces as on interior faces. This happens whenever the surface vector is not collinear with the vector joining the centroids of the element and boundary face. The diffusion flux along the boundary face is discretized as

$$\begin{aligned}\mathbf{J}_b^{\phi,D}\cdot\mathbf{S}_b &= -\Gamma_b^{\phi}(\nabla\phi)_b\cdot\mathbf{S}_b = -\Gamma_b^{\phi}(\nabla\phi)_b\cdot(\mathbf{E}_b+\mathbf{T}_b)\\ &= -\Gamma_b^{\phi}\left(\frac{\phi_b-\phi_C}{d_{Cb}}\right)E_b-\Gamma_b^{\phi}(\nabla\phi)_b\cdot\mathbf{T}_b\\ &= FluxC_b\phi_C+FluxV_b\end{aligned} \tag{8.81}$$

where

$$\begin{aligned}FluxC_b &= \Gamma_b^{\phi}gDiff_b\\ FluxV_b &= -\Gamma_b^{\phi}gDiff_b\phi_b-\Gamma_b^{\phi}(\nabla\phi)_b\cdot\mathbf{T}_b\\ gDiff_b &= \frac{E_b}{d_{Cb}}\end{aligned} \tag{8.82}$$

Substituting into Eq. (8.79), the modified coefficients are obtained as

$$\begin{aligned}a_F &= FluxF_f \quad a_C = \sum_{f\sim nb(C)} FluxC_f+FluxC_b\\ b_C &= Q_C^{\phi}V_C-FluxV_b-\sum_{f\sim nb(C)} FluxV_f\end{aligned} \tag{8.83}$$

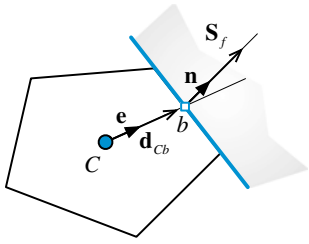


Fig. 8.14 Dirichlet boundary for a non-orthogonal mesh

8.6.8.2 Neumann Boundary Condition

The Neumann type condition for non-orthogonal grids follows that for orthogonal grids. In this case the user-specified flux at the boundary is just added as a source term as with orthogonal grid. The algebraic equation at the boundary is given by Eq. (8.40) with the modified coefficients specified by Eq. (8.41).

8.6.8.3 Mixed Boundary Condition

For the mixed boundary condition case (Fig. 8.5), denoting the convection transfer coefficient by h_∞ and the surrounding value of ϕ by ϕ_∞, the diffusion flux at the boundary can be written as

$$\begin{aligned}\mathbf{J}_b^{\phi,D}\cdot\mathbf{S}_b &= -\Gamma_b^\phi\left(\frac{\phi_b-\phi_C}{d_{Cb}}\right)E_b-\Gamma_b^\phi(\nabla\phi)_b\cdot\mathbf{T}_b\\ &= -h_\infty(\phi_\infty-\phi_b)S_b\end{aligned} \tag{8.84}$$

from which an equation for ϕ_b is obtained as

$$\phi_b=\frac{h_\infty S_b\phi_\infty+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}\phi_C-\Gamma_b^\phi(\nabla\phi)_b\cdot\mathbf{T}_b}{h_\infty S_b+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}} \tag{8.85}$$

Substituting ϕ_b back in Eq. (8.84), the flux equation is transformed to

$$\begin{aligned}\mathbf{J}_b^{\phi,D}\cdot\mathbf{S}_b &= \underbrace{-\left[\frac{h_\infty S_b\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}{h_\infty S_b+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}\right]}_{a_b}(\phi_\infty-\phi_C)\underbrace{-\frac{h_\infty S_b\Gamma_b^\phi(\nabla\phi)_b\cdot\mathbf{T}_b}{h_\infty S_b+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}}_{S_b^{\phi,CD}}\\ &= FluxC_b\phi_C+FluxV_b\end{aligned} \tag{8.86}$$

where now

$$\begin{aligned}FluxC_b &= \frac{h_\infty S_b\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}{h_\infty S_b+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}\\ FluxV_b &= -FluxC_b\phi_\infty-\frac{h_\infty S_b\Gamma_b^\phi(\nabla\phi)_b\cdot\mathbf{T}_b}{h_\infty S_b+\dfrac{\Gamma_b^\phi E_b}{d_{Cb}}}\end{aligned} \tag{8.87}$$

and the modified coefficients for the boundary element are obtained as

$$\begin{aligned} a_F &= FluxF_f - \Gamma_f^{\phi} \frac{E_f}{d_{Cf}} \\ a_C &= FluxC_b + \sum_{f \sim nb(C)} FluxC_f \\ b_C &= Q_C^{\phi} V_C - FluxV_b - \sum_{f \sim nb(C)} FluxV_f \end{aligned} \tag{8.88}$$

8.7 Skewness

To evaluate many of the terms constituting the general discretized equation of a quantity ϕ, it is necessary to estimate its value at element faces. An estimated value at the face should be the average value of the entire face. At different steps of the discretization process, linear variation of variables between nodes is assumed. If this is extended to variation of variables along the face, then the average value of any variable ϕ should be found at the face centroid. The common practice is to use a linear interpolation profile and to estimate the value at the face at the intersection between the face and the line connecting the two nodes straddling the face. When the grid is skewed the line does not necessarily pass through the centroid of the face [9, 10]. An example is depicted in Fig. 8.15 where the intersection point of segment [*CF*] with the face is at point, *f*′, which does not coincide with the face centroid, *f*. To keep the overall accuracy of the discretization method second order, all face integrations need to take place at point *f*. Thus a correction for the interpolated value at *f*′ is needed in order to get the value at *f*.

The skewness correction is derived by expressing the value of ϕ at *f* in terms of its value and the value of its derivative at *f*′ via a Taylor expansion such that

$$\phi_f = \phi_{f'} + (\nabla\phi)_{f'} \cdot \mathbf{d}_{f'f} \tag{8.89}$$

where $\mathbf{d}_{f'f}$ is a vector from the intersection point *f*′ to the face centre *f*.

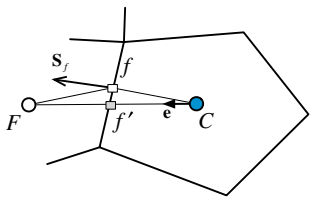


Fig. 8.15 Non-conjunctional elements

8.8 Anisotropic Diffusion

The diffusion equation presented so far assumed the material has no preferred direction for transfer of ϕ with the same diffusion coefficient in all directions, i.e., the medium was assumed to be isotropic. For the case when the diffusion coefficient of the medium is direction dependent, diffusion is said to be anisotropic [11–16]. As mentioned in Chap. 3, some solids are anisotropic for which the semi-discretized diffusion equation becomes

$$\sum_{f\sim nb(C)} \left(-\boldsymbol{\kappa}^{\phi}\cdot\nabla\phi\right)_f \cdot \mathbf{S}_f = S_C^{\phi} V_C \tag{8.90}$$

where $\boldsymbol{\kappa}^{\phi}$ is a second order symmetric tensor. Assuming a general three dimensional situation, the term on the left hand side, through some mathematical manipulation [17], can be rewritten as

$$\left(-\boldsymbol{\kappa}^{\phi}\cdot\nabla\phi\right)_f \cdot \mathbf{S}_f = -\begin{bmatrix} \kappa_{11}^{\phi} & \kappa_{12}^{\phi} & \kappa_{13}^{\phi} \\ \kappa_{21}^{\phi} & \kappa_{22}^{\phi} & \kappa_{23}^{\phi} \\ \kappa_{31}^{\phi} & \kappa_{32}^{\phi} & \kappa_{33}^{\phi} \end{bmatrix}_f \begin{bmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \\ \frac{\partial\phi}{\partial z} \end{bmatrix}_f \cdot \mathbf{S}_f \tag{8.91}$$

Performing matrix multiplication, Eq. (8.91) becomes

$$\left(-\boldsymbol{\kappa}^{\phi}\cdot\nabla\phi\right)_f \cdot \mathbf{S}_f = -\begin{bmatrix} \kappa_{11}^{\phi}\frac{\partial\phi}{\partial x} + \kappa_{12}^{\phi}\frac{\partial\phi}{\partial y} + \kappa_{13}^{\phi}\frac{\partial\phi}{\partial z} \\ \kappa_{21}^{\phi}\frac{\partial\phi}{\partial x} + \kappa_{22}^{\phi}\frac{\partial\phi}{\partial y} + \kappa_{23}^{\phi}\frac{\partial\phi}{\partial z} \\ \kappa_{31}^{\phi}\frac{\partial\phi}{\partial x} + \kappa_{32}^{\phi}\frac{\partial\phi}{\partial y} + \kappa_{33}^{\phi}\frac{\partial\phi}{\partial z} \end{bmatrix}_f \begin{bmatrix} S^x \\ S^y \\ S^z \end{bmatrix}_f \tag{8.92}$$

Further manipulations yield

$$\begin{aligned} \left(-\boldsymbol{\kappa}^{\phi}\cdot\nabla\phi\right)_f \cdot \mathbf{S}_f &= -\begin{bmatrix} \frac{\partial\phi}{\partial x} & \frac{\partial\phi}{\partial y} & \frac{\partial\phi}{\partial z} \end{bmatrix}_f \begin{bmatrix} \kappa_{11}^{\phi} & \kappa_{21}^{\phi} & \kappa_{31}^{\phi} \\ \kappa_{12}^{\phi} & \kappa_{22}^{\phi} & \kappa_{32}^{\phi} \\ \kappa_{13}^{\phi} & \kappa_{23}^{\phi} & \kappa_{33}^{\phi} \end{bmatrix}_f \begin{bmatrix} S^x \\ S^y \\ S^z \end{bmatrix}_f \\ &= -(\nabla\phi)_f \cdot \left[\left(\boldsymbol{\kappa}^{\phi}\right)^T \cdot \mathbf{S}\right]_f = -(\nabla\phi)_f \cdot \mathbf{S}'_f \end{aligned} \tag{8.93}$$

Substituting Eq. (8.93) into Eq. (8.90), the new form of the diffusion equation becomes

$$\sum_{f \sim nb(C)} (-\nabla\phi)_f \cdot \mathbf{S}'_f = Q_C^{\phi} V_C \tag{8.94}$$

It is obvious that in this form the discretization procedure described above can be applied by simply setting Γ^{ϕ} to 1 and replacing $\mathbf{S}_f$ by $\mathbf{S}'_f$. Therefore the same code can be used to solve isotropic and anisotropic diffusion problems.

8.9 Under-Relaxation of the Iterative Solution Process

For a general diffusion problem, Γ^{ϕ} may be a function of the unknown dependent variable ϕ and the grid may be highly non-orthogonal with a large cross diffusion term, which is treated using a deferred correction approach. Therefore large variations in ϕ between iterations result in large source terms and large changes in the coefficients, which may cause divergence of the iterative solution procedure. This divergence is usually due to the non-linearity introduced by the coefficients and the cross-diffusion term, which makes the source term highly affected by the current solution field which is not yet converged. To promote convergence and stabilize the iterative solution process, slowing down the changes in ϕ between iterations is highly desirable and is enforced by a technique called ***under-relaxation***. There are many ways of introducing under-relaxation. One of the practices will be described here, others in Chap. 14. Derivations will be performed on the general discretization equation of the form

$$a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C \tag{8.95}$$

Equation (8.95) can be written as

$$\phi_C = \frac{-\sum_{F \sim NB(C)} a_F \phi_F + b_C}{a_C} \tag{8.96}$$

Let ϕ_C^* represents the value of ϕ_C from the previous iteration. If ϕ_C^* is added to and subtracted from the right hand side, then Eq. (8.96) becomes

$$\phi_C = \phi_C^* + \left(\frac{-\sum_{F \sim NB(C)} a_F \phi_F + b_C}{a_C} - \phi_C^* \right) \tag{8.97}$$

where the expression between the parentheses represents the change in $\boldsymbol{\phi_C}$ produced by the current iteration. This change can be modified by the introduction of a relaxation factor λ^{ϕ}, such that

$$\phi_C = \phi_C^* + \lambda^\phi \left(\frac{-\sum\limits_{F \sim NB(C)} a_F \phi_F + b_C}{a_C} - \phi_C^* \right) \tag{8.98}$$

or

$$\frac{a_C}{\lambda^\phi} \phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C + \frac{(1 - \lambda^\phi) a_C}{\lambda^\phi} \phi_C^* \tag{8.99}$$

At first, it should be noted that at convergence ϕ_C and ϕ_C^* become equal independent of the value of relaxation factor used. This is indeed reflected by Eq. (8.98), which shows that the converged value ϕ_C does satisfy the original equation (Eq. 8.95). This property should be satisfied by any relaxation scheme.

Depending on the value of the relaxation factor λ^ϕ, the equation may be either under relaxed (i.e., $0 < \lambda^\phi < 1$) or over-relaxed (i.e., $\lambda^\phi > 1$). In CFD applications, under relaxation is usually used. A value of λ^ϕ close to 1 implies little under relaxation, while a value close to 0 produces heavy under relaxation effects with very small changes in ϕ_C from iteration to iteration.

The optimum under relaxation factor is problem dependent and is not governed by any general rule. The factors affecting λ^ϕ values include the type of problem solved, the size of the system of equation (i.e., number of grid points in the domain), the grid spacing and its expansion rate, and the adopted iterative method, among others. Usually, values of λ^ϕ are assigned based on experience or from preliminary calculations. Moreover, it is not necessary to use the same under-relaxation value throughout the computational domain and values may vary from iteration to iteration.

Equation (8.99) can be recast into the form of Eq. (8.95), where now the central coefficient a_c becomes

$$a_C \leftarrow \frac{a_C}{\lambda^\phi} \tag{8.100}$$

while the source term is added to produce a new term as

$$b_C \leftarrow b_C + \frac{(1 - \lambda^\phi) a_C}{\lambda^\phi} \phi_C^* \tag{8.101}$$

This method of relaxation plays an important role in stabilizing the solution of non-linear problems.

8.10 Computational Pointers

8.10.1 uFVM

In uFVM the implementation of the diffusion term discretization for interior faces is performed in function **cfdAssembleDiffusionTermInterior**, the core of which is shown in Listing 8.1.

```
gamma_f = cfdInterpolateFromElementsToFaces('Average',gamma);
gamma_f = [gamma_f(iFaces)];
geoDiff_f = [theMesh.faces(iFaces).geoDiff]';
Sf = [theMesh.faces(iFaces).Sf]';
Tf = [theMesh.faces(iFaces).T]';

iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';


theFluxes.FLUXC1f(iFaces,1) =  gamma_f .* gDiff_f;
theFluxes.FLUXC2f(iFaces,1) = -gamma_f .* gDiff_f;

theFluxes.FLUXVf(iFaces,1)  =  gamma_f .* dot(grad_f(:,:)',Tf(:,:)')';
theFluxes.FLUXTf(iFaces,1)       =       theFluxes.FLUXC1f(iFaces)    .*
phi(iOwners)   +   theFluxes.FLUXC2f(iFaces)   .*   phi(iNeighbours)   +
theFluxes.FLUXVf(iFaces);
```

Listing 8.1 Assembly of the diffusion term at interior faces

In the above, **`FLUXC1f`** and **`FLUXC2f`** are equivalent to $FluxC_f$ and $FluxF_f$ in the text and are the coefficients of the owner and neighbor element, respectively. Moreover, **`gamma_f`** and **`gDiff_f`** are arrays defined over all interior faces, and using the dot notation, the expression **`gamma_f.* gDiff_f`** returns an array containing the product of the respective elements of the **`gamma_f`** and **`gDiff_f`** arrays. The dot notation in Matlab® allows for efficient operations over arrays and for the writing of a clearer code and is used in uFVM whenever practical.

The non-orthogonal term is stored in the **`FLUXVf`** coefficient and is equal to **`gamma_f .* dot(grad_f(: , :)',Tf( :, :)')'`**.The total flux passing through face *f*, **`FLUXTf`**, is assembled as in Eq. (8.15).

The diffusion term is setup in **cfdProcessOpenFoamMesh.m** and Listing 8.2 shows the computation of the **`gDiff`** coefficient.

```
theMesh.faces(iFace).dCF = element2.centroid - element1.centroid;
eCF = dCF/cfdMagnitude(dCF);
E = theFace.area*eCF;
theMesh.faces(iFace).gDiff = cfdMagnitude(E)/cfdMagnitude(dCF);
theMesh.faces(iFace).T = theFace.Sf - E;
```

Listing 8.2 Computing the geometric diffusion coefficient

The effects of boundary conditions on the equations of boundary elements should be accounted for and Listing 8.3 shows the assembly of the diffusion term at boundary faces for the case of a Dirichlet boundary condition type.

```
theMesh = cfdGetMesh;
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;
%
theBoundary = theMesh.boundaries(iPatch);
numberOfBFaces = theBoundary.numberOfBFaces;

%
iFaceStart = theBoundary.startFace;
iFaceEnd = iFaceStart+numberOfBFaces-1;
iBFaces = iFaceStart:iFaceEnd;
%
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
iBElements = iElementStart:iElementEnd;

geodiff = [theMesh.faces(iBFaces).geoDiff]';
Tf = [theMesh.faces(iBFaces).T]';
iOwners = [theMesh.faces(iBFaces).iOwner]';

theFluxes.FLUXC1f(iBFaces) =   gamma(iBElements).*geodiff;
theFluxes.FLUXC2f(iBFaces) = - gamma(iBElements).*geodiff;
theFluxes.FLUXVf(iBFaces)                                  =              -
gamma(iBElements).*dot(grad(iBElements,:)',Tf(:,:)')';
```

Listing 8.3 Assembly of the diffusion term at boundary faces for a Dirichlet Condition

The implementation of other boundary condition types can be reviewed in file **`cfdAssembleDiffusionTerm.m`**.

Once the linearized coefficients are computed for each of the interior and boundary faces, then assembly into the global (sparse) matrix can proceed. This approach is used for the discretization of all flux terms in uFVM and the assembly is performed in the **`cfdAssembleIntoGlobalMatrixFaceFluxes`** function. Listing 8.4 shows the assembly into the sparse LHS matrix and the RHS vector of the coefficients at interior faces.

```
%
% Assemble fluxes of interior faces
%
for iFace = 1:numberOfInteriorFaces
    theFace = theMesh.faces(iFace);
    iOwner          = theFace.iOwner;
    iOwnerNeighbourCoef     = theFace.iOwnerNeighbourCoef;
    iNeighbour      = theFace.iNeighbour;
    iNeighbourOwnerCoef = theFace.iNeighbourOwnerCoef;
    %
    %  assemble fluxes for owner cell
    ac(iOwner) = ac(iOwner)
                 + vf_f(iFace)*theFluxes.FLUXC1f(iFace);
 anb{iOwner}(iOwnerNeighbourCoef) = anb{iOwner}(iOwnerNeighbourCoef)
                 + vf_f(iFace)*theFluxes.FLUXC2f(iFace);
    bc(iOwner) = bc(iOwner)
                 - vf_f(iFace)*theFluxes.FLUXTf(iFace);
    %
    %  assemble fluxes for neighbour cell
    ac(iNeighbour) = ac(iNeighbour)
            - vf_f(iFace)*theFluxes.FLUXC2f(iFace);
    anb{iNeighbour}(iNeighbourOwnerCoef)       =       anb{iNeighbour}
(iNeighbourOwnerCoef)
            - vf_f(iFace)*theFluxes.FLUXC1f(iFace);
    bc(iNeighbour)                          = bc(iNeighbour)
            + vf_f(iFace)*theFluxes.FLUXTf(iFace);
end
```

Listing 8.4 Assembly into LHS spare matrix and RHS array

For each interior face the coefficients are assembled into both the owner and neighbor element equations. For the owner the coefficients are (**`ac(iOwner)`**, **`anb(iOwner)(iOwnerNeighbourCoef)`**, and **`bc(iOwner)`**), while for the neighbor the coefficients become: **`ac(iNeighbour)`**, **`anb(iNeighbour)(iNeighbourOwnerCoef)`**, and **`bc(iNeighbour)`**). The different signs used in the assembly of the two equations is due to the fact that the face surface vector is pointing into the neighbor cell and out of the owner cell.

8.10.2 OpenFOAM®

In OpenFOAM® [18] the diffusion term can be evaluated explicitly using "fvc:: laplacian(gamma, phi)"or implicitly via "fvm::laplacian(gamma,phi)" namespaces and functions. The "fvc::laplacian(gamma,phi)" returns a field in which the laplacian of a generic field ϕ (phi) is evaluated at each cell. The field is added to the right hand side of the system of equations. The "fvm::laplacian(gamma,phi)" returns instead an fvMatrix of coefficients evaluated as per Eq. (8.19), which is added to the left hand side of the system of equations, in addition to a field containing the non-orthogonal terms that is added to the right hand side of the system. The definition of the laplacian operator is located in the directory "$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme" in the files "gaussLaplacianScheme.C", "gaussLaplacianScheme.H", and "gaussLaplacianSchemes.C".

The “fvm::laplacian” definition displayed in Listing 8.5 reads

```
template<>
Foam::tmp<Foam::fvMatrix<Foam::Type> >
Foam::fv::gaussLaplacianScheme<Foam::Type, Foam::scalar>::fvmLaplacian
(
    const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = this->mesh();

    GeometricField<scalar, fvsPatchField, surfaceMesh> gammaMagSf
    (
        gamma*mesh.magSf()
    );

    tmp<fvMatrix<Type> > tfvm = fvmLaplacianUncorrected
    (
        gammaMagSf,
        this->tsnGradScheme_().deltaCoeffs(vf),
        vf
    );
    fvMatrix<Type>& fvm = tfvm();

    if (this->tsnGradScheme_().corrected())
    {
        if (mesh.fluxRequired(vf.name()))
        {
            fvm.faceFluxCorrectionPtr() = new
            GeometricField<Type, fvsPatchField, surfaceMesh>
            (
                gammaMagSf*this->tsnGradScheme_().correction(vf)
            );

            fvm.source() -=
                mesh.V()*
                fvc::div
                (
                    *fvm.faceFluxCorrectionPtr()
                )().internalField();
        }
        else
        {
            fvm.source() -=
                mesh.V()*
                fvc::div
                (
                    gammaMagSf*this->tsnGradScheme_().correction(vf)
                )().internalField();
        }
    }
    return tfvm;
}
```

Listing 8.5 Calculation of the Laplacian operator

with the following additional functions (Listing 8.6):

```
template<class Type, class GType>
tmp<fvMatrix<Type> >
gaussLaplacianScheme<Type, GType>::fvmLaplacianUncorrected
(
    const surfaceScalarField& gammaMagSf,
    const surfaceScalarField& deltaCoeffs,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
    deltaCoeffs.dimensions()*gammaMagSf.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm();

    fvm.upper()= deltaCoeffs.internalField()*gammaMagSf.internalField();
    fvm.negSumDiag();

    forAll(vf.boundaryField(), patchi)
    {
        const fvPatchField<Type>& pvf = vf.boundaryField()[patchi];
        const  fvsPatchScalarField&  pGamma  =  gammaMagSf.boundaryField()
[patchi];
        const fvsPatchScalarField& pDeltaCoeffs =
            deltaCoeffs.boundaryField()[patchi];

        if (pvf.coupled())
        {
           fvm.internalCoeffs()[patchi] =
               pGamma*pvf.gradientInternalCoeffs(pDeltaCoeffs);

           fvm.boundaryCoeffs()[patchi] =
              -pGamma*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);

        }
        else
        {
           fvm.internalCoeffs()[patchi] = pGamma*pvf.gradientInternalCoeffs();
           fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
        }
    }
    return tfvm;
}
```

Listing 8.6 Additional functions needed for the calculation of the Laplacian operator

It is worth noting that in OpenFOAM® the assembly takes place directly into the global coefficients, which as described earlier in Chaps. 5, 6, and 7 are stored in three arrays, namely **`fvm.upper()`**, **`fvm.lower()`**, and **`fvm.diag()`**. The main part of the discretization is defined in the **`fvmLaplacianUncorrected`** function, where Eq. (8.19) is evaluated by first defining an **`fvMatrix`** object and then filling its upper triangle by the extra diagonal coefficients (this approach relies on the fact that the Laplacian operator returns a symmetric matrix).

The "**deltaCoeffs.internalField()**" represents the ***gDiff*** field while gamma indicates the diffusion field. The diagonal coefficients are evaluated in **fvm.negSumDiag()**,where the negative sum of the upper and lower coefficients are assembled into the diagonal coefficients as per Eq. (8.18).

The "**fvm.negSumDiag()**" method is defined in **lduMatrix Operations.C** as (Listing 8.7)

```
void Foam::lduMatrix::negSumDiag()
{
    const scalarField& Lower = const_cast<const lduMatrix&>(*this).lower();
    const scalarField& Upper = const_cast<const lduMatrix&>(*this).upper();
    scalarField& Diag = diag();

    const labelUList& l = lduAddr().lowerAddr();
    const labelUList& u = lduAddr().upperAddr();

    for (register label face=0; face<l.size(); face++)
    {
        Diag[l[face]] -= Lower[face];
        Diag[u[face]] -= Upper[face];
    }
}
```

Listing 8.7 Calculation of the negative sum of the upper and lower coefficients

The boundary conditions are implemented exactly as in Eqs. (8.36) and (8.41). In OpenFOAM® the boundary coefficients are stored in "**internalCoeffs**" ($FluxC_b$) and "**boundaryCoeffs**" ($\boldsymbol{FluxV_b}$), already defined in Sect. 7.6.

In "**fvmUncorrected**"only the orthogonal discretization is accounted for. The non-orthogonal contribution is added, as shown in Listing 8.8, using

```
fvm.faceFluxCorrectionPtr() = new
 GeometricField<Type, fvsPatchField, surfaceMesh>
 (
     gammaMagSf*this->tsnGradScheme_().correction(vf)
 );

 fvm.source() -=
     mesh.V()*
     fvc::div
     (
         *fvm.faceFluxCorrectionPtr()
     )().internalField();
```

Listing 8.8 Adding the nonorthogonal diffusion contribution

Again this term represents exactly the implementation of the last term of Eq. (8.77) in which the *snGrad* class wraps into the correction function the non-orthogonal term as shown below in Listing (8.9).

```
template<class Type>
Foam::tmp<Foam::GeometricField<Type,
Foam::fvsPatchField,
Foam::surfaceMesh> >
Foam::fv::correctedSnGrad<Type>::correction
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    const fvMesh& mesh = this->mesh();

    // construct GeometricField<Type, fvsPatchField, surfaceMesh>
    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tssf
    (
        new GeometricField<Type, fvsPatchField, surfaceMesh>
        (
             IOobject
            (
                "snGradCorr("+vf.name()+')',
                vf.instance(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            vf.dimensions()*mesh.nonOrthDeltaCoeffs().dimensions()
        )
    );
    GeometricField<Type, fvsPatchField, surfaceMesh>& ssf = tssf();

    for (direction cmpt = 0; cmpt < pTraits<Type>::nComponents; cmpt++)
    {
        ssf.replace
        (
            cmpt,
            correctedSnGrad<typename pTraits<Type>::cmptType>(mesh)
           .fullGradCorrection(vf.component(cmpt))
        );
    }

template<class Type>
Foam::tmp<Foam::GeometricField<Type,  Foam::fvsPatchField,  Foam::surfaceMesh>
>Foam::fv::correctedSnGrad<Type>::fullGradCorrection
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    const fvMesh& mesh = this->mesh();

    // construct GeometricField<Type, fvsPatchField, surfaceMesh>
    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tssf =
        mesh.nonOrthCorrectionVectors()
      & linear<typename outerProduct<vector, Type>::type>(mesh).interpolate
        (
            gradScheme<Type>::New
            (
                mesh,
                mesh.gradScheme("grad(" + vf.name() + ')')
            )().grad(vf, "grad(" + vf.name() + ')')
        );
    tssf().rename("snGradCorr(" + vf.name() + ')');

    return tssf;
}
```

Listing 8.9 Implementation of the nonorthogonal term into the correction function

The non-orthogonal correction term is defined in the *fullGradCorrection* function where "mesh.nonOrthCorrectionVectors()" returns the **T** vector of Eq. (8.66).

The type of Laplacian discretization to be used is specified in the "fvSchemes" file, which is part of the case definition and is located in the system directory

```
laplacianSchemes
{
    laplacian(gamma,phi) Gauss linear corrected;
}
```

Listing 8.10 The discretization type for the Laplacian operator

(Listing 8.10), in which **Gauss** (the only available choice) defines the standard **Gauss** discretization resulting in Eq. (8.18), **linear** refers to the type of interpolation used for calculating the diffusivity gamma at the face, and **corrected** describes the kind of non-orthogonal correction.

More details on the implementation of boundary conditions in OpenFOAM® are presented in later chapters.

8.11 Closure

In this chapter the discretization of the diffusion equation was described. A number of issues were also addressed, such as the use of orthogonal and non-orthogonal grid systems, the implementation of boundary conditions, and under and over relaxation. The next chapter will concentrate on the calculation of the gradient field.

8.12 Exercises

Exercise I

Consider the diffusion of a property ϕ in the one-dimensional domain shown in Fig. 8.16 with no internal sources. The domain is subdivided into 5 uniform elements of size $\Delta x = 1$ and subject to a Dirichlet and a Neumann condition at boundary '0' and '6', respectively. The diffusion coefficient in the domain is constant with a value of 1, i.e., $\Gamma^{\phi} = 1$.

a. Derive the discrete equation for each of the elements.
b. Solve the system of equations using the Gauss-Seidel iterative method and report the resulting cell-centroid values.
c. Compute the diffusion flux $(-\Gamma d\phi/dx)$ at each of the cell faces and show that conservation is satisfied throughout the domain

d. Compare your solution with the exact solution (Note that even though the computed solution is *conservative*, it is not *exact*).
e. For the case where a zero flux boundary condition is defined at both boundaries '0' and '6', reformulate the equations for elements 1 and 5 and explain why the equations cannot be solved.

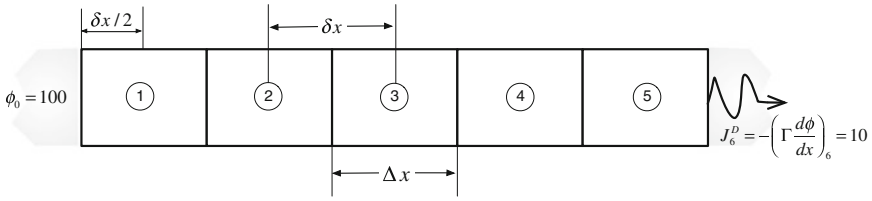


Fig. 8.16 Computational domain for a one dimensional diffusion problem with no internal sources

Exercise 2

A fin is exposed to the surrounding fluid at a temperature T_a = 25 °C, as shown in Fig. 8.17, with a heat transfer coefficient of h = 100 W/m^2 K and a thermal conductivity with value of k = 160 W/m K. The fin has a length L = 0.1 m, a cross sectional area $A = 10^{-5}$ m^2, and a perimeter P = 0.1004 m.

The temperature distribution in the fin is governed by the following differential equation

$$-\frac{d}{dx}\left(k\frac{dT}{dx}\right)+\frac{Ph}{A}(T-T_a)=0$$

Discretize the above equation by subdividing the computational domain into 5 elements of equal size and find the values of the temperature field at the element centroids and boundaries in the following two situations:

a. $T(x=0)=200\ ^\circ\text{C}$ and $T(x=L)=90\ ^\circ\text{C}$
b. $q(x=0)=20\ \text{kW/m}^2$ and $q(x=L)=10\ \text{kW/m}^2$

where q is the rate of heat transfer given by

$$q=-k\frac{dT}{dx}$$

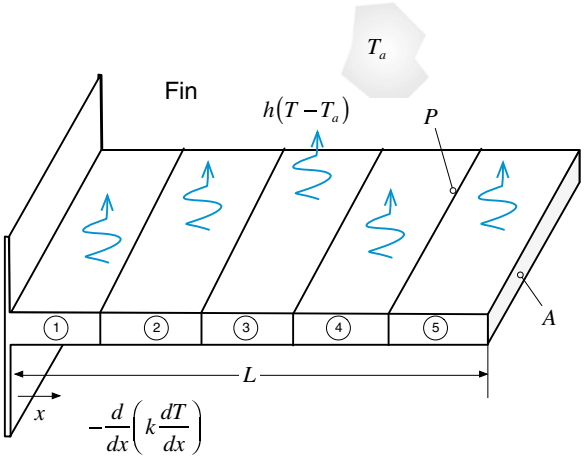


Fig. 8.17 Computational domain for heat transfer from a fin

Exercise 3

The heat conduction in the two-dimensional domain shown in Fig. 8.18 is governed by the following differential equation:

$$-\nabla \cdot k\nabla T = 0$$

The domain is subdivided into uniform elements and the boundary conditions are as shown in the figure.

a. Derive the algebraic equations for all elements.
b. Solve the system of equations obtained and compute the T values at the centroids of the elements.

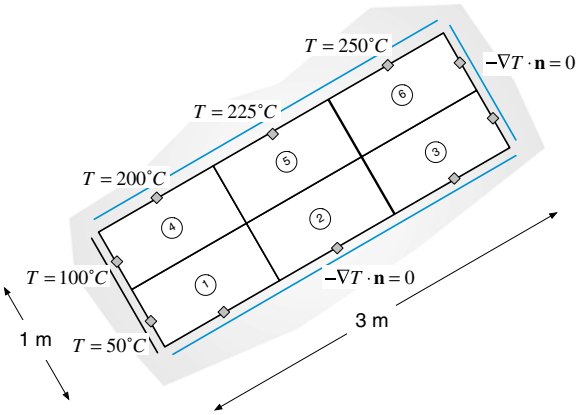


Fig. 8.18 Heat conduction in a two dimensional tilted Cartesian domain

c. Compute the values of T at the bottom and right boundaries.
d. Compute the net heat transfer through the top and left boundaries

Exercise 4

Consider steady state conduction heat transfer in the non-orthogonal domain discretized into the four equal elements shown in Fig. 8.19, where $L = 1$ m. Derive the discretization equations for the cells using two different methods for the decomposition of the diffusion flux and compare the temperature values at the element centroids after 4 coefficient iterations.

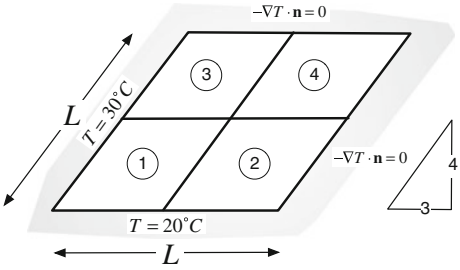


Fig. 8.19 Computational domain for Exercise 4

Exercise 5

Discretize the equation $-\nabla \cdot k\nabla T = Q^T$ where $Q^T = 500$ and $k = 200$, for the mesh composed of quadrilateral triangles of side 0.1 shown in Fig. 8.20. Write the discretized equations in the form of general algebraic equations.

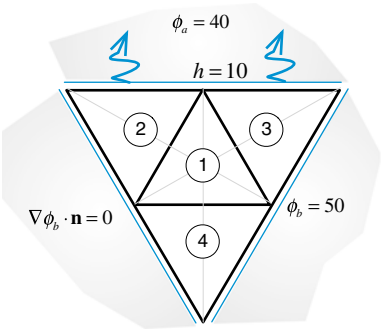


Fig. 8.20 A triangular domain covered with an unstructured grid system

Exercise 6
The gradient for a variable ϕ over the computational domain shown in Fig. 8.21 is given by

$$\nabla\phi = 20\mathbf{i} + 30\mathbf{j}$$

Compute the value of ϕ at nodes 1 and 2 given that the length and width of the computational domain are 10 and 5 cm, respectively.

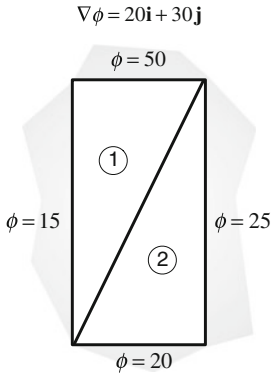


Fig. 8.21 A rectangular domain decomposed into two triangular elements

Exercise 7
Using the mesh constructed in exercise 3 of Chap. 7 (displayed in Fig. 7.12), setup a case in OpenFOAM® and uFVM to solve the diffusion equation subject to the following conditions ($\Gamma^{\phi} = 1$):
Patch#1 $\phi = 1$
Patch#2 $\phi = 0$
Patch#3 $\nabla\phi \cdot \mathbf{n} = 0$
Patch#4 $h_{\infty} = 1$, $\phi_{\infty} = 0.5$

Exercise 8
Build an oblique parallelogram (size of horizontal side is 1 and size of oblique side is 2 inclined at 60 degrees with respect to the horizontal) using blockMesh in OpenFOAM® with the boundary conditions shown in the figure below to solve the diffusion equation with no source term in two dimensions. Generate a grid by decomposing each side of the parallelogram into 50 equal segments. Setup the case and using a diffusion coefficient of 1 compare the convergence history for the three different approaches mentioned in this chapter to resolve the non-orthogonal term for the diffusion equation (one at a time) in uFVM and in OpenFOAM®. Use an under-relaxation factor of value 0.9 (Fig. 8.22).

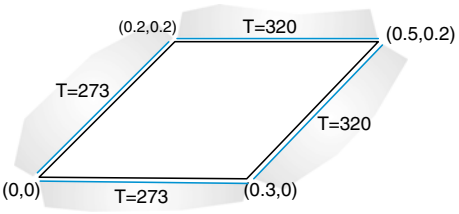


Fig. 8.22 An oblique parallelogram

Exercise 9

a. Use Doxygen [19] to find the list of all overloaded functions under the name fvmLaplacian and then under then name fvcLaplacian.
b. Define in the dictionary file fvSchemes the option to exclude the non orthogonal correction (Gauss linear uncorrected;).
c. Define in the dictionary file fvSchemes the option to use a limited non orthogonal correction (Gauss linear limited 0.8;).
d. List of the possible Laplacian non orthogonal correction type available in OpenFOAM® (Hint: just mistype a scheme, i.e., banana, and launch any solver or application).

References

1. Patankar SV (1980) Numerical heat transfer and fluid flow. Hemisphere Publishing Corporation, McGraw-Hill, USA
2. Jiang T, Przekwas AJ (1994) Implicit, pressure-based incompressible Navier-stokes equations solver for unstructured meshes. AIAA-94-0305
3. Davidson L (1996) A pressure correction method for unstructured meshes with arbitrary control volumes. Int J Numer Meth Fluids 22:265–281
4. Barth TJ (1992) Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-stokes equations. Special course on unstructured grid methods for advection dominated flows. AGARD Report 787
5. Perez-Segarra CD, Farre C, Cadafalch J, Oliva A (2006) Analysis of different numerical schemes for the resolution of convection-diffusion equations using finite-volume methods on three-dimensional unstructured grids, Part I: discretization schemes. Numer Heat Transfer Part B: Fundam 49(4):333–350
6. Demirdzic I, Lilek Z, Peric M (1990) Finite volume method for prediction of fluid flow in arbitrary shaped domains with moving boundary. Int J Numer Meth Fluids 10:771–790
7. Demirdzic I, Lilek Z, Peric M (1993) A collocated finite volume method for predicting flows at all speeds. Int J Numer Meth Fluids 16:1029–1050
8. Warsi ZUA (1993) Fluid dynamics: theoretical and computational approaches. CRC Press, Boca Raton
9. Berend FD, van Wachem GM (2014) Compressive VOF method with skewness correction to capture sharp interfaces on arbitrary meshes. J Comput Phys 279:127–144
10. Jasak H (1996) Error analysis and estimation for the finite volume method with applications to fluid flow. Ph.D. thesis, Imperial College London

11. Balckwell BF, Hogan RE (1993) Numerical solution of axisymmetric heat conduction problems using the finite control volume technique. J Thermophys Heat Transfer 7:462–471
12. Wang S (2002) Solving convection-dominated anisotropic diffusion equations by an exponentially fitted finite volume method. Comput Math Appl 44:1249–1265
13. Jayantha PA, Turner IW (2003) On the use of surface interpolation techniques in generalised finite volume strategies for simulating transport in highly anisotropic porous media. J Comput Appl Math 152:199–216
14. Bertolazzi E, Manzini G (2006) A second-order maximum principle preserving finite volume method for steady convection-diffusion problems. SIAM J Numer Anal 43:2172–2199
15. Domelevo K, Omnes P (2005) A finite volume method for the Laplace equation on almost arbitrary two-dimensional grids. Math Model Numer Anal 39:1203–1249
16. Eymard E, Gallouet T, Herbin R (2004) A finite volume scheme for anisotropic diffusion problems. Comptes Rendus Mathematiques (CR MATH) 339:299–302
17. Darwish M, Moukalled F (2009) A compact procedure for discretization of the anisotropic diffusion operator. Numer Heat Transfer, Part B 55:339–360
18. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org
19. OpenFOAM Doxygen, 2015 Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 9
Gradient Computation

Abstract As was shown in the previous chapter, the discretization of the gradients of ϕ at cell centroids and faces is fundamental to constructing the discretized sets of diffusion equations and, as will be revealed in later chapters, of equations involving the convection term. In addition, the evaluation of gradients is needed for the evaluation of various operators. For example, pressure derivatives are directly needed in the discretized momentum equations, while velocity gradients are required to compute the production term in turbulence models, and the strain rate in non-Newtonian viscosity models. This chapter describes several techniques for evaluating gradients on a general mesh topology. The chapter starts with a description of the techniques for computing the gradient on cartesian structured grids and proceeds with gradient evaluation on unstructured grids. The presented methods follow either the Green-Gauss or the least square approach. Methods to interpolate the gradient to element faces are also presented.

9.1 Computing Gradients in Cartesian Grids

For the one-dimensional problem shown in Fig. 9.1 discretized using a uniform grid, a linear profile assumption for the variation of ϕ between cell centroids results in the following expression for the derivative at cell face e:

$$\left(\frac{d\phi}{dx}\right)_e = \frac{\phi_E - \phi_C}{x_E - x_C} = \frac{\phi_E - \phi_C}{\delta x_e} \tag{9.1}$$

In the same way, the derivative at the cell centroid C (Fig. 9.1) can be written using the values at the two adjacent cells as

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*, Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_9

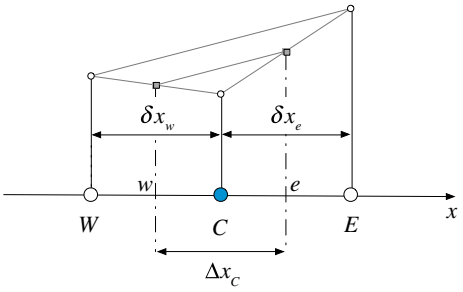


Fig. 9.1 Computing the gradient on a uniform one dimensional grid

$$\left(\frac{d\phi}{dx}\right)_C = \frac{\phi_e - \phi_w}{x_e - x_w} = \frac{\left(\frac{\phi_E+\phi_C}{2}\right) - \left(\frac{\phi_C+\phi_W}{2}\right)}{\Delta x_C} = \frac{\phi_E - \phi_W}{2\Delta x_C} \tag{9.2}$$

This expression is usually referred to as the "central difference" approximation of the first derivative.

For Cartesian grids in multiple dimensions, the derivatives can be computed by applying the same principle along the respective coordinate directions. Consider, for example, the two dimensional grid shown in Fig. 9.2, using the central difference approximation introduced earlier, the partial derivatives in the x and y directions are obtained as

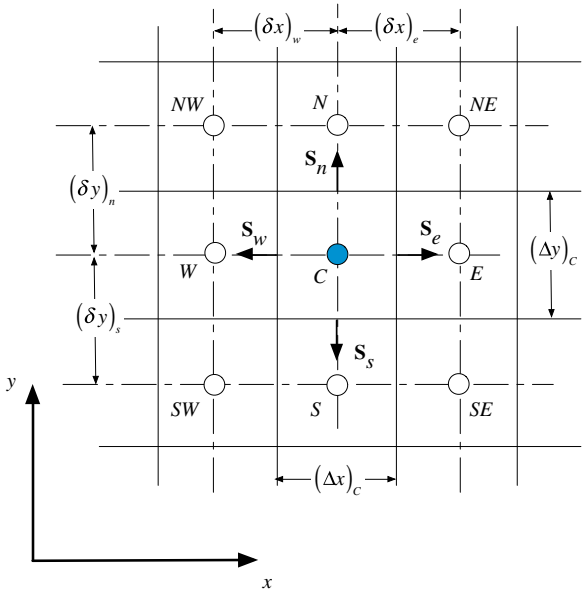


Fig. 9.2 Computing the gradient on a two dimensional Cartesian grid

$$\left(\frac{\partial \phi}{\partial x}\right)_C = \frac{\phi_E - \phi_W}{x_E - x_W} \quad \left(\frac{\partial \phi}{\partial y}\right)_C = \frac{\phi_N - \phi_S}{x_N - x_S} \tag{9.3}$$

A similar equation holds in the z direction for three dimensional grids.

When dealing with unstructured grids the computation of the gradient using the above method becomes unpractical and leads to the use of more general methods, some of these are now presented.

9.2 Green-Gauss Gradient

This is one of the most widely used methods for computing the gradient. It was introduced in the previous chapter and will not be repeated here. Rather the final form of the equation will be given and some additional methods to compute the face values will be introduced.

As derived in the previous chapter, the gradient at the centroid of an element C with volume V_C (Fig. 8.8) is computed as

$$\nabla \phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_f \mathbf{S}_f \tag{9.4}$$

where f refers to a face and $\mathbf{S}_f$ to its surface vector. The face values ϕ_f still need to be defined before the formula can be used. Two approaches are presented for computing ϕ_f. One is face-based with a generally compact stencil involving face neighbors, and the second is vertex-based with a larger stencil involving vertex neighbors. The number of cells in this extended stencil is about twice the compact one.

The use of a compact stencil is attractive with implicit methods because it leads to more compact Jacobian matrices. However, the enlarged stencil brings more information into the reconstruction, and is therefore expected to be more accurate.

Method 1: Compact Stencil [1]

For the two and three dimensional grid system shown in Fig. 9.3a, b, respectively, a simple approximation for calculating ϕ_f is to use the average values of the two cells sharing the face. In this case the value of ϕ_f is calculated as

$$\phi_f = g_C \phi_C + (1 - g_C)\phi_F \tag{9.5}$$

where g_C is a geometric weighting factor equal to

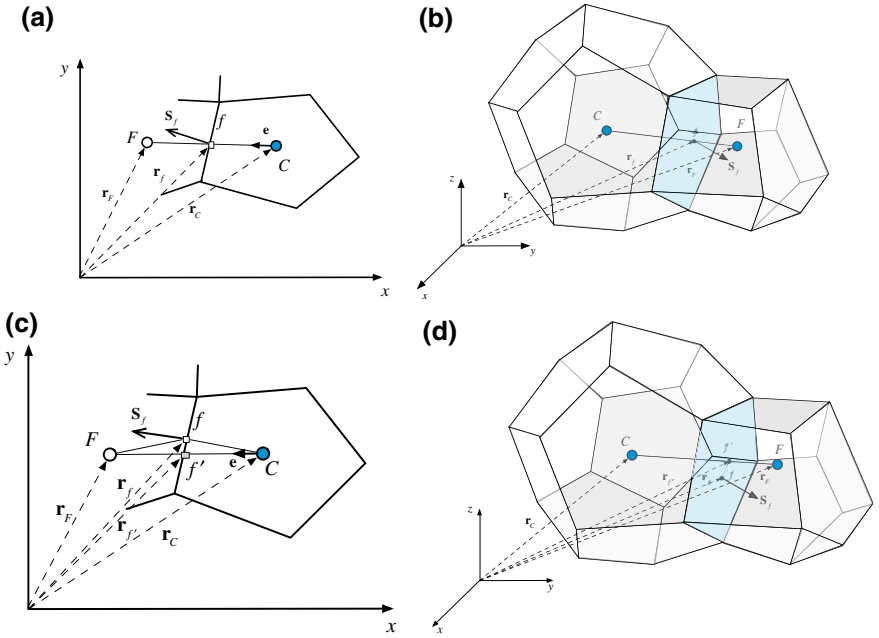


Fig. 9.3 Conjunctional face in **a** a two dimensional and **b** a three dimensional configuration; non-conjunctional face in **c** a two dimensional and **d** a three dimensional configuration

$$g_C = \frac{\| \mathbf{r}_F - \mathbf{r}_f \|}{\| \mathbf{r}_F - \mathbf{r}_C \|} = \frac{d_{Ff}}{d_{FC}} \tag{9.6}$$

where $\mathbf{r}$ designates the position vector and d the distance between two points. When the face is situated halfway between the two cell centers, ϕ_f is found to be

$$\phi_f = \frac{\phi_C + \phi_F}{2} \tag{9.7}$$

This approach is simple to implement in two and three dimensional situations and all operations involved are face-based, not requiring any additional grid connectivity. Accuracy-wise, the above relation leads to a second order approximation of ϕ_f only when the segment $[CF]$ and face S_f intersection point coincides with the centroid of the face S_f, i.e., with the Gaussian integration point f. Thus a second order accurate representation of the gradient is generally not achieved except for the above special case.

Such a condition is not usually satisfied with a general structured non-orthogonal or unstructured grid systems as shown for the two and three dimensional grid systems displayed in Fig. 9.3c, d, respectively. Rather, the skewness of the mesh (non-conjunctionality) results in the segment $[CF]$ and the surface S_f intersecting at

a point f', different from the face centroid f. In this case a correction is needed for the interpolated $\phi_{f'}$ value to get ϕ_f. The correction equation is given by

$$\begin{aligned}\phi_f &= \phi_{f'} + correction \\ &= \phi_{f'} + (\nabla\phi)_{f'} \cdot (\mathbf{r}_f - \mathbf{r}_{f'})\end{aligned} \tag{9.8}$$

which may also be written in expanded form as

$$\begin{aligned}\phi_f &= g_C\{\phi_C + (\nabla\phi)_C \cdot (\mathbf{r}_f - \mathbf{r}_C)\} + (1 - g_C)\{\phi_F + (\nabla\phi)_F \cdot (\mathbf{r}_f - \mathbf{r}_F)\} \\ &= \phi_{f'} + \underbrace{g_C(\nabla\phi)_C \cdot (\mathbf{r}_f - \mathbf{r}_C) + (1 - g_C)(\nabla\phi)_F \cdot (\mathbf{r}_f - \mathbf{r}_F)}_{correction}\end{aligned} \tag{9.9}$$

Since g_C depends on f', Eq. (9.9) indicates that improved estimates of the gradient can be obtained iteratively. At each iteration, the average value at the face is computed using the gradients calculated in the pervious iteration. These face values are then used to compute new estimates of the gradients. Doing excessive number of iterations may cause oscillations and usually not more than two iterations are performed.

In this case, the calculation of g_C requires locating the intersection point f' between $[CF]$ and the face S_f. For that purpose three options will be described.

Option 1: In this option f' is taken to be the exact intersection between $[CF]$ and the face S_f of surface vector $\mathbf{S}_f$. With $\mathbf{n}$ denoting the surface unit vector (i.e., $\mathbf{n} = \mathbf{S}_f/\|\mathbf{S}_f\|$) and $\mathbf{e}$ the unit vector along CF (i.e., $\mathbf{e} = \mathbf{CF}/\|\mathbf{CF}\|$), the location of f' (Fig. 9.3c, d) can be found by exploiting the orthogonality condition that exists between $\mathbf{n}$ and the segment ff' (i.e., $\mathbf{n}$ is normal to the face S_f containing the segment ff') to write

$$(\mathbf{r}_f - \mathbf{r}_{f'}) \cdot \mathbf{n} = 0 \tag{9.10}$$

Further, since f' is a point on CF the vector $\boldsymbol{Cf'}$ can be expressed in terms of $\mathbf{e}$ as

$$\boldsymbol{Cf'} = (\mathbf{r}_{f'} - \mathbf{r}_C) = k\mathbf{e} \tag{9.11}$$

where k is a scalar quantity. Combining Eqs. (9.10) and (9.11) yields

$$\mathbf{r}_{f'} = \frac{\mathbf{r}_f \cdot \mathbf{n}}{\mathbf{e} \cdot \mathbf{n}}\mathbf{e} \tag{9.12}$$

with f' located, g_C is computed as

$$g_C = \frac{\|\mathbf{r}_F - \mathbf{r}_{f'}\|}{\|\mathbf{r}_F - \mathbf{r}_C\|} = \frac{d_{Ff'}}{d_{FC}} \tag{9.13}$$

Then, the calculation procedure involves the following steps:

During the first iteration, calculate the gradient field over the entire domain as follows:

1. Calculate $\phi_{f'}$ using $\phi_{f'} = g_C \phi_C + (1 - g_C)\phi_F$
2. Calculate $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_{f'} \mathbf{S}_f$

From the second iteration onward, correct the gradient field according to the following procedure:

3. Update ϕ_f using $\phi_f = \phi_{f'} + g_C(\nabla\phi)_C \cdot (\mathbf{r}_f - \mathbf{r}_C) + (1 - g_C)(\nabla\phi)_F \cdot (\mathbf{r}_f - \mathbf{r}_F)$
4. Update $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_f \mathbf{S}_f$
5. Go back to step 3 and repeat.

Option 2: For f' chosen to be at the centre of segment $[CF]$ [Fig. 9.4a in two dimensions and Fig. 9.4b in three dimensions] the equations become simpler and the calculation of the gradient field over the domain proceeds as follows:

During the first iteration, calculate the gradient field over the entire domain as follows:

1. Calculate $\phi_{f'}$ using $\phi_{f'} = \frac{\phi_C + \phi_F}{2}$
2. Calculate $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_{f'} \mathbf{S}_f$

From the second iteration onward, correct the gradient field according to the following procedure:

3. Update ϕ_f using $\phi_f = \phi_{f'} + 0.5 * \left[(\nabla\phi)_C + (\nabla\phi)_F\right] \cdot \left[\mathbf{r}_f - 0.5 * (\mathbf{r}_C + \mathbf{r}_F)\right]$
4. Update $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C} \sum_{f \sim nb(C)} \phi_f \mathbf{S}_f$
5. Go back to step 3 and repeat

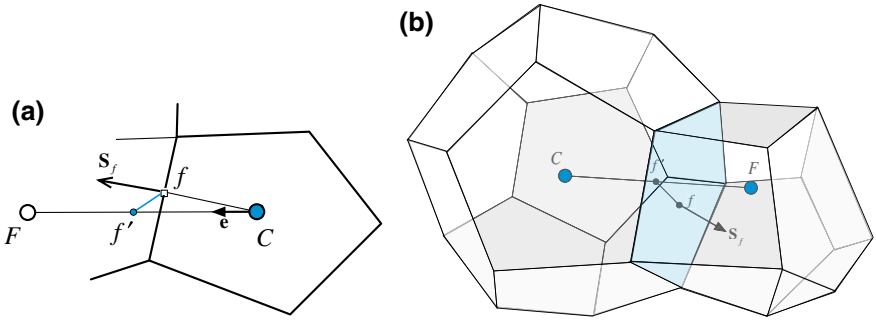


Fig. 9.4 Correction to Non-Conjunctionality using the midpoint approach: **a** two dimensional configuration; **b** three dimensional configuration

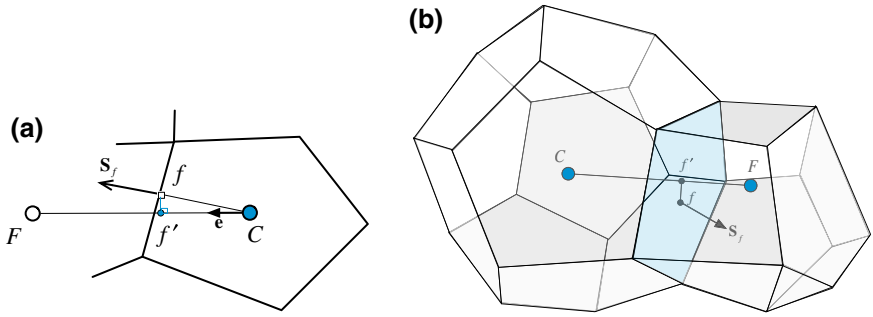


Fig. 9.5 Correction to Non-Conjunctionality using the minimum distance approach: **a** two dimensional configuration; **b** three dimensional configuration

Option 3: The position of f' can be chosen such that the distance ff' is the shortest possible [Fig. 9.5a in two dimensions and Fig. 9.5b in three dimensions], i.e., $[ff']$ perpendicular to $[CF]$. This leads to a more accurate computation of the gradient during the first iteration. In this case f' is computed by minimizing the distance between f and f'. In general $\mathbf{r}_{f'}$ is described by

$$\mathbf{r}_{f'} = \mathbf{r}_C + q(\mathbf{r}_C - \mathbf{r}_F) \tag{9.14}$$

where $0 < q < 1$.

Denoting the distance $f'f$ by d, then its square is obtained as

$$\begin{aligned} d^2 &= (\mathbf{r}_f - \mathbf{r}_{f'}) \cdot (\mathbf{r}_f - \mathbf{r}_{f'}) \\ &= [\mathbf{r}_f - \mathbf{r}_C - q(\mathbf{r}_C - \mathbf{r}_F)] \cdot [\mathbf{r}_f - \mathbf{r}_C - q(\mathbf{r}_C - \mathbf{r}_F)] \\ &= (\mathbf{r}_f - \mathbf{r}_C) \cdot (\mathbf{r}_f - \mathbf{r}_C) - 2q(\mathbf{r}_f - \mathbf{r}_C) \cdot (\mathbf{r}_C - \mathbf{r}_F) + q^2(\mathbf{r}_C - \mathbf{r}_F) \cdot (\mathbf{r}_C - \mathbf{r}_F) \end{aligned} \tag{9.15}$$

Minimizing the function d^2 with respect to q, yields

$$\frac{\partial(d^2)}{\partial q} = 0 \Rightarrow -2(\mathbf{r}_f - \mathbf{r}_C) \cdot (\mathbf{r}_C - \mathbf{r}_F) + 2q(\mathbf{r}_C - \mathbf{r}_F) \cdot (\mathbf{r}_C - \mathbf{r}_F) = 0 \tag{9.16}$$

Solving, q is obtained as

$$q = -\frac{\mathbf{r}_{Cf} \cdot \mathbf{r}_{CF}}{\mathbf{r}_{CF} \cdot \mathbf{r}_{CF}} \tag{9.17}$$

Knowing the q values, the gradient calculation over the domain proceeds as follows:

During the first iteration, calculate the gradient field over the entire domain as follows:

1. Calculate $\mathbf{r}_{f'}$ using $\quad \mathbf{r}_{f'} = \mathbf{r}_C - \dfrac{\mathbf{r}_{Cf} \cdot \mathbf{r}_{CF}}{\mathbf{r}_{CF} \cdot \mathbf{r}_{CF}} (\mathbf{r}_C - \mathbf{r}_F)$
2. Calculate g_C using $\quad g_C = \|\mathbf{r}_F - \mathbf{r}_{f'}\| / \|\mathbf{r}_F - \mathbf{r}_C\|$

3. Calculate $\phi_{f'}$ using $\phi_{f'} = g_C\phi_C + (1 - g_C)\phi_F$
4. Calculate $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C}\sum_{f\sim nb(C)} \phi_{f'}\mathbf{S}_f$

From the second iteration onward, correct the gradient field according to the following procedure:

5. Calculate $\nabla\phi_{f'}$ using $\nabla\phi_{f'} = g_C\nabla\phi_C + (1 - g_C)\nabla\phi_F$
6. Update ϕ_f using $\phi_f = \phi_{f'} + \nabla\phi_{f'} \cdot (\mathbf{r}_f - \mathbf{r}_{f'})$
7. Update $\nabla\phi_C$ using $\nabla\phi_C = \frac{1}{V_C}\sum_{f\sim nb(C)} \phi_f\mathbf{S}_f$
8. Go back to step 5 and repeat

Example 1

For the mesh shown in Fig. 9.6, the coordinates of the grid point C and its neighbors F_1 through F_6 are given by

$$C(13, 11) \quad F_1(4.5, 9.5) \quad F_2(8, 3) \quad F_3(17, 3.5)$$
$$F_4(22, 10) \quad F_5(16, 20) \quad F_6(7, 18)$$

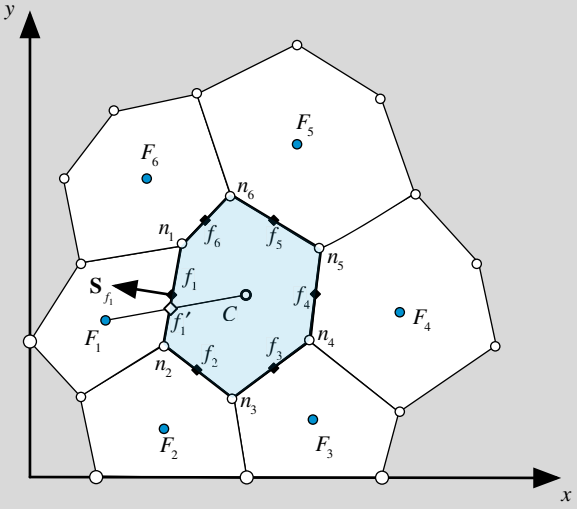


Fig. 9.6 Grid layout for Examples 1 and 2

while the nodes n_1 through n_6 are located at

$$n_1(9, 14) \quad n_2(8, 8) \quad n_3(12, 5)$$
$$n_4(17, 9) \quad n_5(17.5, 14) \quad n_6(12, 17)$$

If the values of the dependent variable ϕ at the centroids are known to be

$$\begin{aligned} &\phi_C = 167 \\ &\phi_{F_1} = 56.75 \quad \phi_{F_2} = 35 \quad \phi_{F_3} = 80 \\ &\phi_{F_4} = 252 \quad \phi_{F_5} = 356 \quad \phi_{F_6} = 151 \end{aligned}$$

and the values of the gradient of ϕ, $(\nabla\phi)$, at all neighboring elements to C are given by

$$\begin{aligned} &\nabla\phi_{F_1} = 10.5\mathbf{i} + 5.5\mathbf{j} \quad \nabla\phi_{F_2} = 4\mathbf{i} + 9\mathbf{j} \quad \nabla\phi_{F_3} = 4.5\mathbf{i} + 18\mathbf{j} \\ &\nabla\phi_{F_4} = 11\mathbf{i} + 23\mathbf{j} \quad \nabla\phi_{F_5} = 21\mathbf{i} + 17\mathbf{j} \quad \nabla\phi_{F_6} = 19\mathbf{i} + 8\mathbf{j} \end{aligned}$$

If the volume of cell C is V_C = 76, find the gradient $\nabla\phi_C$ using

a. *The Green-Gauss method with no correction*
b. *The Green-Gauss method alongside correction to skewness with f′ chosen to be at the centre of segment* [*CF*].

Solution The Green-Gauss method with no correction

a. The interpolation factors are needed to perform the calculations. This, in turn, necessitates computing the coordinates of the face centroids. The coordinates of the centroid f_1 are found as

$$\left.\begin{aligned} x_{f_1} &= 0.5 * (x_{n_1} + x_{n_2}) = 0.5 * (9 + 8) = 8.5 \\ y_{f_1} &= 0.5 * (y_{n_1} + y_{n_2}) = 0.5 * (14 + 8) = 11 \end{aligned}\right\} \Rightarrow f_1(8.5, 11)$$

In a similar way, the coordinates of other face centroids are found to be

$$\begin{aligned} &f_2(10, 6.5) \quad f_3(14.5, 7) \\ &f_4(17.25, 11.5) \quad f_5(14.75, 15.5) \quad f_6(10.5, 15.5) \end{aligned}$$

The surface vectors are calculated as

$$\begin{aligned} &\mathbf{S}_{f_1} = -6\mathbf{i} + \mathbf{j} \quad \mathbf{S}_{f_2} = -3\mathbf{i} - 4\mathbf{j} \quad \mathbf{S}_{f_3} = 4\mathbf{i} - 5\mathbf{j} \\ &\mathbf{S}_{f_4} = 5\mathbf{i} - 0.5\mathbf{j} \quad \mathbf{S}_{f_5} = 3\mathbf{i} + 5.5\mathbf{j} \quad \mathbf{S}_{f_6} = -3\mathbf{i} + 3\mathbf{j} \end{aligned}$$

The interpolation factor $(g_C)_1$ is computed using

$$\left.\begin{aligned}(g_C)_1 &= \frac{F_1 f_1}{F_1 f_1 + Cf_1}\\ F_1 f_1 &= \sqrt{(4.5-8.5)^2+(9.5-11)^2} = 4.272\\ Cf_1 &= \sqrt{(13-8.5)^2+(11-11)^2} = 4.5\end{aligned}\right\} \Rightarrow (g_C)_1 = 0.487$$

In a similar way, the other interpolation factors are found to be

$$(g_C)_2 = 0.427\ (g_C)_3 = 0.502\ (g_C)_4 = 0.538\ (g_C)_5 = 0.492\ (g_C)_6 = 0.455$$

Using Eq. (9.5) the ϕ_f values are computed as

$$\phi_{f_1} = 110.442 \quad \phi_{f_2} = 91.364 \quad \phi_{f_3} = 123.674$$
$$\phi_{f_4} = 206.27 \quad \phi_{f_5} = 263.012 \quad \phi_{f_6} = 158.28$$

Using Eq. (9.4), $\nabla\phi_C$ is calculated as

$$\nabla\phi_C = \frac{1}{76}\left\{\begin{bmatrix}110.442\times(-6)+91.364\times(-3)+123.674\times 4+\\ 206.27\times 5+263.012\times 3+158.28\times(-3)\end{bmatrix}\mathbf{i}+\right.$$
$$\left.\begin{bmatrix}110.442\times(1)+91.364\times(-4)+123.674\times(-5)+\\ 206.27\times(-0.5)+263.012\times 5.5+158.28\times(3)\end{bmatrix}\mathbf{j}\right\}$$
$$= 11.889\mathbf{i} + 12.433\mathbf{j}$$

b. The Green-Gauss method alongside correction to skewness with f' chosen to be at the centre of segment $[CF]$.

The values at the f' locations are computed as half the sum of the values at the nodes straddling the face and are given by

$$\phi_{f_1} = 111.875 \quad \phi_{f_2} = 101 \quad \phi_{f_3} = 123.5 \quad \phi_{f_4} = 209.5 \quad \phi_{f_5} = 261.5 \quad \phi_{f_6} = 158.5$$

Using these values, the first estimate for the gradient is obtained using Eq. (9.4) as

$$\nabla\phi_C = 11.53\mathbf{i} + 11.826\mathbf{j}$$

Defining $\mathbf{d_f} = \mathbf{r_f} - 0.5 \times (\mathbf{r_C} + \mathbf{r_F})$, the various values are found to be

$$\mathbf{d}_{f_1} = -0.25\mathbf{i} + 0.75\mathbf{j} \quad \mathbf{d}_{f_2} = -0.5\mathbf{i} - 0.5\mathbf{j} \quad \mathbf{d}_{f_3} = -0.5\mathbf{i} - 0.25\mathbf{j}$$
$$\mathbf{d}_{f_4} = -0.25\mathbf{i} + \mathbf{j} \quad \mathbf{d}_{f_5} = 0.25\mathbf{i} \quad \mathbf{d}_{f_6} = 0.5\mathbf{i} + \mathbf{j}$$

Using $\phi_f = \phi_{f'} + 0.5 * [(\nabla\phi)_C + (\nabla\phi)_F] \cdot [\mathbf{r}_f - 0.5 * (\mathbf{r}_C + \mathbf{r}_F)]$ the updated values at the faces are obtained as

$$\phi_{f_1} = 115.619 \quad \phi_{f_2} = 91.911 \quad \phi_{f_3} = 115.764$$
$$\phi_{f_4} = 224.097 \quad \phi_{f_5} = 265.566 \quad \phi_{f_6} = 176.046$$

These values are used in Eq. (9.4) yielding the updated value of the gradient as

$$\nabla\phi_C = 11.614\mathbf{i} + 13.761\mathbf{j}$$

Method 2: Extended Stencil [2]

The value of ϕ_f at the surface centroid f can be computed as the mean of the values at the vertices defining the surface. This necessitates the estimation of the properties at the vertices. The properties at a vertex node are calculated using the weighted average of the properties within the cells surrounding that node. Figure 9.7 shows the cells that are considered for the weighted average of the properties at the vertex nodes n_1 and n_2. The weight is taken as the inverse of the distance of the vertex from the cell centre [3]. The resulting equation for the properties at the vertices are written as,

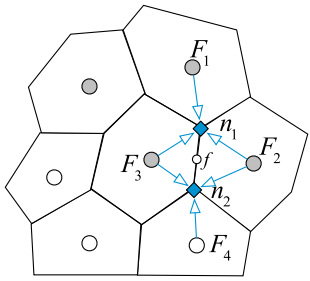


Fig. 9.7 Cells Contributing to node n for the Weighted Average

$$\phi_n = \frac{\sum_{k=1}^{NB(n)} \frac{\phi_{F_k}}{\|\mathbf{r}_n - \mathbf{r}_{F_k}\|}}{\sum_{k=1}^{NB(n)} \frac{1}{\|\mathbf{r}_n - \mathbf{r}_{F_k}\|}} \tag{9.18}$$

where n refers to the vertex node, F_k to the neighboring cell node, $NB(n)$ the total number of cell nodes surrounding the vertex node n, and $\|\mathbf{r}_n - \mathbf{r}_{F_k}\|$ the distance from the vertex node to the centroid of the neighboring cells.

Once the values ϕ_n at the vertices are found, the values ϕ_f at the surface centroids are calculated followed by the gradients at the control volume centroids. In two dimensional situations, ϕ_f is computed as

$$\phi_f = \frac{\phi_{n_1} + \phi_{n_2}}{2} \tag{9.19}$$

Then the gradient at C is found using

$$\nabla\phi_C = \frac{1}{V_C} \sum_{f=nb(C)} \phi_f \mathbf{S}_f = \frac{1}{V_C} \sum_{f \sim nb(C)} \left(\frac{\phi_{n_1} + \phi_{n_2}}{2} \right)_f \mathbf{S}_f \tag{9.20}$$

In three dimensional situations, the calculations are a little more involved as the number of a face vertices depends on the element type. The value of ϕ_f is found from the values at the vertices using

$$\phi_f = \frac{\sum_{k=1}^{nb(f)} \frac{\phi_{n_k}}{\|\mathbf{r}_{n_k} - \mathbf{r}_f\|}}{\sum_{k=1}^{nb(f)} \frac{1}{\|\mathbf{r}_{n_k} - \mathbf{r}_f\|}} \tag{9.21}$$

where n represents the number of vertices of face f. Once the values ϕ_f are calculated, the gradient at C is computed using Eq. (9.4).

One of the disadvantages of this approach is that information from the wrong side of the cell face also contributes to the weighted average values of the conserved variables. This can be overcome by using upwind biased gradients as discussed by Cabello [4]. The higher order calculations based on the upwind biased gradients, however, have both higher memory overheads required to store the information about the cells used for the upwind biased gradient calculation and increased coding complexity.

Example 2
Using the data of example I, calculate ϕ_{f_1} *using the extended stencil approach* via *Eq.* (9.16).

Solution First the distances have to be calculated and are given by

$$\|\mathbf{r}_{n_1} - \mathbf{r}_{F_6}\| = \sqrt{(9-7)^2+(14-18)^2} = 4.472$$
$$\|\mathbf{r}_{n_1} - \mathbf{r}_{F_1}\| = \sqrt{(9-4.5)^2+(14-9.5)^2} = 6.364$$
$$\|\mathbf{r}_{n_1} - \mathbf{r}_{C}\| = \sqrt{(9-13)^2+(14-11)^2} = 5$$
$$\|\mathbf{r}_{n_2} - \mathbf{r}_{F_1}\| = \sqrt{(8-4.5)^2+(8-9.5)^2} = 3.808$$
$$\|\mathbf{r}_{n_2} - \mathbf{r}_{F_2}\| = \sqrt{(8-8)^2+(8-3)^2} = 5$$
$$\|\mathbf{r}_{n_2} - \mathbf{r}_{C}\| = \sqrt{(8-13)^2+(8-11)^2} = 5.831$$

The values at nodes n_1 and n_2 are computed using Eq. (9.15) as

$$\phi_{n_1} = \frac{\frac{151}{4.472}+\frac{56.75}{6.364}+\frac{167}{5}}{\frac{1}{4.472}+\frac{1}{6.364}+\frac{1}{5}} = 131.009 \quad \phi_{n_2} = \frac{\frac{56.75}{3.808}+\frac{35}{5}+\frac{167}{5.831}}{\frac{1}{3.808}+\frac{1}{5}+\frac{1}{5.831}} = 79.708$$

The value of ϕ_{f_1} is found to be

$$\phi_{f_1} = 0.5(131.009 + 79.708) = 105.3585$$

9.3 Least-Square Gradient

Using least-square methods to compute the gradients [5] offers more flexibility with regard to the order of accuracy achieved [6] and the stencil used [7]. In the least-square method, the divergence-based gradient can be recovered as a special case. This flexibility comes at a cost, as proper weighting is needed for the stencil terms, and computation of the weights adds to the computational cost. The method is described next.

Considering a control volume and its immediate neighbors (Fig. 9.8), the change in centroid values between C and F is given by $(\phi_F - \phi_C)$, if the cell gradient $(\nabla\phi_C)$ is exact, then the difference can also be computed as

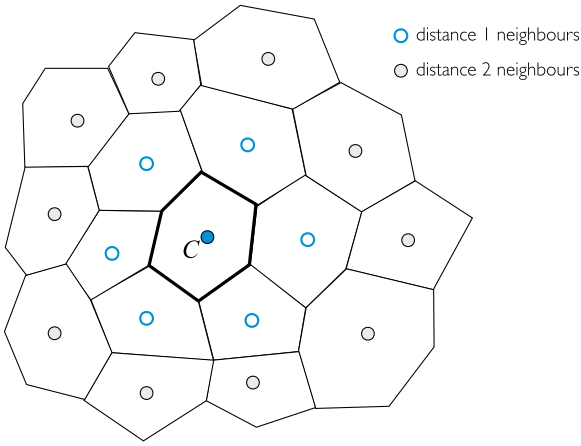


Fig. 9.8 A control volume with its immediate neighbors

$$\phi_F = \phi_C + (\nabla\phi)_C \cdot \underbrace{(\mathbf{r}_F - \mathbf{r}_C)}_{\mathbf{r}_{CF}} \tag{9.22}$$

However unless the solution field is linear the cell gradient cannot be exact because C has more neighbors than the gradient vector has components. In the least square methods, a gradient is computed by an optimization procedure that finds the minimum of the function G_C defined as

$$\begin{aligned} G_C &= \sum_{k=1}^{NB(C)} \left\{ w_k \left[\phi_{F_k} - (\phi_C + \nabla\phi_C \cdot \mathbf{r}_{CF_k}) \right]^2 \right\} \\ &= \sum_{k=1}^{NB(C)} \left\{ w_k \left[\Delta\phi_k - \left(\Delta x_k \left(\frac{\partial\phi}{\partial x} \right)_C + \Delta y_k \left(\frac{\partial\phi}{\partial y} \right)_C + \Delta z_k \left(\frac{\partial\phi}{\partial z} \right)_C \right) \right]^2 \right\} \end{aligned} \tag{9.23}$$

where w_k is some weighting factor. The various terms in the above equation represent

$$\begin{aligned} \Delta\phi_k &= \phi_{F_k} - \phi_C \\ \Delta x_k &= \mathbf{r}_{CF_k} \cdot \mathbf{i} \\ \Delta y_k &= \mathbf{r}_{CF_k} \cdot \mathbf{j} \\ \Delta z_k &= \mathbf{r}_{CF_k} \cdot \mathbf{k} \end{aligned} \tag{9.24}$$

The function G_C is minimized by enforcing the conditions

$$\frac{\partial G_C}{\partial\left(\frac{\partial\phi}{\partial x}\right)} = \frac{\partial G_C}{\partial\left(\frac{\partial\phi}{\partial y}\right)} = \frac{\partial G_C}{\partial\left(\frac{\partial\phi}{\partial z}\right)} = 0 \tag{9.25}$$

to yield the following set of three equations in three unknowns:

$$\begin{aligned}
&\sum_{k=1}^{NB(C)} \left\{ 2w_k \Delta x_k \left[-\Delta\phi_k + \Delta x_k \left(\frac{\partial\phi}{\partial x}\right)_C + \Delta y_k \left(\frac{\partial\phi}{\partial y}\right)_C + \Delta z_k \left(\frac{\partial\phi}{\partial z}\right)_C \right] \right\} = 0 \\
&\sum_{k=1}^{NB(C)} \left\{ 2w_k \Delta y_k \left[-\Delta\phi_k + \Delta x_k \left(\frac{\partial\phi}{\partial x}\right)_C + \Delta y_k \left(\frac{\partial\phi}{\partial y}\right)_C + \Delta z_k \left(\frac{\partial\phi}{\partial z}\right)_C \right] \right\} = 0 \\
&\sum_{k=1}^{NB(C)} \left\{ 2w_k \Delta z_k \left[-\Delta\phi_k + \Delta x_k \left(\frac{\partial\phi}{\partial x}\right)_C + \Delta y_k \left(\frac{\partial\phi}{\partial y}\right)_C + \Delta z_k \left(\frac{\partial\phi}{\partial z}\right)_C \right] \right\} = 0
\end{aligned} \tag{9.26}$$

which can be written in matrix form as

$$\begin{aligned}
&\begin{bmatrix}
\sum_{k=1}^{NB(C)} w_k \Delta x_k \Delta x_k & \sum_{k=1}^{NB(C)} w_k \Delta x_k \Delta y_k & \sum_{k=1}^{NB(C)} w_k \Delta x_k \Delta z_k \\
\sum_{k=1}^{NB(C)} w_k \Delta y_k \Delta x_k & \sum_{k=1}^{NB(C)} w_k \Delta y_k \Delta y_k & \sum_{k=1}^{NB(C)} w_k \Delta y_k \Delta z_k \\
\sum_{k=1}^{NB(C)} w_k \Delta z_k \Delta x_k & \sum_{k=1}^{NB(C)} w_k \Delta z_k \Delta y_k & \sum_{k=1}^{NB(C)} w_k \Delta z_k \Delta z_k
\end{bmatrix}
\begin{bmatrix}
\left(\frac{\partial\phi}{\partial x}\right)_C \\
\left(\frac{\partial\phi}{\partial y}\right)_C \\
\left(\frac{\partial\phi}{\partial z}\right)_C
\end{bmatrix} \\
&= \begin{bmatrix}
\sum_{k=1}^{NB(C)} w_k \Delta x_k \Delta\phi_k \\
\sum_{k=1}^{NB(C)} w_k \Delta y_k \Delta\phi_k \\
\sum_{k=1}^{NB(C)} w_k \Delta z_k \Delta\phi_k
\end{bmatrix}
\end{aligned} \tag{9.27}$$

The solution to the above set of equations yields the gradient $(\nabla\phi)_C$. A solution exists provided that the matrix on the left hand side is not singular. Moreover, the choice of the w_k is important in determining the properties of the gradient. For example if w_k is chosen to be 1 for all neighbors of C, then all neighboring points will have the same weight in the computation of the gradient irrespective of whether they are near or far from point C. Actually points that are farther from C will have a more important influence as the error function will be more affected by their error.

Another choice for w_k, which was used earlier with the extended stencil method, is the inverse distance between C and F given by

$$w_k = \frac{1}{|\mathbf{r}_{F_k} - \mathbf{r}_C|} = \frac{1}{\sqrt{\Delta x_{F_k}^2 + \Delta y_{F_k}^2 + \Delta z_{F_k}^2}} \tag{9.28}$$

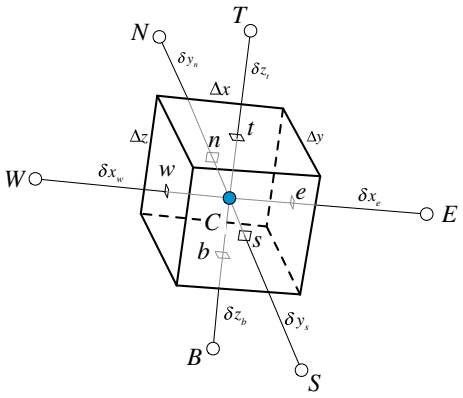


Fig. 9.9 A three dimensional Cartesian control volume

Other options that can be pursued include the inverse distance raised to any power n such that

$$w_k = \frac{1}{\left|\mathbf{r}_{F_k} - \mathbf{r}_C\right|^n} \tag{9.29}$$

where n can be set to 1, 2, 3, etc.

As mentioned above, the divergence based gradient is a special case of the least-square formulation. This can be shown for a Cartesian grid (see Fig. 9.9) where substituting the geometric quantities into Eq. (9.27) would yield the following set of three equations.

$$\begin{bmatrix} x_E - x_W & 0 & 0 \\ 0 & y_N - y_S & 0 \\ 0 & 0 & z_T - z_B \end{bmatrix} \begin{bmatrix} (\partial\phi/\partial x)_C \\ (\partial\phi/\partial y)_C \\ (\partial\phi/\partial z)_C \end{bmatrix} = \begin{bmatrix} \phi_E - \phi_W \\ \phi_N - \phi_S \\ \phi_T - \phi_B \end{bmatrix} \tag{9.30}$$

Solving the above equation, the derivatives in the various directions are found to be

$$\left(\frac{\partial\phi}{\partial x}\right)_C = \frac{\phi_E - \phi_W}{x_E - x_W} \quad \left(\frac{\partial\phi}{\partial y}\right)_C = \frac{\phi_N - \phi_S}{y_N - y_S} \quad \left(\frac{\partial\phi}{\partial z}\right)_C = \frac{\phi_T - \phi_B}{z_T - z_B} \tag{9.31}$$

The obtained values are exactly the ones given in Eq. (9.3), demonstrating that the divergence-based gradient is a special case of the least-square method. Finally it can easily be demonstrated that the accuracy of the resulting gradient is at least first order. Indeed, the Taylor series expansion of the ϕ value around node C can be written as

$$\phi(\mathbf{r}) - \phi(\mathbf{r}_C) = (\nabla\phi)_C \cdot (\mathbf{r} - \mathbf{r}_C) + O\left(\mathbf{r}^2\right) \tag{9.32}$$

which when solved for $(\nabla\phi)_C$ results in $O(\mathbf{r})$.

9.4 Interpolating Gradients to Faces

It was shown in Chap. 8 that the discretization of the diffusion term in non-orthogonal grids requires the use of correction terms involving gradients at control volume faces. Thus in this situation the gradients need to be interpolated from the control volume centroids where they were computed to the control volume faces where they will be used. Figure 9.10a shows the stencil used in the Gauss gradient computation for two neighboring control volumes. The gradient at the face will have exactly the same stencil, while ideally it should be similar to that of Fig. 9.10b.

A better insight is gained by considering the configuration in Fig. 9.11a, which shows the gradients $\nabla\phi_C$ and $\nabla\phi_F$ of the variable ϕ at the two nodes C and F, respectively. The interpolated gradient at the face, $\overline{\nabla\phi}_f$, is obtained by averaging the values at nodes C and F, as shown in Fig. 9.11b. It is important for the stencil of the gradient at the face to be heavily based on the nodes straddling the face, which is not guaranteed by this simple averaging practice. As schematically displayed in Fig. 9.11c, this can be accomplished by forcing the face gradient along the CF direction to be equal to the local gradient defined by the values of ϕ at C and F. Mathematically this can be written as

$$\nabla\phi_f = \overline{\nabla\phi}_f + \underbrace{\left[\frac{\phi_F - \phi_C}{d_{CF}} - \left(\overline{\nabla\phi}_f \cdot \mathbf{e}_{CF}\right)\right]\mathbf{e}_{CF}}_{\textit{Correction interpolated face gradient}} \tag{9.33}$$

where

$$\overline{\nabla\phi}_f = g_C\nabla\phi_C + g_F\nabla\phi_F, \quad \mathbf{e}_{CF} = \frac{\mathbf{d}_{CF}}{d_{CF}}, \quad \mathbf{d}_{CF} = \mathbf{r}_F - \mathbf{r}_C \tag{9.34}$$

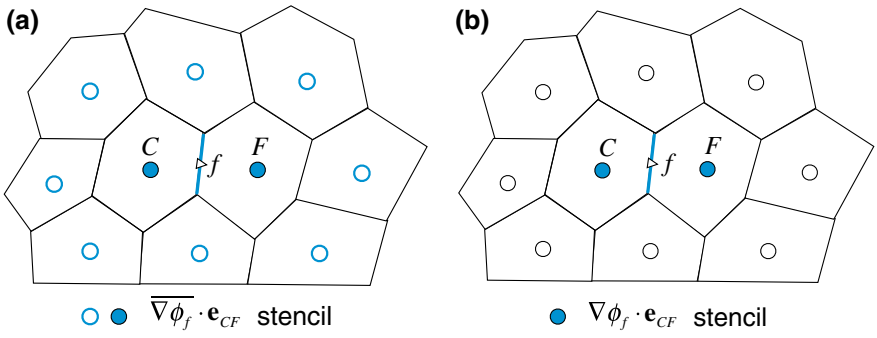


Fig. 9.10 **a** Interpolated and **b** corrected gradient at a control volume face

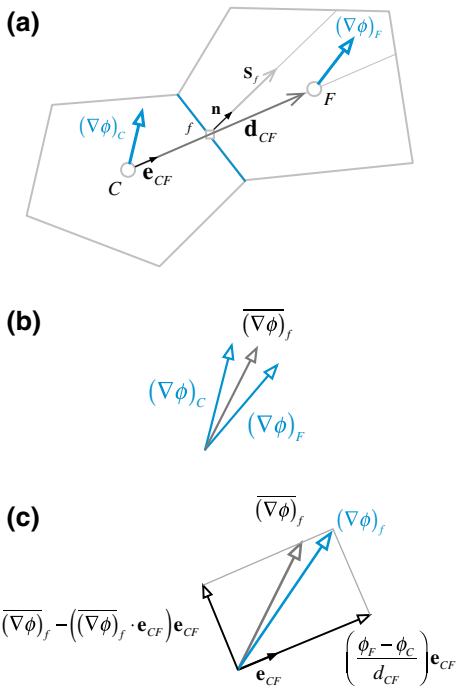


Fig. 9.11 **a** Schematics of the gradients at the two nodes C and F straddling face f; **b** computing $\overline{\nabla\phi}_f$ as a simple average of $\nabla\phi_C$ and $\nabla\phi_F$ using Eq. (9.31); **c** the gradient at the face with its value heavily based on the nodes straddling the face

where $\mathbf{r}_F$ and $\mathbf{r}_C$ are position vectors, as displayed in Fig. 9.3. This approach is applicable to both structured and unstructured grids. For unstructured grids, while the stencil used for the face gradient might not be decreased, the gradient across a face will still be based on the nodes straddling that face.

9.5 Computational Pointers

9.5.1 uFVM

In uFVM, the functions cfdComputeGradientGauss0 and cfdComputeGradientNodal are used to compute the element gradients. The Gauss gradient routine (Listing 9.1) takes as input the phi element array and returns the element gradient array. The values are interpolated to the faces using a weighted factor with no correction for non-conjunctionality, hence the 0 digit in the function name.

```
function phiGrad = cfdComputeGradientGauss0(phi,theMesh)
%=================================================
%  written by the CFD Group @ AUB, Fall 2006
%=================================================
%
if(nargin<2)
    theMesh = cfdGetMesh;
end

theSize = size(phi);
theNumberOfComponents = theSize(2);
if(theNumberOfComponents > 3)
   echo('********* ERROR **********');
   exit;
end
%-------------------------------------------------------
% INTERIOR FACES contribution to gradient
%-------------------------------------------------------
iFaces = 1:theMesh.numberOfInteriorFaces;
iBFaces = theMesh.numberOfInteriorFaces+1:theMesh.numberOfFaces;
iElements = 1:theMesh.numberOfElements;
iBElements        =        theMesh.numberOfElements+1:theMesh.numberOfElements
+theMesh.numberOfBFaces;

iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';

Sf = [theMesh.faces(iFaces).Sf]';
gf = [theMesh.faces(iFaces).gf]';
%-------------------------------------------------------
% Initialize phiGrad Array
%-------------------------------------------------------

phiGrad       =       zeros(theMesh.numberOfElements+theMesh.numberOfBElements,
3,theNumberOfComponents);

for iComponent=1:theNumberOfComponents
    phi_f = gf.*phi(iNeighbours,iComponent) + (1-gf).*phi(iOwners,iComponent);
    %
    for iFace=iFaces
        phiGrad(iOwners(iFace),:,iComponent)                  =
phiGrad(iOwners(iFace),:,iComponent)     + phi_f(iFace)*Sf(iFace,:);
        phiGrad(iNeighbours(iFace),:,iComponent)     =
phiGrad(iNeighbours(iFace),:,iComponent) - phi_f(iFace)*Sf(iFace,:);
    end
end

%-------------------------------------------------------
% BOUNDARY FACES contribution to gradient
%-------------------------------------------------------
iBOwners = [theMesh.faces(iBFaces).iOwner]';
phi_b = phi(iBElements,iComponent);
Sb = [theMesh.faces(iBFaces).Sf]';
for iComponent=1:theNumberOfComponents
    %
    for k=1:theMesh.numberOfBFaces
        phiGrad(iBOwners(k),:,iComponent)                   =
phiGrad(iBOwners(k),:,iComponent)     + phi_b(k)*Sb(k,:);
    end
end
```

Listing 9.1 Function used in uFVM to compute the gradient field at the centroids of elements following the Green-Gauss approach with phi values at the face computed using simple a weighted average interpolation technique with no correction to non-conjunctionality

```
%-----------------------------------------------------
% Get Average Gradient by dividing with element volume
%-----------------------------------------------------
volumes = [theMesh.elements(iElements).volume]';
for iComponent=1:theNumberOfComponents
    for iElement =1:theMesh.numberOfElements
            phiGrad(iElement,:,iComponent)  =  phiGrad(iElement,:,iComponent)/
volumes(iElement);
    end
end
%-----------------------------------------------------
% Set boundary Gradient equal to associated element
% Gradient
%-----------------------------------------------------
phiGrad(iBElements,:,:) = phiGrad(iBOwners,:,:);


end
```

Listing 9.1 (continued)

The nodal gradient routine, listed below (Listing 9.2), is very similar to the Gauss gradient routine except that it uses the functions cfdInterpolateFrom-ElementsToNodes and cfdInterpolateFromNodesToFaces to interpolate phi values to the face that are subsequently used in the Gauss algorithm.

```
%
%=====================================================
% INTERIOR Gradients
%=====================================================
theNumberOfElements = theMesh.numberOfElements;
theNumberOfBElements = theMesh.numberOfBElements;
theNumberOfInteriorFaces = theMesh.numberOfInteriorFaces;
%
phiNodes = cfdInterpolateFromElementsToNodes(phi);
phi_f = cfdInterpolateFromNodesToFaces(phiNodes);
%
phiGrad = zeros(3,theNumberOfElements+theNumberOfBElements);
%-----------------------------------------------------
% INTERIOR FACES contribution to gradient
%-----------------------------------------------------
fvmFaces = theMesh.faces;

% interpolate phi to faces
%
for iFace=1:theNumberOfInteriorFaces
   %
   theFace = fvmFaces(iFace);
   %
   iElement1 = theFace.iOwner;
   iElement2 = theFace.iNeighbour;
   %
```

Listing 9.2 Function used in uFVM to compute the gradient field at the centroids of elements following the Green-Gauss approach with phi values at the face computed using nodal values, i.e., the extended stencil approach

```
   Sf = theFace.Sf;
   %
   %
   phiGrad(:,iElement1) = phiGrad(:,iElement1) + phi_f(iFace)*Sf;
   phiGrad(:,iElement2) = phiGrad(:,iElement2) - phi_f(iFace)*Sf;

end


%======================================================
% BOUNDARY FACES contribution to gradient
%======================================================
for iBPatch=1:theNumberOfBElements
    %
    iBFace = theNumberOfInteriorFaces+iBPatch;
    iBElement = theNumberOfElements+iBPatch;
    theFace = fvmFaces(iBFace);
    %
    iElement1 = theFace.iOwner;
    %
    Sb = theFace.Sf;
    phi_b = phi(iBElement);
    %
    phiGrad(:,iElement1) = phiGrad(:,iElement1) + phi_b*Sf;

end
%------------------------------------------------------
% Get Average Gradient by dividing with element volume
%------------------------------------------------------
for iElement =1:theNumberOfElements
   theElement = fvmElements(iElement);
   phiGrad(:,iElement) = phiGrad(:,iElement)/theElement.volume;
end


%------------------------------------------------------
% Set boundary Gradient equal to associated element
% Gradient
%------------------------------------------------------
for iBPatch = 1:theNumberOfBElements
    iBElement = iBPatch+theNumberOfElements;
    iBFace = iBPatch+theNumberOfInteriorFaces;
    theBFace = fvmFaces(iBFace);
    iOwner = theBFace.iOwner;
    phiGrad(:,iBElement) = phiGrad(:,iOwner);
end
```

Listing 9.2 (continued)

For face interpolation of gradients, a variety of interpolation options are allowed in the function **cfdInterpolateGradientsFromElementsToInteriorFaces**. The input to the function is **theInterpolationScheme**, the element gradient **grad**, the element array **phi** and **mdot** for cases where an upwind or downwind scheme is used. The function is shown in Listing 9.3.

```
function grad_f=
cfdInterpolateGradientsFromElementsToInteriorFaces(theInterpolationScheme,grad
,phi,mdot_f)
%=================================================

%  written by the CFD Group @ AUB, Fall 2006
%=================================================

theMesh= cfdGetMesh;

numberOfInteriorFaces = theMesh.numberOfInteriorFaces;

iOwners = [theMesh.faces(1:numberOfInteriorFaces).iOwner]';
iNeighbours = [theMesh.faces(1:numberOfInteriorFaces).iNeighbour]';
gf = [theMesh.faces(1:numberOfInteriorFaces).gf]';

if(strcmp(theInterpolationScheme,'Average')==1)
    grad_f(:,1) = (1-gf).*grad(iNeighbours,1) + gf.*grad(iOwners,1);
    grad_f(:,2) = (1-gf).*grad(iNeighbours,2) + gf.*grad(iOwners,2);
    grad_f(:,3) = (1-gf).*grad(iNeighbours,3) + gf.*grad(iOwners,3);

elseif(strcmp(theInterpolationScheme,'Upwind')==1)
     pos = zeros(size(mdot_f));
     pos((mdot_f>0))=1;
     %
     grad_f(:,1) = pos.*grad(iNeighbours,1) + (1-pos).*grad(iOwners,1);
     grad_f(:,2) = pos.*grad(iNeighbours,2) + (1-pos).*grad(iOwners,2);
     grad_f(:,3) = pos.*grad(iNeighbours,3) + (1-pos).*grad(iOwners,3);
elseif(strcmp(theInterpolationScheme,'Downwind')==1)
     pos = zeros(size(mdot_f));
     pos((mdot_f>0))=1;
     %
     grad_f(:,1) = (1-pos).*grad(iNeighbours,1) + pos.*grad(iOwners,1);
     grad_f(:,2) = (1-pos).*grad(iNeighbours,2) + pos.*grad(iOwners,2);
     grad_f(:,3) = (1-pos).*grad(iNeighbours,3) + pos.*grad(iOwners,3);

elseif(strcmp(theInterpolationScheme,'Average:Corrected')==1)
    grad_f(:,1) = (1-gf).*grad(iNeighbours,1) + gf.*grad(iOwners,1);
    grad_f(:,2) = (1-gf).*grad(iNeighbours,2) + gf.*grad(iOwners,2);
    grad_f(:,3) = (1-gf).*grad(iNeighbours,3) + gf.*grad(iOwners,3);

    d_CF     =     [theMesh.elements(iNeighbours).centroid]'     -
[theMesh.elements(iOwners).centroid]';
    dmag=cfdMagnitude(d_CF);

    e_CF(:,1)=d_CF(:,1)./dmag;
    e_CF(:,2)=d_CF(:,2)./dmag;
    e_CF(:,3)=d_CF(:,3)./dmag;

    local_grad_mag_f = (phi(iNeighbours)-phi(iOwners))./dmag;
    local_grad(:,1) = local_grad_mag_f.*e_CF(:,1);
    local_grad(:,2) = local_grad_mag_f.*e_CF(:,2);
    local_grad(:,3) = local_grad_mag_f.*e_CF(:,3);

    local_avg_grad_mag = dot(grad_f',e_CF')';
    local_avg_grad(:,1)=local_avg_grad_mag.*e_CF(:,1);
    local_avg_grad(:,2)=local_avg_grad_mag.*e_CF(:,2);
    local_avg_grad(:,3)=local_avg_grad_mag.*e_CF(:,3);

    grad_f = grad_f - local_avg_grad + local_grad;
else
    theInterpolationScheme
    grad,
    phi,mdot
    exit;
end
```

Listing 9.3 Function used to interpolation the element gradient field to the faces

It should be noted that **theInterpolationScheme** "*Average:Corrected*" implements the face gradient correction technique used to get a more accurate gradient representation along the *CF* direction, i.e., Eq. (9.33).

9.5.2 OpenFOAM®

In OpenFOAM® [8] several types of gradient evaluation techniques are defined: the standard Green-Gauss method, the second order least square method, and the fourth order least square method. Attention will be focussed here on the Green-Gauss method. Nonetheless the discretization in all cases is performed explicitly and is thus part of the fvc operator namespace. The Green-Gauss gradient is defined at the cell centre, as in Eq. (9.4), and its source code in OpenFOAM is located in the directory "src/finiteVolume/finiteVolume/gradSchemes/gaussGrad".

Implementation-wise, the gradient evaluation is performed in the following two steps:

- Face interpolation of the variable
- Green-Gauss formula evaluation

The interpolation to the face is in the **calcGrad** routine listed below (Listing 9.4).

```
Foam::fv::gaussGrad<Type>::calcGrad
(
    const GeometricField<Type, fvPatchField, volMesh>& vsf,
    const word& name
) const
{
    typedef typename outerProduct<vector, Type>::type GradType;
    tmp<GeometricField<GradType, fvPatchField, volMesh> > tgGrad
    (

        gradf(tinterpScheme_().interpolate(vsf), name)
    );
    GeometricField<GradType, fvPatchField, volMesh>& gGrad = tgGrad();
    correctBoundaryConditions(vsf, gGrad);
    return tgGrad;
}
```

Listing 9.4 Script to interpolate variable values to cell faces

As a first step, values are interpolated to the faces and stored in the generic field "vsf" (the interpolation class will be described with more details in Chap. 11). Then the gradient is computed based on the Green-Gauss formula in the gradf routine shown in Listing 9.5.

```
Foam::fv::gaussGrad<Type>::gradf
(
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf,
    const word& name
)
{
//     Info << "Calculating gradient for " << name  << endl;
    typedef typename outerProduct<vector, Type>::type GradType;

    const fvMesh& mesh = ssf.mesh();
    tmp<GeometricField<GradType, fvPatchField, volMesh> > tgGrad
    (
        new GeometricField<GradType, fvPatchField, volMesh>
        (
            IOobject
            (
                name,
                ssf.instance(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensioned<GradType>
            (
                "0",
                ssf.dimensions()/dimLength,
                pTraits<GradType>::zero
            ),
            zeroGradientFvPatchField<GradType>::typeName
        )
    );
    GeometricField<GradType, fvPatchField, volMesh>& gGrad = tgGrad();

    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();
    const vectorField& Sf = mesh.Sf();

    Field<GradType>& igGrad = gGrad;
    const Field<Type>& issf = ssf;

    forAll(owner, facei)
    {
        GradType Sfssf = Sf[facei]*issf[facei];
        igGrad[owner[facei]] += Sfssf;
        igGrad[neighbour[facei]] -= Sfssf;
    }

    forAll(mesh.boundary(), patchi)
    {
        const labelUList& pFaceCells =
            mesh.boundary()[patchi].faceCells();
        const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
        const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];
        forAll(mesh.boundary()[patchi], facei)
        {
            igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
        }
    }
    igGrad /= mesh.V();
    gGrad.correctBoundaryConditions();
    return tgGrad;
}
```

Listing 9.5 Routine used to compute the gradient using the Green-Gauss method

The sum over cell faces is performed using the LDU addressing. As such the "for" loop that evaluates the sum over the faces of the cell is based only on the global face numbering and uses the upper and lower addressing vectors to add or subtract (Listing 9.6) the flux to cell values. After all fluxes have been processed, the net value is divided by the volume to yield the gradient, as in Eq. 9.4.

```
        igGrad[owner[facei]] += Sfssf;
        igGrad[neighbour[facei]] -= Sfssf;
```

Listing 9.6 Adding or subtracting fluxes to cell values

The type of gradient is defined in fvSchemes shown in Listing 9.7.

```
gradSchemes
{
    default         none;
    grad(phi)   Gauss;
}
```

Listing 9.7 Defining the gradient calculation method

The face interpolation scheme is defined as displayed in Listing 9.8.

```
interpolationSchemes
{
    interpolate(phi) linear;
}
```

Listing 9.8 Defining the interpolation method used in calculating face values

The idea of evaluating the gradient based on a generic interpolation scheme that has to be defined by dictionary, allows computing the gradient with the Green-Gauss formula in several ways by just changing the interpolation scheme.

If skew correction, as defined in Eq. (9.8), is required then the above interpolation scheme definition should be replaced by (Listing 9.9)

```
interpolationSchemes
{
    interpolate(phi) skewCorrected linear;
}
```

Listing 9.9 Defining the interpolation scheme to calculate face values with skew correction

A more compact syntax can be used for defining the gradient in which the interpolation type is specified directly under gradSchemes shown in Listing 9.10.

```
gradSchemes
{
    default         none;
    grad(phi)   Gauss linear;
}
```

Listing 9.10 Defining the gradient calculation method: compact syntax

In this case the interpolation method is defined directly in the gradient dictionary. The choice of the syntax type is up to the user keeping in mind that a separate definition of the interpolation scheme helps clarifying the various steps of the gradient calculation.

9.6 Closure

This chapter presented the discretization details of two methods for computing the gradient at the centroids of control volume meshes in general non-orthogonal grid systems. One method is based on the Green-Gauss theorem and the second on the Least-Square reconstruction approach. Chapter 10 will be devoted to methods used for solving systems of algebraic equations.

9.7 Exercises

Exercise 1
In the configuration depicted in Fig. 9.22, the values of the variable ϕ and its gradient $\nabla\phi$ at C are known. Using these known values estimate the value of ϕ at F.

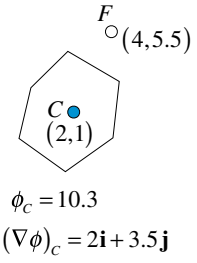


Fig. 9.22 A two dimensional element of centroid C and neighbor F

Exercise 2

For the two cells shown in Fig. 9.23 the value of some scalar ϕ at their centroids C and F can be calculated from

$$\phi(x, y) = 100\left(x^2 y + y^2 x\right)$$

The gradient at C and F are also computed to be the exact gradient of the function at these points such that

$$\nabla\phi(x, y) = 100\left(2xy + y^2\right)\mathbf{i} + 100\left(x^2 + 2yx\right)\mathbf{j}$$

With C and F located at (0.32, −0.1) and (0.52, 0.31), respectively, find the value of the gradient at face f (0.43, 0.1) numerically (not from the expression for the gradient) using

a. A simple averaging between C and F.
b. A corrected averaging between C and F.
c. Compare the two computed values with the exact gradient at point f.

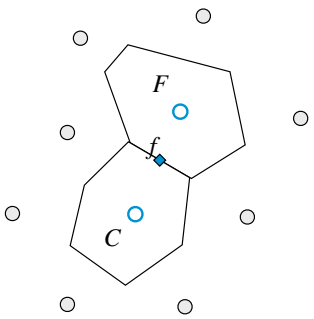


Fig. 9.23 Two elements in a two dimensional plane with their centroids ***C*** and ***F***

Exercise 3

Consider the mesh composed of equilateral triangular elements shown in Fig. 9.24. The coordinates of the mesh vertices are as shown. The temperatures at the cell centroids are given by

$$T(x, y, z) = 100\left(x^2 + y^2 + 1\right)$$

a. Compute the gradient at 1 using the Green-Gauss method.
b. Compute the gradient at 1 using the least squares approach with a limited stencil (i.e., nodes 2, 3 and 4) and with an extended stencil (i.e., nodes 2–13) used for the reconstruction.
c. Compare the least squares gradient with the limited and extended stencil and the gauss gradient to the exact gradient.

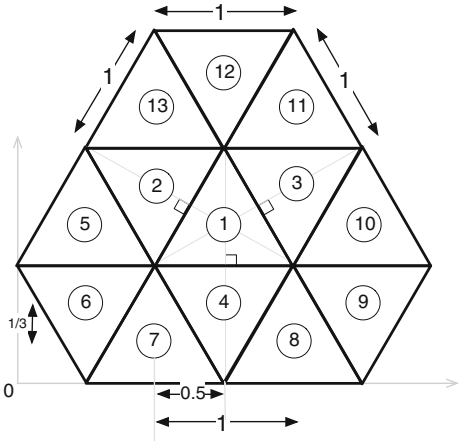


Fig. 9.24 A domain discretized using triangular elements

Exercise 4

Consider the uniform mesh shown in Fig. 9.25. The coordinates of the mesh vertices are as shown. The temperatures at the cell nodes are given by

$$T(x, y) = 100\left(x^3 + y^3 + xy + 1\right)$$

compute the gradient at the centroid of cell 1 using the gauss gradient with and without face corrections and compare to the exact gradient given by

$$\nabla T(x, y) = 100\left(3x^2 + y\right)\mathbf{i} + 100\left(3y^2 + x\right)\mathbf{j}$$

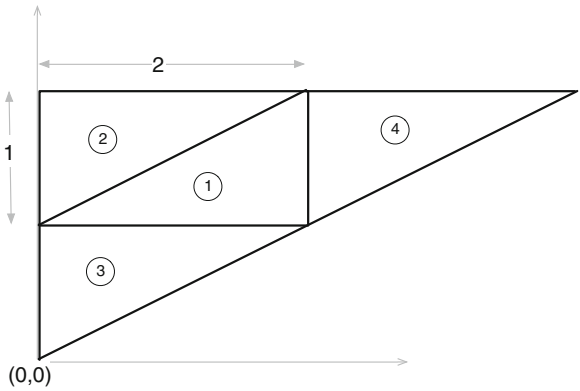


Fig. 9.25 Mesh system for Exercise 4

Exercise 5 (uFVM, OpenFOAM®)
Using uFVM and then OpenFOAM® write a program that

a. reads an OpenFOAM® mesh and sets a scalar field with values equal to $\phi(x, y, z) = 10x^2y^2 + 3y$
b. computes the gradients using the second order least square method,
c. computes the gradients using the gauss gradient method with linear interpolation,
d. computes the gradients using the gauss gradient method with vertex interpolation,
e. and computes the root mean square error between each of the computed gradients and the exact gradient.

Exercise 6 (uFVM, OpenFOAM®)
Using uFVM and then OpenFOAM® write a program that

a. reads an OpenFOAM® mesh and sets a scalar field with values equal to $\phi(x, y, z) = 10x^2y^2 + 3x$

Use the exact gradients at the element nodes to compute the gradients at the interior faces using

b. simple linear interpolation,
c. corrected linear interpolation,
d. and exact gradient formulation at the face centroids.
e. Compute the root mean square error between each of the computed gradients and the exact gradient.

Exercise 7
Starting with Eq. (9.8) derive Eq. (9.9).
Hint: $\mathbf{r}_f - \mathbf{r}_{f'} = \mathbf{r}_f - \mathbf{r}_C - \mathbf{r}_{cf'} = \mathbf{r}_f - \mathbf{r}_f - \mathbf{r}_{Ff'}$.

Exercise 8 (OpenFOAM®)

a. List all possible gradient type definitions available in OpenFOAM® by modifying the gradSchemes in the fvSchemes dictionary (Hint: just mistype a scheme, e.g., banana, and launch any solver or application, i.e., gradSchemes {default banana;}).
b. Define in the dictionary file fvSchemes the option to evaluate the explicit gradient with the least square algorithm.
c. Use the Doxygen documentation [9] to analyze the member function correctBoundaryConditions and explain its operations and aim.
d. Define in the dictionary file fvSchemes the option to use a limited gradient (face and cell) (Gauss limited 0.8;).

References

1. Ferziger JH, Perić M (2002) Computational Methods for Fluid Dynamics, 3rd edn. Springer, Berlin
2. Soni B (1998) Hybrid techniques in computational fluid dynamics. Technical Report no. MSSU-COE-ERC-98-3, Engineering Research Center for Computational Field Simulation, Mississippi State University
3. Frink NT, Parikh P, Pirzadeh S (1991) Aerodynamic Analysis of Complex Configurations Using Unstructured Grids. AIAA 91-3292
4. Cabello J, Morgon K, Lohner R (1994) A Comparison of Higher Order Schemes Used in a Finite Volume Solver For Unstructured Grids. AIAA 94-2293. Presented at the 25th AIAA Plasmadynamics and Lasers Conference, Colorado Springs, CO
5. Musaferija I, Gosman D (1997) Finite-Volume CFD procedure and adaptive error control strategy for grids of arbitrary topology. J Comp Phys 138:766–787
6. Barth TJ, Jespersen DC (1989) The design and application of upwind schemes on unstructured meshes. AIAA 89-0366
7. Ollivier-Gooch C, Van Altena M (2002) A high-order-accurate unstructured mesh finite-volume scheme for the advection–diffusion equation. J Comp Phys 181:729–752
8. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
9. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 10
Solving the System of Algebraic Equations

Abstract The result of the discretization process is a system of linear equations of the form $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$ where the unknowns $\boldsymbol{\phi}$, located at the centroids of the mesh elements, are the sought after values. In this system, the coefficients of the unknown variables constituting matrix $\mathbf{A}$ are the result of the linearization procedure and the mesh geometry, while vector $\mathbf{b}$ contains all sources, constants, boundary conditions, and non-linearizable components. Techniques for solving linear systems of equations are generally grouped into *direct* and *iterative* methods, with many sub-groups in each category. Since flow problems are highly non-linear, the coefficients resulting from their linearization process are generally solution dependent. For this reason and since an accurate solution is not needed at each iteration, direct methods have been rarely used in CFD applications. Iterative methods on the other hand have been more popular because they are more suited for this type of applications requiring lower computational cost per iteration and lower memory. The chapter starts by presenting few direct methods applicable to structured and/or unstructured grids (Gauss elimination, LU factorization, Tridiagonal and Pentadiagonal matrix algorithms) to set the ground for discussing the more widely used iterative methods in CFD applications. Then the performance and limitations of some of the basic iterative methods with and without preconditioning are reviewed. This include the Jacobi, Gauss-Siedel, Incomplete LU factorization, and the conjugate gradient methods. This is followed by an introduction to the multigrid method that is generally used in combination with iterative solvers to help addressing some of their important limitations.

10.1 Introduction

The starting point for any linear solver is the set of equations generated by the discretization process, which are written mathematically as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \tag{10.1}$$

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_10

where $\mathbf{A}$ is the matrix of coefficients of elements a_{ij}, $\boldsymbol{\phi}$ the vector of unknown variables ϕ_i, and $\mathbf{b}$ the vector of sources b_i. Using matrix numbering, the expanded form of Eq. (10.1) is given by

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN-1} & a_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} \tag{10.2}$$

Generally each row in the above matrix represents an equation defined over one element of the computational domain, and the non-zero coefficients are those related to the neighbors of that element. The coefficient a_{ij} measures the strength of the link between the value of ϕ_i at the centroid of the control volume and its neighbors. As a cell is connected to only few neighbors, with their number depending on the connectivity of the elements in the discretized domain, many of the coefficients are zeros and the resulting $\mathbf{A}$ matrix is always sparse (i.e., the non-zero coefficients are a very small fraction of the matrix). If in addition the method uses a structured grid system, the matrix $\mathbf{A}$ will be banded with all non-zero elements aligned along few diagonals. Therefore methods for efficiently solving such systems should exploit this characteristic.

As mentioned above techniques for solving algebraic systems of equations are broadly divided into two categories denoted by direct and iterative methods, respectively. In a direct method, matrix $\mathbf{A}$ is inverted and the solution $\boldsymbol{\phi}$ is computed in one step as $\boldsymbol{\phi} = \mathbf{A}^{-1}\mathbf{b}$. When the matrix $\mathbf{A}$ is large, computing its inverse is computationally very expensive requiring large memory. It is basically impractical to apply direct linear solvers in CFD applications as they generally involve non-linear systems of equations with their coefficients depending on the solution necessitating the use of an iterative process.

On the other hand, with iterative algebraic solvers the solution algorithm is repeatedly applied as many times as required until a pre-assigned level of convergence is reached without the need for a fully converged solution be attained at every iteration.

The presentation starts with few direct linear solvers applicable to structured and unstructured grid methods. This is followed by a description of solution algorithms that take advantage of the banded structure of the coefficient matrix in structured grid systems. The main focus of the chapter is, however, on a specific class of iterative linear algebraic solvers that has been identified to be generally very efficient and economical with the FVM, and has been exclusively implemented as the linear solver in almost all finite volume-based codes.

10.2 Direct or Gauss Elimination Method

Even though direct methods are not efficient at solving sparse systems of linear algebraic equations due to their high computational cost, their discussion will pave the way for introducing efficient iterative methods in the next section. The simplest direct method for finding solutions to the system of equations described by Eq. (10.1) is the Gauss elimination technique, which will be described first. The transformation of the system into an equivalent upper triangular system, when using the Gauss elimination method, has motivated the development of the Lower-Upper (LU) triangulation method, which will also be presented. In this approach, matrix **A** is decomposed into the product of two matrices **L** and **U** with **L** being a lower triangular matrix and **U** an upper triangular one. This procedure is also known as LU factorization. In addition, direct methods benefiting from the banded structure of **A**, applicable to structured grid methods, will be discussed.

10.2.1 Gauss Elimination

The best way to describe the Gauss elimination technique is to start with a simple example. For that purpose a linear system of equations in the two unknowns ϕ_1 and ϕ_2 is considered. The equations are given by

$$a_{11}\phi_1 + a_{12}\phi_2 = b_1 \tag{10.3}$$

$$a_{21}\phi_1 + a_{22}\phi_2 = b_2 \tag{10.4}$$

The system can be solved by eliminating one of the variables from one of the equations [say ϕ_1 from Eq. (10.4)]. This can be done by multiplying Eq. (10.3) by a_{21}/a_{11} and subtracting the resulting equation from Eq. (10.4). This yields

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)\phi_2 = b_2 - \frac{a_{21}}{a_{11}}b_1 \tag{10.5}$$

The obtained equation involves only one unknown. As such it can be used to solve for ϕ_2, which is obtained as

$$\phi_2 = \frac{b_2 - \dfrac{a_{21}}{a_{11}}b_1}{a_{22} - \dfrac{a_{12}a_{21}}{a_{11}}} \tag{10.6}$$

Knowing ϕ_2, its value can be substituted back into Eq. (10.3) to find ϕ_1. Performing this step, ϕ_1 is found to be

$$\phi_1 = \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} \frac{b_2 - \frac{a_{21}}{a_{11}} b_1}{a_{22} - \frac{a_{12}}{a_{11}} a_{21}} \tag{10.7}$$

The above procedure is composed of two steps. In the first step the equations are manipulated in order to eliminate one of the unknowns. The end result of this step is an equation with one unknown. In the second step, this equation is solved directly and the result back-substituted into one of the equations to solve for the remaining unknown. The same procedure can be generalized to a system of N equations described by Eq. (10.1) or (10.2), as detailed next.

10.2.2 Forward Elimination

In the derivations to follow, the first row of **A** refers to the discretized equation for ϕ_1, the second row represents the equation for ϕ_2, and in general the ith row refers to the equation for ϕ_i. The procedure starts by eliminating ϕ_1 from all equations below row 1 in **A**. To eliminate ϕ_1 from the ith row ($i = 2, 3, \ldots, N$), the coefficients of the first row are multiplied by a_{i1}/a_{11} and the resulting equation is subtracted from the ith row. The system of equations by the end of this step becomes

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N-1} & a_{1N} \\ 0 & a'_{22} & \ldots & a'_{2N-1} & a'_{2N} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & a'_{N2} & \ldots & a'_{NN-1} & a'_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ \vdots \\ \vdots \\ b'_N \end{bmatrix} \tag{10.8}$$

Then ϕ_2 is eliminated from all equations below row 2 in the modified **A**. To eliminate ϕ_2 from the ith row ($i = 3, 4, \ldots, N$), the coefficients of the second row are multiplied by a'_{i2}/a'_{22} and the resulting equation is subtracted from the ith row. Then ϕ_3 is eliminated from all rows below the third row in the modified coefficient matrix and the process is continued until ϕ_{N-1} is eliminated from the Nth row leading to the following equivalent system of equations with its matrix **A** transformed into an upper triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1N-1} & a_{1N} \\ 0 & a'_{22} & a'_{23} & \ldots & a'_{2N-1} & a'_{2N} \\ 0 & 0 & a''_{33} & \cdots & a''_{3N-1} & a''_{3N} \\ \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & a^{N-1}_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b^{N-1}_N \end{bmatrix} \tag{10.9}$$

The resulting algorithm is as described below.

10.2.3 Forward Elimination Algorithm

```
For k = 1 to N − 1
  {
    For i = k + 1 to N
      {
        Ratio = a_ik / a_kk
          {
            For j = k + 1 to N
                a_ij = a_ij − Ratio * a_kj
          }
        b_i = b_i − Ratio * b_k
      }
  }
```

10.2.4 Backward Substitution

The modified system of equations given by Eq. (10.9) indicates that the only unknown in the Nth equation is ϕ_N. Therefore this equation can be used to obtain ϕ_N as

$$\phi_N = \frac{b_N^{N-1}}{a_{NN}^{N-1}} \tag{10.10}$$

The $(N - 1)$th equation is function of ϕ_{N-1} and ϕ_N. Having found ϕ_N then this equation can be used to find ϕ_{N-1} as

$$\phi_{N-1} = \frac{b_{N-1}^{N-2} - a_{N-1N}^{N-2}\phi_N}{a_{N-1N-1}^{N-2}} \tag{10.11}$$

The process continues moving backward and by the time the ith equation is reached, the values $\phi_{i+1}, \phi_{i+2}, \phi_{i+3}, \ldots, \phi_{N-1}, \phi_N$ would have become available and as such ϕ_i can be computed using

$$\phi_i = \frac{b_i^{i-1} - \sum_{j=i+1}^{N} a_{ij}^{i-1}\phi_j}{a_{ii}^{i-1}} \tag{10.12}$$

the process is continued until ϕ_1 is calculated. Algorithmically, this is represented as shown below.

10.2.5 Back Substitution Algorithm

$$\phi_N = \frac{b_N}{a_{NN}}$$
$$For\ i = N-1\ to\ 1$$
$$\{$$
$$Term = 0$$
$$\{$$
$$For\ j = i+1\ to\ N$$
$$Term = Term + a_{ij} * \phi_j$$
$$\}$$
$$\phi_i = \frac{\phi_i - Term}{a_{ii}}$$
$$\}$$

Techniques to improve the performance of the method to avoid division by zero through pivoting (interchanging rows in order to select the largest pivoting element) and to reduce roundoff errors in large systems are available but are not discussed here. Interested readers may consult specialized textbooks on the subject [1–4]. The presented algorithm shows that the method is expensive and the number of operations required to solve a linear system of N equations is proportional to $N^3/3$ of which only $N^2/2$ arithmetic operations are required for back substitution. This high computational cost has enticed researchers to look for more efficient specialized solvers for systems with sparse matrices.

10.2.6 LU Decomposition

Another direct method for solving linear algebraic systems of equations is the LU or more generally the PLU (where P refers to the pivoting process mentioned above), in this book reference is made only to LU factorization method, which are variants

of the Gauss elimination method. The advantage of these methods over the Gauss elimination method is that once the (P)LU factorization is performed the linear system can be solved as many times as needed for different values of the right hand side vector **b** without performing any additional elimination, which would still be required with the Gauss method.

Based on the elimination performed in the previous section, Eq. (10.1) was transformed into an upper triangular matrix as given by Eq. (10.9), which can be written as

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1N-1} & u_{1N} \\ 0 & u_{22} & u_{23} & \dots & u_{2N-1} & u_{2N} \\ 0 & 0 & u_{33} & \dots & u_{3N-1} & u_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & u_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_N \end{bmatrix} \tag{10.13}$$

Using compact matrix notation Eq. (10.13) can be simplified to

$$\mathbf{U}\boldsymbol{\phi} - \mathbf{c} = 0 \tag{10.14}$$

Let **L** be a unit lower triangular matrix (diagonal elements are set to 1 in order to make the factorization unique) given by

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \ell_{21} & 1 & 0 & \dots & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ \ell_{N1} & \ell_{N2} & \ell_{N3} & \dots & \ell_{NN-1} & 1 \end{bmatrix} \tag{10.15}$$

such that if Eq. (10.14) is multiplied by **L**, Eq. (10.1) is recovered. If this holds, then the following can be written:

$$\mathbf{L}(\mathbf{U}\boldsymbol{\phi} - \mathbf{c}) = \mathbf{L}\mathbf{U}\boldsymbol{\phi} - \mathbf{L}\mathbf{c} = \mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.16}$$

Based on matrix properties it follows that

$$\mathbf{L}\mathbf{U} = \mathbf{A} \tag{10.17}$$

and

$$\mathbf{L}\mathbf{c} = \mathbf{b} \tag{10.18}$$

Equation (10.17) indicates that **A** is written as the product of a proper lower and an upper triangular matrix, known as LU factorization.

10.2.7 The Decomposition Step

The efficient procedure to find the **L** and **U** coefficients described next is denoted by the Crout decomposition [1–4]. In the original Crout algorithm a unit upper triangular matrix is used whereas here a unit lower triangular matrix is assumed. The procedure is based on multiplying **L** and **U** to obtain **A** such that

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \ell_{21} & 1 & 0 & \dots & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ \ell_{N1} & \ell_{N2} & \ell_{N3} & \dots & \ell_{NN-1} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1N-1} & u_{1N} \\ 0 & u_{22} & u_{23} & \dots & u_{2N-1} & u_{2N} \\ 0 & 0 & u_{33} & \dots & u_{3N-1} & u_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & u_{NN} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2N-1} & a_{2N} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N-1} & a_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN-1} & a_{NN} \end{bmatrix} \tag{10.19}$$

The calculation of the coefficients starts by multiplying the first row of **L** by all columns of **U**, and equating with the corresponding coefficients of **A** to yield

$$u_{1j} = a_{1j} \quad j = 1, 2, 3, \dots, N \tag{10.20}$$

Then the second through *N*th rows of **L** are multiplied by the first column of **U** leading to

$$\ell_{i1} u_{11} = a_{i1} \Rightarrow \ell_{i1} = \frac{a_{i1}}{u_{11}} \quad i = 2, 3, \dots, N \tag{10.21}$$

The process is repeated by multiplying the second row of **L** by the second through *N*th columns of **U** to give

$$u_{2j} = a_{2j} - \ell_{21} u_{1j} \quad j = 2, 3, \dots, N \tag{10.22}$$

after that the third through *N*th rows of **L** are multiplied by the second column of **U** to give

$$\ell_{i2} u_{22} + \ell_{i1} u_{12} = a_{i2} \Rightarrow \ell_{i2} = \frac{a_{i2} - \ell_{i1} u_{12}}{u_{22}} \quad i = 3, 4, \dots, N \tag{10.23}$$

In general, the ith row of $\mathbf{L}$ is multiplied by the ith through Nth columns of $\mathbf{U}$, resulting in

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \quad j = i, i+1, \ldots, N \tag{10.24}$$

and the $(i + 1)$th through Nth rows of $\mathbf{L}$ are multiplied by the ith column of $\mathbf{U}$, giving

$$\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}} \quad k = i+1, i+2, \ldots, N \tag{10.25}$$

For the Nth row of $\mathbf{L}$, its coefficients are multiplied by the coefficients of the Nth column of $\mathbf{U}$ from which u_{NN} is obtained as

$$u_{NN} = a_{NN} - \sum_{k=1}^{N-1} \ell_{Nk} u_{kN} \tag{10.26}$$

A summary of the LU factorization is shown algorithmically below.

10.2.8 LU Decomposition Algorithm

$u_{1j} = a_{1j} \quad j = 1 \; to \; N$

$\ell_{i1} = \dfrac{a_{i1}}{u_{11}} \quad i = 2 \; to \; N$

For $i = 2 \; to \; N-1$

{

$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \qquad j = i, i+1, ..., N$

$\ell_{ki} = \dfrac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}} \qquad k = i+1, i+2, ..., N$

}

$u_{NN} = a_{NN} - \sum_{i=1}^{N-1} \ell_{Ni} u_{iN}$

10.2.9 The Substitution Step

Having decomposed the original matrix **A** into **L** and **U**, the system of equations can be solved in a two step procedure via Eqs. (10.18) and (10.14). Note that the two-step procedure is equivalent to solving two linear systems of equations but now simplified by the fact that **L** and **U** are of lower and upper triangular form, respectively.

In the first step the vector **c** is obtained from Eq. (10.18) by **forward substitution**. The process can be described as

$$\begin{aligned} c_1 &= b_1 \\ c_i &= b_i - \sum_{j=1}^{i-1} \ell_{ij} c_j \quad i = 2, 3, \ldots, N \end{aligned} \tag{10.27}$$

In the second step, the ϕ values are found from Eq. (10.14) by **back substitution**. The process is described by

$$\begin{aligned} \phi_N &= \frac{c_N}{u_{NN}} \\ \phi_i &= \frac{c_i - \sum_{j=i+1}^{N} u_{ij}\phi_j}{u_{ii}} \quad i = N-1, N-2, \ldots, 3, 2, 1 \end{aligned} \tag{10.28}$$

The elements of **L** and **U** can be directly stored in the original matrix **A** if it is no longer needed. This is because the elements of **A** are only needed when the corresponding elements of either **L** or **U** are calculated. The number of operations required to perform the LU factorization of a square matrix of size $N \times N$ is $2N^3/3$, which is double the number of operations required to solve the same system of equations by Gauss elimination. Again the advantage of using LU factorization is when the same matrix **A** applies to many systems with different **b** vectors. Nevertheless, the main reason for introducing the LU factorization is because it forms the basis for developing some of the more efficient iterative solvers of linear algebraic systems of equations, which will be introduced in the next section.

10.2.10 LU Decomposition and Gauss Elimination

It may not be apparent, but Gauss elimination can be used to perform LU decomposition. It was shown that the forward elimination step results in an upper triangular matrix **U**. In the process however, **L** is actually produced. The elements of **L** are the factors (denoted by *ratio* in the Gauss elimination algorithm) by which the rows are multiplied during the various elimination steps. The below algorithm, which assumes a unit lower triangular matrix **L**, performs the LU decomposition of **A** by Gauss elimination.

10.2.11 LU Decomposition Algorithm by Gauss Elimination

$u_{1j} = a_{1j} \quad j = 1 \text{ to } N$

For $k = 1$ *to* $N-1$

{

For $i = k+1$ *to* N

{

$$\ell_{ik} = \frac{a_{ik}}{a_{kk}}$$

{

For $j = k+1$ *to* N

$$u_{ij} = a_{ij} - \ell_{ik} * a_{kj}$$

}

}

}

Example 1

Solve the following system of linear algebraic equations using the LU decomposition technique:

$$\begin{bmatrix} 3 & -1 & 0 & 0 \\ -2 & 6 & -1 & 0 \\ 0 & -2 & 6 & -1 \\ 0 & 0 & -2 & 7 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ -3 \end{bmatrix}$$

Solution

The system is of the form $\mathbf{A}\boldsymbol{\phi} = b$. The **L** and **U** should satisfy

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} 3 & -1 & 0 & 0 \\ -2 & 6 & -1 & 0 \\ 0 & -2 & 6 & -1 \\ 0 & 0 & -2 & 7 \end{bmatrix}$$

Following the procedure described above the elements are calculated as follows:

$$u_{1j} = a_{1j} \quad j = 1,2,3,4 \Rightarrow \begin{cases} u_{11} = 3 \\ u_{12} = -1 \\ u_{13} = 0 \\ u_{14} = 0 \end{cases}$$

$$\ell_{i1} = \frac{a_{i1}}{u_{11}} \quad i = 2,3,4 \Rightarrow \begin{cases} \ell_{21} = \dfrac{a_{21}}{u_{11}} = -\dfrac{2}{3} \\ \ell_{31} = \dfrac{a_{31}}{u_{11}} = 0 \\ \ell_{41} = \dfrac{a_{41}}{u_{11}} = 0 \end{cases}$$

$$u_{2j} = a_{2j} - \ell_{21}u_{1j} \quad j = 2,3,4 \Rightarrow \begin{cases} u_{22} = a_{22} - \ell_{21}u_{12} = \dfrac{16}{3} \\ u_{23} = a_{23} - \ell_{21}u_{13} = -1 \\ u_{24} = a_{24} - \ell_{21}u_{14} = 0 \end{cases}$$

$$\ell_{i2} = \frac{a_{i2} - \ell_{i1}u_{12}}{u_{22}} \quad i = 3,4 \Rightarrow \begin{cases} \ell_{32} = \frac{a_{32} - \ell_{31}u_{12}}{u_{22}} = -\dfrac{3}{8} \\ \ell_{42} = \frac{a_{42} - \ell_{41}u_{12}}{u_{22}} = 0 \end{cases}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj} \quad i = 3, j = 3,4 \Rightarrow \begin{cases} u_{33} = a_{33} - \ell_{31}u_{13} - \ell_{32}u_{23} = \dfrac{45}{8} \\ u_{34} = a_{34} - \ell_{31}u_{14} - \ell_{32}u_{24} = -1 \end{cases}$$

$$\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj}u_{ji}}{u_{ii}} \quad i = 3, k = 4 \Rightarrow \ell_{43} = \frac{a_{43} - \ell_{41}u_{13} - \ell_{42}u_{23}}{u_{33}} = -\frac{16}{45}$$

$$u_{NN} = a_{NN} - \sum_{k=1}^{N-1} \ell_{Nk}u_{kN} \Rightarrow u_{44} = a_{44} - \ell_{41}u_{14} - \ell_{42}u_{24} - \ell_{43}u_{34} = \frac{299}{45}$$

Therefore the **L** and **U** matrices are given by

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\dfrac{2}{3} & 1 & 0 & 0 \\ 0 & -\dfrac{3}{8} & 1 & 0 \\ 0 & 0 & -\dfrac{16}{45} & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 3 & -1 & 0 & 0 \\ 0 & \dfrac{16}{3} & -1 & 0 \\ 0 & 0 & \dfrac{45}{8} & -1 \\ 0 & 0 & 0 & \dfrac{299}{45} \end{bmatrix}$$

The **c** vector should satisfy

$$\mathbf{Lc} = \mathbf{b} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{2}{3} & 1 & 0 & 0 \\ 0 & -\frac{3}{8} & 1 & 0 \\ 0 & 0 & -\frac{16}{45} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ -3 \end{bmatrix}$$

with its solution obtained as

$$\begin{aligned} c_1 &= 3 \\ -\frac{2}{3}c_1 + c_2 &= 4 \Rightarrow c_2 = 6 \\ -\frac{3}{8}c_2 + c_3 &= 5 \Rightarrow c_3 = \frac{29}{4} \\ -\frac{16}{45}c_3 + c_4 &= -3 \Rightarrow c_4 = -\frac{19}{45} \end{aligned}$$

The solution to the original equation is obtained by solving

$$\mathbf{U}\boldsymbol{\phi} = \mathbf{c} \Rightarrow \begin{bmatrix} 3 & -1 & 0 & 0 \\ 0 & \frac{16}{3} & -1 & 0 \\ 0 & 0 & \frac{45}{8} & -1 \\ 0 & 0 & 0 & \frac{299}{45} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ \frac{29}{4} \\ -\frac{19}{45} \end{bmatrix}$$

with the solution found as

$$\left.\begin{aligned} \frac{299}{45}\phi_4 = -\frac{19}{45} \Rightarrow \phi_4 = -\frac{19}{299} \\ \frac{45}{8}\phi_3 - \phi_4 = \frac{29}{4} \Rightarrow \phi_3 = \frac{382}{299} \\ \frac{16}{3}\phi_2 - \phi_3 = 6 \Rightarrow \phi_2 = \frac{408}{299} \\ 3\phi_1 - \phi_2 = 3 \Rightarrow \phi_1 = \frac{435}{299} \end{aligned}\right\} \Rightarrow \boldsymbol{\phi} = \begin{bmatrix} \frac{435}{299} \\ \frac{408}{299} \\ \frac{382}{299} \\ -\frac{19}{299} \end{bmatrix}$$

10.2.12 Direct Methods for Banded Sparse Matrices

The Gauss elimination and LU decomposition methods are applicable to any system of equations. In specific it can be used for solving the system of equations resulting from the discretization of the conservation equations of interest in this book on

structured or unstructured grid networks. When a structured grid method is used the discretization process results in a system of equations with the non-zero elements of its matrix of coefficients aligning along few diagonals. Depending on the discretization stencil used and the dimension of the problem being solved, tridiagonal or pentadiagonal matrices may arise, for which efficient algorithms have been developed as described next.

10.2.13 TriDiagonal Matrix Algorithm (TDMA)

The TriDiagonal Matrix Algorithm (TDMA), also known as Thomas algorithm [5, 6], solves the system of algebraic equations with a tridiagonal coefficient matrix written as

$$a_i\phi_i + b_i\phi_{i+1} + c_i\phi_{i-1} = d_i \quad i = 1, 2, 3, \ldots, N \quad c_1 = b_N = 0 \tag{10.29}$$

For the grid arrangement adopted in this book, i refers to the grid point location shown in Fig. 10.1.

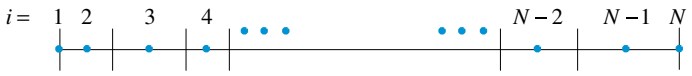


Fig. 10.1 One dimensional grid arrangement

For $i = 1$ the equation can be used to solve for ϕ_1 in term of ϕ_2 as

$$i = 1 \Rightarrow a_1\phi_1 = -b_1\phi_2 + d_1 \Rightarrow \phi_1 = -\frac{b_1}{a_1}\phi_2 + \frac{d_1}{a_1} \tag{10.30}$$

Similarly for $i = 2$ Eq. (10.29) with the help of Eq. (10.30) allows expressing ϕ_2 solely in term of ϕ_3 as

$$i = 2 \Rightarrow a_2\phi_2 = -b_2\phi_3 - c_2\phi_1 + d_2 \Rightarrow \phi_2 = -\frac{a_1 b_2}{a_1 a_2 - c_2 b_1}\phi_3 + \frac{d_2 a_1 - c_2 d_1}{a_1 a_2 - c_2 b_1} \tag{10.31}$$

The same can be repeated for ϕ_3 through ϕ_N suggesting that in general ϕ_i can be expressed as function of ϕ_{i+1} according to

$$\phi_i = P_i\phi_{i+1} + Q_i \quad i = 1, 2, 3, \ldots, N \tag{10.32}$$

Equation (10.32) for $i - 1$ when combined with Eq. (10.29) results in

$$\left.\begin{array}{c}\phi_{i-1} = P_{i-1}\phi_i + Q_{i-1} \\ a_i\phi_i + b_i\phi_{i+1} + c_i\phi_{i-1} = d_i\end{array}\right\} \Rightarrow \phi_i = -\frac{b_i}{a_i + c_iP_{i-1}}\phi_{i+1} + \frac{d_i - c_iQ_{i-1}}{a_i + c_iP_{i-1}} \quad (10.33)$$

Comparing Eq. (10.32) with Eq. (10.33) the following recurrence relations for P_i and Q_i are found:

$$P_i = -\frac{b_i}{a_i + c_iP_{i-1}} \quad Q_i = \frac{d_i - c_iQ_{i-1}}{a_i + c_iP_{i-1}} \quad i = 1, 2, \ldots, N \quad (10.34)$$

For $i = 1$ the values for P_1 and Q_1 are computed from Eq. (10.30) as

$$P_1 = -\frac{b_1}{a_1} \quad Q_1 = \frac{d_1}{a_1} \quad (10.35)$$

For $i = N$, since $b_N = 0$ the following is deduced:

$$b_N = 0 \Rightarrow P_N = 0 \Rightarrow \phi_N = Q_N \quad (10.36)$$

The TDMA solution algorithm can be summarized as follows:

1. Compute the values for P_1 and Q_1 using Eq. (10.35)
2. For $i = 2, 3, \ldots, N$ use forward recursion to compute the values of P_i and Q_i from Eq. (10.34)
3. Set $\phi_N = Q_N$ as given by Eq. (10.36)
4. For $i = N-1, N-2, \ldots 3, 2, 1$ use backward recursion to compute the values of ϕ_i from Eq. (10.32)

10.2.14 PentaDiagonal Matrix Algorithm (PDMA)

The PentaDiagonal Matrix Algorithm (PDMA) [7–10], solves the system of algebraic equations with a pentadiagonal coefficient matrix arising from discretization schemes that relate the value of ϕ_i at grid point i to the values at its two upstream ($i - 1$ and $i - 2$) and two downstream ($i + 1$ and $i + 2$) neighboring node values. For the notation illustrated schematically in Fig. 10.1, the general algebraic equation is written as

$$a_i\phi_i + b_i\phi_{i+2} + c_i\phi_{i+1} + d_i\phi_{i-1} + e_i\phi_{i-2} = f_i \quad i = 1, 2, 3, \ldots, N \quad (10.37)$$

Subject to

$$\begin{array}{c} d_1 = e_1 = e_2 = 0 \\ b_{N-1} = b_N = c_N = 0 \end{array} \quad (10.38)$$

For $i = 1$, Eq. (10.37) gives

$$\phi_1 = -\frac{b_1}{a_1}\phi_3 - \frac{c_1}{a_1}\phi_2 + \frac{f_1}{a_1} \tag{10.39}$$

while for $i = 2$ the value of ϕ_2 is found to be

$$\phi_2 = -\frac{a_1 b_2}{a_1 a_2 - d_2 c_1}\phi_4 - \frac{a_1 c_2 - b_1 d_2}{a_1 a_2 - d_2 c_1}\phi_3 + \frac{a_1 f_2 - d_2 f_1}{a_1 a_2 - d_2 c_1} \tag{10.40}$$

The process can be continued for other values of i and in general ϕ_i can be expressed as

$$\phi_i = P_i \phi_{i+2} + Q_i \phi_{i+1} + R_i \quad i = 1, 2, 3, \ldots, N \tag{10.41}$$

Computing ϕ_{i-1} and ϕ_{i-2} using Eq. (10.41) and substituting their values in Eq. (10.37), an equation for ϕ_i is derived as

$$\begin{aligned}\phi_i = &-\frac{b_i}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\phi_{i+2} \\ &-\frac{c_i + (d_i + e_i Q_{i-2})P_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\phi_{i+1} \\ &+\frac{f_i - e_i R_{i-2} - (d_i + e_i Q_{i-2})R_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\end{aligned} \tag{10.42}$$

Comparing Eqs. (10.41) and (10.42) P_i, Q_i, and R_i are found as

$$\begin{aligned}P_i &= -\frac{b_i}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}} \\ Q_i &= -\frac{c_i + (d_i + e_i Q_{i-2})P_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}} \\ R_i &= \frac{f_i - e_i R_{i-2} - (d_i + e_i Q_{i-2})R_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\end{aligned} \tag{10.43}$$

with their values for $i = 1$ and 2 given by

$$\begin{aligned}&P_1 = -\frac{b_1}{a_1} \quad Q_1 = -\frac{c_1}{a_1} \quad R_1 = \frac{f_1}{a_1} \\ &P_2 = -\frac{b_2}{a_2 + d_2 Q_1} \quad Q_2 = -\frac{c_2 + d_2 P_1}{a_2 + d_2 Q_1} \quad R_2 = \frac{f_2 - d_2 R_1}{a_2 + d_2 Q_1}\end{aligned} \tag{10.44}$$

Since $b_{N-1} = b_N = c_N = 0$ then $P_{N-1} = P_N = Q_N = 0$. Thus, the equations for ϕ_{N-1} and ϕ_N are found from

$$\begin{aligned}\phi_N &= R_N \\ \phi_{N-1} &= Q_{N-1}\phi_N + R_{N-1}\end{aligned} \tag{10.45}$$

The PDMA solution algorithm can be summarized as follows:

1. Compute the values for P_1, Q_1, R_1, P_2, Q_2, and R_2 using Eq. (10.44).
2. For $i = 3,4,...,N$ use forward recursion to compute the values of P_i, Q_i, and R_i from Eq. (10.43).
3. Compute ϕ_N and ϕ_{N-1} from Eq. (10.45).
4. For $i = N-2,...3,2,1$ use backward recursion to compute the values of ϕ_i from Eq. (10.41).

10.3 Iterative Methods

Direct methods are generally not appropriate for solving large systems of equations particularly when the coefficient matrix is sparse, i.e., when most of the matrix elements are zero. This is more so when the linearized system of equations is nonlinear with solution dependent coefficients, or when dealing with time dependent problems. This is exactly the type of equations encountered when solving fluid flow problems.

In contrast, iterative methods are more appealing for these problems since the solution of the linearized system becomes part of the iterative solution process. Add to that the low computer storage and low computational cost requirements of this approach relative to the direct method.

There are many families of iterative methods and for a thorough review of this approach the reader is directed to dedicated books on the topic [11–14]. In this chapter a brief examination of basic iterative methods is provided along with an appraisal of multigrid algorithms that are generally used to address their deficiency. The Gauss elimination and LU decomposition direct methods were introduced for the sole purpose of clarifying some fundamental numerical processes needed for understanding iterative methods.

To unify the presentation of these methods, the coefficient matrix will be written in the following form:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U} \tag{10.46}$$

where **D**, **L**, and **U** refers to a diagonal, strictly lower, and strictly upper matrix, respectively.

Iterative methods for solving a linear system of the type $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$, compute a series of solutions $\boldsymbol{\phi}^{(n)}$ that, if certain conditions are satisfied, converge to the exact solution $\boldsymbol{\phi}$. Thus, for the solution, a starting point is chosen (i.e., $\boldsymbol{\phi}^{(0)}$ is selected as

the initial condition or initial guess) and an iterative procedure that computes $\boldsymbol{\phi}^{(n)}$ from the previously computed $\boldsymbol{\phi}^{(n-1)}$ field is developed.

A "fixed-point" iteration can always be associated to the above system by decomposing matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{M} - \mathbf{N} \tag{10.47}$$

Using this decomposition, Eq. (10.1) is rewritten as

$$(\mathbf{M} - \mathbf{N})\boldsymbol{\phi} = \mathbf{b} \tag{10.48}$$

Applying a fixed point iteration solution procedure, Eq. (10.48) becomes

$$\mathbf{M}\boldsymbol{\phi}^{(n)} = \mathbf{N}\boldsymbol{\phi}^{(n-1)} + \mathbf{b} \tag{10.49}$$

which can be rewritten in the following form:

$$\boldsymbol{\phi}^{(n)} = \mathbf{B}\boldsymbol{\phi}^{(n-1)} + \mathbf{C}\mathbf{b} \quad n = 1, 2, \ldots \tag{10.50}$$

where $\mathbf{B} = \mathbf{M}^{-1}\mathbf{N}$ and $\mathbf{C} = \mathbf{M}^{-1}$. Different choices of these matrices define different iterative methods.

Before embarking on the description of the various iterative methods, a minimal set of characteristics that an iterative method should possess to guarantee convergence is first presented.

A. The iterative equation can be written at convergence as

$$\boldsymbol{\phi} = \mathbf{B}\boldsymbol{\phi} + \mathbf{C}\mathbf{b} \tag{10.51}$$

which, after rearranging, becomes

$$\mathbf{C}^{-1}(\mathbf{I} - \mathbf{B})\boldsymbol{\phi} = \mathbf{b} \tag{10.52}$$

Comparing Eq. (10.52) with Eq. (10.1), the coefficient matrix is obtained as

$$\mathbf{A} = \mathbf{C}^{-1}(\mathbf{I} - \mathbf{B}) \tag{10.53}$$

or, alternatively, as

$$\mathbf{B} + \mathbf{C}\mathbf{A} = \mathbf{I} \tag{10.54}$$

This relation between the various matrices ensures that once the exact solution is reached all consecutive iterations will not modify it.

B. Starting from some guess $\boldsymbol{\phi}^{(0)} \neq \boldsymbol{\phi}$, the method should guarantee that $\boldsymbol{\phi}^{(n)}$ will converge to $\boldsymbol{\phi}$ as n increases. Since $\boldsymbol{\phi}^{(n)}$ can be expressed in terms of $\boldsymbol{\phi}^{(0)}$ as

$$\boldsymbol{\phi}^{(n)} = \mathbf{B}^n \boldsymbol{\phi}^{(0)} + \sum_{i=0}^{n-1} \mathbf{B}^i \mathbf{C}\mathbf{b} \tag{10.55}$$

then, for the above to be true, $\mathbf{B}$ should satisfy

$$\lim_{n\to\infty} \mathbf{B}^n = \lim_{n\to\infty} \underbrace{\mathbf{B} * \mathbf{B} * \mathbf{B} \cdots * \mathbf{B}}_{n \text{ times}} = 0 \tag{10.56}$$

Equation (10.56) implies that the spectral radius of $\mathbf{B}$ should be less than 1, i.e.,

$$\rho(\mathbf{B}) < 1 \tag{10.57}$$

This condition guaranties that the iterative method is self corrective, i.e., it is robust to any error adversely inserted into the solution vector $\boldsymbol{\phi}$.
More insight into the above condition can be obtained by defining the error $\mathbf{e}^{(n)}$ in the solution as the difference between the exact value and the value at any iteration (n), then

$$\mathbf{e}^{(n)} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi} \quad \text{and} \quad \mathbf{e}^{(n-1)} = \boldsymbol{\phi}^{(n-1)} - \boldsymbol{\phi} \tag{10.58}$$

Subtracting Eq. (10.51) from Eq. (10.50) and using the definitions in Eq. (10.58), a relation between the error at iteration n and $n-1$ is obtained as

$$\mathbf{e}^{(n)} = \mathbf{B}\mathbf{e}^{(n-1)} \tag{10.59}$$

Thus, for the method to converge, the following should be satisfied:

$$\lim_{n\to\infty} \mathbf{e}^{(n)} = 0 \tag{10.60}$$

To translate Eq. (10.60) into something meaningful, the eigenvectors of $\mathbf{B}$ are assumed to be complete and to form a full set, meaning that they form a basis for $\mathbf{R}^N$. This being the case then $\mathbf{e}$ can be expressed as a linear combination of the N eigenvectors $\mathbf{v}$ of $\mathbf{B}$. That is

$$\mathbf{e} = \sum_{i=1}^{N} \alpha_i \mathbf{v}_i \tag{10.61}$$

with each of the eigenvectors satisfying

$$\mathbf{B}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{10.62}$$

where λ_i is the eigenvalue corresponding to the eigenvector $\mathbf{v}_i$. Starting with the first iteration, Eq. (10.59) gives

$$\mathbf{e}^{(1)} = \mathbf{B}\mathbf{e}^{(0)} = \mathbf{B}\sum_{i=1}^{N}\alpha_i\mathbf{v}_i = \sum_{i=1}^{N}\alpha_i(\mathbf{B}\mathbf{v}_i) = \sum_{i=1}^{N}\alpha_i\lambda_i\mathbf{v}_i \tag{10.63}$$

For the second iteration, the error is obtained as

$$\mathbf{e}^{(2)} = \mathbf{B}\mathbf{e}^{(1)} = \mathbf{B}\sum_{i=1}^{N}\alpha_i\lambda_i\mathbf{v}_i = \sum_{i=1}^{N}\alpha_i\lambda_i(\mathbf{B}\mathbf{v}_i) = \sum_{i=1}^{N}\alpha_i\lambda_i^2\mathbf{v}_i \tag{10.64}$$

This procedure can be continued and it is easily shown by induction that

$$\mathbf{e}^{(n)} = \sum_{i=1}^{N}\alpha_i\lambda_i^n\mathbf{v}_i \tag{10.65}$$

Therefore for the iterative procedure to converge as n approaches infinity, all eigenvalues should be less than 1. If any of them is greater than 1 then the error will tend to infinity. This explains the importance of the spectral radius ρ of the matrix $\mathbf{B}$ defined as

$$\rho(\mathbf{B}) = \max_{i=1}^{N}(\lambda_i) \tag{10.66}$$

that was mentioned above. The convergence of iterative methods is accelerated by reducing the spectral radius of the iterative matrix. This is at the heart of iterative techniques.

C. Some type of a stopping criterion is needed with iterative methods. Many used criteria are based on a variation of the norm of the residual error defined as

$$\mathbf{r}^{(n)} = \mathbf{A}\boldsymbol{\phi}^{(n)} - \mathbf{b} \tag{10.67}$$

One criterion is to find the maximum residual in the domain and to require its value to become less than some threshold ε to declare a solution converged, i.e.,

$$\underset{i=1}{\overset{N}{Max}}\left|b_i - \sum_{j=1}^{N}a_{ij}\phi_j^{(n)}\right| \leq \varepsilon \tag{10.68}$$

or that the root mean square residual be smaller than $\boldsymbol{\varepsilon}$, i.e.,

$$\frac{\sum_{i=1}^{N}\left(b_i - \sum_{j=1}^{N}a_{ij}\phi_j^{(n)}\right)^2}{N} \leq \varepsilon \tag{10.69}$$

Another possible criterion is for the maximum normalized difference between two consecutive iterations to drop below ε. This condition can be written as

$$\underset{i=1}{\overset{N}{Max}} \left| \frac{\phi_i^{(n)} - \phi_i^{(n-1)}}{\phi_i^{(n)}} \right| \times 100 \leq \varepsilon \tag{10.70}$$

10.3.1 Jacobi Method

Perhaps the simplest of the iterative methods for solving a linear system of equations is the Jacobi method, which is graphically presented in Fig. 10.2.

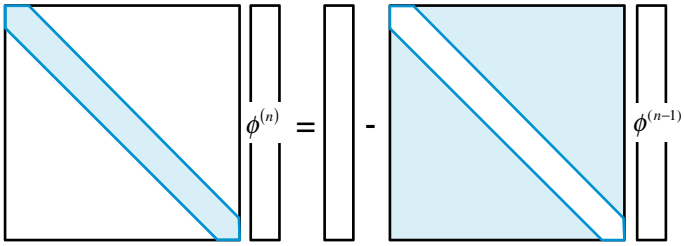


Fig. 10.2 A graphical representation of the Jacobi method

Considering the system of equations described by Eq. (10.1), if the diagonal elements are nonzero, then the first equation can be used to solve for ϕ_1, the second equation to solve for ϕ_2, and so on. The solution process starts by assigning guessed values to the unknown vector $\boldsymbol{\phi}$. These guessed values are used to calculate new estimates starting with ϕ_1, then ϕ_2, and computations proceed until a new estimate for ϕ_N is computed. This represents one iteration. Results obtained are treated as a new guess for the next iteration and the solution process is repeated. Iterations continue until the changes in the predictions between two consecutive iterations drop below a vanishing value or until a preset convergence criterion is satisfied. Once this happens the final solution is reached. In this method, given some current estimate $\phi^{(n-1)}$, an update is obtained using the following relation:

$$\phi_j^{(n)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} \phi_j^{(n-1)} \right) \quad i = 1, 2, 3, \ldots, N \tag{10.71}$$

Equation (10.71) indicates that values obtained during an iteration are not used in the subsequent calculations during the same iteration but rather retained for the next iteration. Using matrices, the expanded form of Eq. (10.71) is given by

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 & 0 \\ 0 & a_{22} & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \dots & 0 & a_{N-1,N-1} & 0 \\ 0 & 0 & \dots & 0 & a_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & 0 & \dots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N-1,1} & a_{N-1,2} & \dots & 0 & a_{N-1,N} \\ a_{N1} & a_{N2} & a_{N3} & \dots & 0 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} \tag{10.72}$$

Solving for $\boldsymbol{\phi}^{(n)}$, Eq. (10.72) yields

$$\begin{bmatrix} \phi_1^{(n)} \\ \phi_2^{(n)} \\ \vdots \\ \phi_{N-1}^{(n)} \\ \phi_N^{(n)} \end{bmatrix} = \begin{bmatrix} a_{11} & 0 & \dots & 0 & 0 \\ 0 & a_{22} & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \dots & 0 & a_{N-1,N-1} & 0 \\ 0 & 0 & \dots & 0 & a_{NN} \end{bmatrix}^{-1} \left(\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} - \begin{bmatrix} 0 & a_{12} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & 0 & \dots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N-1,1} & a_{N-1,2} & \dots & 0 & a_{N-1,N} \\ a_{N1} & a_{N2} & a_{N3} & \dots & 0 \end{bmatrix} \begin{bmatrix} \phi_1^{(n-1)} \\ \phi_2^{(n-1)} \\ \vdots \\ \phi_{N-1}^{(n-1)} \\ \phi_N^{(n-1)} \end{bmatrix} \right) \tag{10.73}$$

Using Eqs. (10.46) and (10.73) can be more concisely written as

$$\phi^{(n)} = -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\phi^{(n-1)} + \mathbf{D}^{-1}\mathbf{b} \tag{10.74}$$

The Jacobi method converges as long as $\rho\left(-\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\right) < 1$. This condition is satisfied for a large class of matrices including diagonally dominant ones where their coefficients satisfy

$$\sum_{\substack{j=1 \\ j\neq i}}^{N} |a_{ij}| \leq |a_{ii}| \quad i = 1, 2, 3, \ldots, N \tag{10.75}$$

10.3.2 Gauss-Seidel Method

A more popular take on the Jacobi is the Gauss-Seidel method, which has better convergence characteristics. It is somewhat less expensive memory-wise since it does not require storing the new estimates in a separate array. Rather, it uses the latest available estimate of ϕ in its calculations. The iterative formula in the Gauss Seidel method, schematically displayed in Fig. 10.3, is given as

$$\phi_i^{(n)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\phi_j^{(n)} - \sum_{j=i+1}^{N} a_{ij}\phi_j^{(n-1)} \right) \quad i = 1, 2, 3, \ldots, N \tag{10.76}$$

In matrix form Eq. (10.76) is written as

$$\boldsymbol{\phi}^{(n)} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\boldsymbol{\phi}^{(n-1)} + (\mathbf{D} + \mathbf{L})^{-1}\mathbf{b} \tag{10.77}$$

In effect the Gauss-Seidel method uses the most recent values in its iteration, specifically all $\phi_j^{(n)}$ values for $j < i$ since by the time ϕ_i is to be calculated, the values of $\phi_1, \phi_2, \phi_3, \ldots, \phi_{i-1}$ at the current iteration are already calculated. This approach also saves memory since the newer value is always overwriting the previous one. The Gauss-Seidel iterations converge as long as

$$\rho\left(-(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\right) < 1 \tag{10.78}$$

Although in some cases the Jacobi method converges faster, Gauss-Seidel is the preferred method.

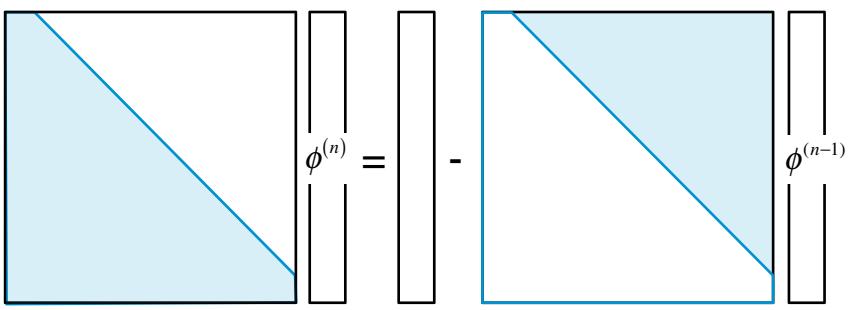


Fig. 10.3 A graphical representation of the Gauss-Seidel method

Example 2

Apply 5 iterations of the Gauss-Seidel and Jacobi methods to the system of equations in Example 1 and compute the errors at each iteration using the exact solution.

$$\boldsymbol{\phi} = \left[\frac{435}{299} \quad \frac{408}{299} \quad \frac{382}{299} \quad -\frac{19}{299}\right].$$

As an initial guess start with the field $\boldsymbol{\phi}^* = [0 \quad 0 \quad 0 \quad 0]$

Solution

Denoting with a superscript $(^*)$ values from the previous iteration, the equations to be solved in the Jacobi method are as follows:

$$\begin{aligned}
\phi_1 &= \frac{1}{3}\left(\phi_2^* + 3\right) \\
\phi_2 &= \frac{1}{6}\left(2\phi_1^* + \phi_3^* + 4\right) \\
\phi_3 &= \frac{1}{6}\left(2\phi_2^* + \phi_4^* + 5\right) \\
\phi_4 &= \frac{1}{7}\left(2\phi_3^* - 3\right)
\end{aligned}$$

with the error given as $\boldsymbol{\varepsilon} = \left|\boldsymbol{\phi}_{exact} - \boldsymbol{\phi}_{computed}\right|$. The solution for the first iteration is obtained as

$$\begin{aligned}
\phi_1 &= \frac{1}{3}(0+3) = 1 \Rightarrow \varepsilon_1 = |1.4548 - 1| = 0.4548 \\
\phi_2 &= \frac{1}{6}(0+0+4) = 0.6667 \Rightarrow \varepsilon_2 = |1.3645 - 0.6667| = 0.6978 \\
\phi_3 &= \frac{1}{6}(0+0+5) = 0.8333 \Rightarrow \varepsilon_3 = |1.2776 - 0.8333| = 0.4443 \\
\phi_4 &= \frac{1}{7}(0-3) = -0.4286 \Rightarrow \varepsilon_4 = |-0.06354 + 0.4286| = 0.3650
\end{aligned}$$

Computations proceed in the same manner with solution obtained treated as the new guess. The results for the first five iterations are given in Table 10.1.

Table 10.1 Summary of results obtained using the Jacobi iterative method

Iter #	ϕ_1	ε_1	ϕ_2	ε_2	ϕ_3	ε_3	ϕ_4	ε_4
0	0	1.4548	0	1.3645	0	1.2776	0	0.06354
1	1	0.4548	0.6667	0.6978	0.8333	0.4443	−0.4286	0.3650
2	1.2222	0.2326	1.1389	0.2257	0.9841	0.2935	−0.1905	0.1269
3	1.3796	7.52E−02	1.2381	0.1265	1.1812	9.64E−02	−0.1474	8.38E−02
4	1.4127	4.22E−02	1.3234	4.11E−02	1.2215	5.61E−02	−9.11E−02	2.75E−02
5	1.4411	1.37E−02	1.3411	2.34E−02	1.2593	1.83E−02	−7.96E−02	1.6E−02

Denoting with a superscript $(^*)$ values from the previous iteration, the equations to be solved in the Gauss-Seidel method are as follows:

$$\phi_1 = \frac{1}{3}\left(\phi_2^* + 3\right)$$
$$\phi_2 = \frac{1}{6}\left(2\phi_1 + \phi_3^* + 4\right)$$
$$\phi_3 = \frac{1}{6}\left(2\phi_2 + \phi_4^* + 5\right)$$
$$\phi_4 = \frac{1}{7}\left(2\phi_3 - 3\right)$$

with the error given as $\boldsymbol{\varepsilon} = \left|\boldsymbol{\phi}_{exact} - \boldsymbol{\phi}_{computed}\right|$. The solution for the first iteration is obtained as

$$\phi_1 = \frac{1}{3}(0+3) = 1 \Rightarrow \varepsilon_1 = |1.4548 - 1| = 0.4548$$
$$\phi_2 = \frac{1}{6}(2*1+0+4) = 1 \Rightarrow \varepsilon_2 = |1.3645 - 1| = 0.3645$$
$$\phi_3 = \frac{1}{6}(2*1+0+5) = 1.1667 \Rightarrow \varepsilon_3 = |1.2776 - 1.1667| = 0.1109$$
$$\phi_4 = \frac{1}{7}(2*1.1667-3) = -0.09523 \Rightarrow \varepsilon_4 = |-0.06354 + 0.09523| = 0.03169$$

Computations proceed in the same manner with solution obtained treated as the new guess. The results for the first five iterations are given in Table 10.2.

Table 10.2 Summary of results obtained using the Gauss-Siedel iterative method

Iter #	ϕ_1	ε_1	ϕ_2	ε_2	ϕ_3	ε_3	ϕ_4	ε_4
0	0	1.4548	0	1.3645	0	1.2776	0	0.06354
1	1	0.4548	1	0.3645	1.1667	0.1109	−0.09523	0.03169
2	1.3333	0.1215	1.3056	5.90E−02	1.2526	2.49E−02	−7.07E−02	7.13E−03
3	1.4352	1.97E−02	1.3538	1.07E−02	1.2728	4.76E−03	−6.49E−02	1.36E−03
4	1.4513	3.57E−03	1.3626	1.98E−03	1.2767	8.88E−04	−6.38E−02	2.54E−04
5	1.4542	6.61E−04	1.3642	3.68E−04	1.2774	1.65E−04	−6.36E−02	4.72E−05

10.3.3 Preconditioning and Iterative Methods

The rate of convergence of iterative methods depends on the spectral properties of the iteration matrix **B**, which is contingent on the matrix of coefficients. Based on that an iterative method looks for a transformation of the system of equations into

an equivalent one that has the same solution, but of better spectral properties. Under these conditions the eigenvalues of the equivalent system are more clustered allowing the iterative solution to be obtained faster than with the original system. A preconditioner is defined as a matrix that effects such a transformation.

A preconditioning matrix $\mathbf{P}$ is defined such that the system

$$\mathbf{P}^{-1}\mathbf{A}\boldsymbol{\phi} = \mathbf{P}^{-1}\mathbf{b} \tag{10.79}$$

has the same solution as the original system $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$, but the spectral properties of its coefficient matrix $\mathbf{P}^{-1}\mathbf{A}$ are more conducive. In defining the preconditioner $\mathbf{P}$, the difficulty is to find a matrix that approximates $\mathbf{A}^{-1}$ and is easy to invert (i.e., to find $\mathbf{P}^{-1}$) at a reasonable cost.

Writing again Eq. (10.47), but now with $\mathbf{P}$ replacing $\mathbf{M}$ (i.e., $\mathbf{M} = \mathbf{P}$ and $\mathbf{A} = \mathbf{P} - \mathbf{N}$) the associated fixed point iteration system is given by

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= \mathbf{B}\boldsymbol{\phi}^{(n-1)} + \mathbf{C}\mathbf{b} \\ &= \mathbf{P}^{-1}\mathbf{N}\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b} \\ &= \mathbf{P}^{-1}(\mathbf{P} - \mathbf{A})\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b} \\ &= \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b}\end{aligned} \tag{10.80}$$

which in residual form can be written as

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b} \\ &= \boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right) \\ &= \boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{r}^{(n-1)}\end{aligned} \tag{10.81}$$

From both equations it is now clear that the iterative procedure is just a fixed-point iteration on a preconditioned system associated with the decomposition $\mathbf{A} = \mathbf{P} - \mathbf{N}$ where the spectral properties are now

$$\rho\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right) < 1 \tag{10.82}$$

By comparison, the preconditioning matrix for the Jacobi (J) and Gauss-Seidel (GS) methods are simply

$$\begin{aligned}\mathbf{P}_J &= \mathbf{D} \\ \mathbf{P}_{GS} &= \mathbf{D} + \mathbf{L}\end{aligned} \tag{10.83}$$

where $\mathbf{D}$ and $\mathbf{L}$ are respectively the diagonal and lower triangular part of matrix $\mathbf{A}$.

Thus, preconditioning is a manipulation of the original system to improve its spectral properties with the preconditioning matrix $\mathbf{P}$ used in the associated iterative procedure. As will be described in the following sections, it is possible to develop

more advanced preconditioning matrices in which the coefficients are defined in a more complex way.

10.3.4 Matrix Decomposition Techniques

The low rate of convergence of the Gauss-Seidel and Jacobi methods was the prime motivator for the development of faster iterative techniques. One approach to accelerate the convergence rate of solvers and to develop iterative methods is through the use of more advanced preconditioners. A simple, yet efficient, approach for that purpose is to perform an *incomplete* factorization of the original matrix of coefficients **A**. The stress on incomplete is essential since a complete factorization of **A** into a lower **L** and an upper triangular matrix **U** is tantamount to a direct solution and is very expensive in term of memory requirements (fill in and loss of sparsity) and computational cost.

10.3.5 Incomplete LU (ILU) Decomposition

As can be seen in Example 1, the **L** and **U** matrices result in non-zero elements at locations that were 0 in the original matrix **A** (this is known as fill-in). So if an incomplete LU (ILU) factorization of **A** is performed such that the resulting lower **L** and upper **U** matrices have the same nonzero structure as the lower and upper parts of **A**, then

$$\mathbf{A} = \mathbf{LU} + \mathbf{R} \tag{10.84}$$

where **R** is the residual of the factorization procedure. The matrices **L** and **U** being sparse (same structure as **A**) are easier to deal with then if they were obtained from a complete factorization. However, their product being an approximation to **A**, necessitates the use of an iterative solution procedure to solve the system of equations. The first step in the solution process is to rewrite Eq. (10.1) as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \Rightarrow 0 = \mathbf{b} - \mathbf{A}\boldsymbol{\phi} \Rightarrow (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} + (\mathbf{b} - \mathbf{A}\boldsymbol{\phi}) \tag{10.85}$$

Denoting values obtained from the previous iteration with a superscript $(n-1)$, and values obtained at the current iteration with superscript (n), the iterative process is obtained by rewriting Eq. (10.85) in the following form:

$$(\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n)} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n-1)} + \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right) \tag{10.86}$$

Therefore values of $\boldsymbol{\phi}^{(n)}$ at the current iteration can be obtained from knowledge of $\boldsymbol{\phi}^{(n-1)}$ values obtained at the previous iteration. Equation (10.86) is usually solved in residual form whereby the solution $\boldsymbol{\phi}^{(n)}$ at iteration (n) is expressed in terms of the solution $\boldsymbol{\phi}^{(n-1)}$ at iteration $(n-1)$ plus a correction $\boldsymbol{\phi}'^{(n)}$, i.e.,

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{\phi}^{(n-1)} + \boldsymbol{\phi}'^{(n)} \tag{10.87}$$

Thus Eq. (10.86) becomes

$$(\mathbf{A} - \mathbf{R})\boldsymbol{\phi}'^{(n)} = \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right) \tag{10.88}$$

Once $\boldsymbol{\phi}'^{(n)}$ is found, Eq. (10.87) is used to update $\boldsymbol{\phi}$ at every iteration.

The ILU factorization can be performed using Gaussian elimination while dropping some non diagonal elements at preset locations. The locations where elements are to be dropped give rise to different ILU approximations.

10.3.6 Incomplete LU Factorization with no Fill-in ILU(0)

Many variants of the ILU factorization technique exist and the simplest is the one denoted by ILU(0) [15–17]. In ILU(0) the pattern of zero elements in the combined **L** and **U** matrices is taken to be precisely the pattern of zero elements in the original matrix **A**. Using Gaussian elimination, computations are performed as in the case of a full LU factorization, but any new nonzero element (ℓ_{ij} and u_{ij}) arising in the process is dropped if it appears at a location where a zero element exists in the original matrix **A**. Hence, the combined **L** and **U** matrices have together the same number of non zeros as the original matrix **A**. With this approach, the fill-in problem that usually arises when factorizing sparse matrices (i.e., the creation of nonzero elements at locations where the original matrix has zeros) is eliminated. In the process however, the accuracy is reduced thereby increasing the number of required iterations for convergence to be reached. To remedy this shortcoming, more accurate ILU factorization methods, which are often more efficient and more reliable, have been developed. These methods, differing by the level of fill-in allowed, are denoted by ILU(p) where p represents the order of fill-ins. The higher the level of fill-ins, the more expensive the ILU decomposition step becomes. Moreover, when used within a multigrid approach (to be explained in a later section), the ILU(0) method is more than adequate as a smoother. For this reason higher level methods are not presented and for more information interested readers are referred, among others, to the book by Saad [12].

An ILU(0) factorization algorithm which assumes **L** to be a unit lower triangular matrix and for which the same matrix **A** is used to store the elements of the unit lower and upper triangular matrices **L** and **U** is as given next.

10.3.7 ILU(0) Factorization Algorithm

$$
\begin{array}{l}
For\ k = 1\ to\ N-1 \\
\quad \{ \\
\quad\quad For\ i = k+1\ to\ N\ and\ if\ a_{ik} \neq 0\ \ Do: \\
\quad\quad\quad \{ \\
\quad\quad\quad\quad a_{ik} = \dfrac{a_{ik}}{a_{kk}}\ \ (\ell\ values) \\
\quad\quad\quad\quad\quad \{ \\
\quad\quad\quad\quad\quad\quad For\ j = k+1\ to\ N\ and\ if\ a_{ij} \neq 0\ Do: \\
\quad\quad\quad\quad\quad\quad a_{ij} = a_{ij} - a_{ik} * a_{kj}\ \ (u\ values) \\
\quad\quad\quad\quad\quad \} \\
\quad\quad\quad \} \\
\quad \}
\end{array}
$$

It should be mentioned that the ILU decomposition of symmetric positive definite matrices is denoted by *Incomplete Cholesky* decomposition. In this case the factorization is made just of the lower (or upper) triangular part and the approximation to the original matrix is written as

$$\overline{\mathbf{L}}\,\overline{\mathbf{L}}^{\mathrm{T}} \approx \mathbf{A} \tag{10.89}$$

where $\bar{\mathbf{L}}$ is the factorized sparse lower triangular matrix (approximation of $\mathbf{L}$), with the preconditioning matrix $\mathbf{P}$ given by

$$\mathbf{P} = \overline{\mathbf{L}}\,\overline{\mathbf{L}}^{\mathrm{T}} \approx \mathbf{A} \tag{10.90}$$

10.3.8 ILU Factorization Preconditioners

A very popular class of preconditioners is based on incomplete factorizations. In the discussions of direct methods it was shown that decomposing a sparse matrix $\mathbf{A}$ into the product of a lower and an upper triangular matrices may lead to substantial fill-in. Because a preconditioner is only required to be an approximation to $\mathbf{A}^{-1}$, it is sufficient to look for an approximate decomposition of $\mathbf{A}$ such that $\mathbf{A} \approx \overline{\mathbf{L}}\,\overline{\mathbf{U}}$. Choosing $\mathbf{P} = \overline{\mathbf{L}}\,\overline{\mathbf{U}}$ leads also to an efficient evaluation of the inverse of the preconditioned matrix $\mathbf{P}^{-1}$ since the inversion can easily be performed by the forward

and backward substitution, as described above, in which the exact **L** and **U** are now replaced by the approximations $\overline{\mathbf{L}}$ and $\overline{\mathbf{U}}$, respectively.

For the ILU(0) method, the incomplete factorization mimics the nonzero elements sparsity of the original matrix such that the pre-conditioner has exactly the size of the original matrix. In order to reduce the storage needed, Pommerell introduced a simplified version of the ILU called diagonal ILU (DILU) [18]. In the DILU the fill-in of the off-diagonal elements is eliminated (i.e., the upper and lower parts of the matrix are kept unchanged) and only the diagonal elements are modified.

In this case it is possible to write the preconditioner in the form

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}(\mathbf{D}^* + \mathbf{U}) \tag{10.91}$$

where **L** and **U** are the lower and upper triangular decomposition of **A**, and $\mathbf{D}^*$ is now a proper diagonal matrix, different from the diagonal of **A**. The $\mathbf{D}^*$ matrix is thus defined, as shown below, in a way that the diagonal of the product of the matrices in Eq. (10.91) equals the diagonal of **A**.

10.3.9 Algorithm for the Calculation of $\mathbf{D}^*$ in the DILU Method

For $i = 1$ *to* N *Do*:

{

$d_{ii} = a_{ii}$

}

For $i = 1$ *to* N *Do*:

{

For $j = i+1$ *to* N *and if* $a_{ij} \neq 0, a_{ji} \neq 0$ *Do*:

{

$$d_{jj} = d_{jj} - \frac{a_{ji}}{d_{ii}} * a_{ij}$$

}

}

In this case the inverse of the preconditioner, which is defined as

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}(\mathbf{D}^* + \mathbf{U}) = \overline{\mathbf{L}}\,\overline{\mathbf{U}}, \quad \overline{\mathbf{L}} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}, \quad \overline{\mathbf{U}} = (\mathbf{D}^* + \mathbf{U})$$

or

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\left(\mathbf{I} + \mathbf{D}^{*-1}\mathbf{U}\right) = \overline{\mathbf{L}}\,\overline{\mathbf{U}}, \quad \overline{\mathbf{L}} = (\mathbf{D}^* + \mathbf{L}), \quad \overline{\mathbf{U}} = \left(\mathbf{I} + \mathbf{D}^{*-1}\mathbf{U}\right) \tag{10.92}$$

needed in the solution of $\mathbf{P}\boldsymbol{\phi}'^{(n+1)} = \mathbf{r}^{(n)}$ to find the correction field $\boldsymbol{\phi}'^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n)}$, can easily be calculated using the following forward and backward substitution algorithm.

10.3.10 Forward and Backward Solution Algorithm with the DILU Method

For $i = 1$ *to* N *Do*:
{
  For $j = 1$ *to* $i-1$ *Do*:
  {
    $t_i = d_{ii}^{-1}(r_i - \ell_{ij} * t_j)$
  }
}
For $i = N$ *to* 1 *Do*:
{
  For $j = i+1$ *to* N *Do*:
  {
    $\phi'_i = t_i - d_{ii}^{-1}(u_{ij} * t_j)$
  }
}

The clear advantage of the DILU, apart from its recursive formulation, is that it requires only one extra diagonal of storage.

10.3.11 Gradient Methods for Solving Algebraic Systems

Another group of iterative procedures for solving linear algebraic systems of equations is the Gradient Methods, which include the Steepest Descent and the Conjugate Gradient methods. They were initially developed for cases where the

coefficient matrix $\mathbf{A}$ is symmetric positive definite (SPD) to reformulate the problem as a minimization problem for the quadratic vector function $\mathbf{Q}(\boldsymbol{\phi})$ given by

$$\mathbf{Q}(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} + \mathbf{c} \tag{10.93}$$

where $\mathbf{c}$ is a vector of scalars, and other variables are as defined in Eq. (10.1). The minimum of $\mathbf{Q}(\boldsymbol{\phi})$ is obtained when its gradient with respect to $\boldsymbol{\phi}$ is zero. The gradient $\mathbf{Q}'(\boldsymbol{\phi})$ of a vector field $\mathbf{Q}(\boldsymbol{\phi})$, at a given $\boldsymbol{\phi}$, points in the direction of greatest increase of $\mathbf{Q}(\boldsymbol{\phi})$. Through mathematical manipulations the gradient is found as

$$\mathbf{Q}'(\boldsymbol{\phi}) = \frac{1}{2}\mathbf{A}^{\mathrm{T}}\boldsymbol{\phi} + \frac{1}{2}\mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.94}$$

If $\mathbf{A}$ is symmetric $\left(\mathbf{A} = \mathbf{A}^{\mathrm{T}}\right)$, then Eq. (10.94) implies

$$\mathbf{Q}'(\boldsymbol{\phi}) = \mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.95}$$

The minimum is obtained when $\mathbf{Q}'(\boldsymbol{\phi}) = 0$, leading to

$$\mathbf{Q}'(\boldsymbol{\phi}) = 0 \Rightarrow \mathbf{A}\boldsymbol{\phi} = \mathbf{b} \tag{10.96}$$

Therefore minimizing $\mathbf{Q}(\boldsymbol{\phi})$ is equivalent to solving Eq. (10.1) and the solution of the minimization problem yields the solution of the system of linear equations.

Now for the function $\mathbf{Q}(\boldsymbol{\phi})$ to have a global minimum it is necessary for the coefficient matrix $\mathbf{A}$ to be positive definite, i.e., it should satisfy the inequality $\boldsymbol{\phi}^{T}\mathbf{A}\boldsymbol{\phi} > 0$ for all $\boldsymbol{\phi} \neq \mathbf{0}$. This requirement can be established by considering the relationship between the exact solution $\boldsymbol{\phi}$ and its current estimate $\boldsymbol{\phi}^{(n)}$. If $\mathbf{e} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi}$ denotes the difference between the exact solution and the current estimate, then Eq. (10.93) gives

$$\begin{aligned}
\mathbf{Q}(\boldsymbol{\phi}+\mathbf{e}) &= \frac{1}{2}(\boldsymbol{\phi}+\mathbf{e})^{\mathrm{T}}\mathbf{A}(\boldsymbol{\phi}+\mathbf{e}) - \mathbf{b}^{\mathrm{T}}(\boldsymbol{\phi}+\mathbf{e}) + \mathbf{c} \\
&= \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} + \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\mathbf{e} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\mathbf{e} + \mathbf{c} \\
&= \underbrace{\frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} + \mathbf{c}}_{\mathbf{Q}(\boldsymbol{\phi})} + \frac{1}{2}\left(\underbrace{\mathbf{e}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi}}_{\mathbf{e}^{\mathrm{T}}\mathbf{b}=\mathbf{b}^{\mathrm{T}}\mathbf{e}} + \underbrace{\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\mathbf{e}}_{\mathbf{b}^{\mathrm{T}}\mathbf{e}}\right) - \mathbf{b}^{\mathrm{T}}\mathbf{e} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e} \\
&= \mathbf{Q}(\boldsymbol{\phi}) + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e}
\end{aligned} \tag{10.97}$$

indicating that if $\mathbf{A}$ is positive definite, the second term will be always positive except when $\mathbf{e} = \mathbf{0}$, in which case the required solution would have been obtained. Moreover, when $\mathbf{A}$ is positive definite, all its eigenvalues are positive and the function $\mathbf{Q}(\boldsymbol{\phi})$ has a unique minimum.

Thus with a **symmetric** and **positive definite** matrix a converging series of $\boldsymbol{\phi}^{(n)}$ can be derived such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\left(\boldsymbol{\delta\phi}^{(n)}\right) \tag{10.98}$$

where $\alpha^{(n)}$ is some relaxation factor, and $\boldsymbol{\delta\phi}^{(n)}$ is related to the correction needed to minimize the said function at each iteration. This can be accomplished in a variety of ways leading to different methods.

10.3.12 The Method of Steepest Descent

The method of steepest descent for solving linear systems of equations of the form given by Eq. (10.1) is based on minimizing the quadratic form given by Eq. (10.93). If $\boldsymbol{\phi}$ is a one dimensional vector with its components given by the scalar ϕ, then $\boldsymbol{Q}(\phi)$ will represent a parabola. Finding the minimum of a parabolic function iteratively starting at some point ϕ_0, involves moving down along the parabola until hitting the minimum.

The same idea is used in N dimensions. In this case $\mathbf{Q}(\boldsymbol{\phi})$ may be depicted as a paraboloid and the solution is iteratively found starting from an initial position $\boldsymbol{\phi}^{(0)}$ and moving down the paraboloid until the minimum is reached. For quick convergence the sequence of steps $\boldsymbol{\phi}^{(0)}, \boldsymbol{\phi}^{(1)}, \boldsymbol{\phi}^{(2)}, \ldots$ should be selected such that the fastest rate of descent occurs, i.e., in the direction of $-\mathbf{Q}'(\boldsymbol{\phi})$. According to Eq. (10.95) this direction is also given by

$$-\mathbf{Q}'(\boldsymbol{\phi}) = \mathbf{b} - \mathbf{A}\boldsymbol{\phi} \tag{10.99}$$

The exact solution being $\boldsymbol{\phi}$, the error and residual at any step n, denoted respectively by $\mathbf{e}^{(n)}$ and $\mathbf{r}^{(n)}$, are computed as

$$\left.\begin{aligned} \mathbf{e}^{(n)} &= \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi} \\ \mathbf{r}^{(n)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)} &= -\mathbf{Q}'\left(\boldsymbol{\phi}^{(n)}\right) \end{aligned}\right\} \Rightarrow \mathbf{r}^{(n)} = -\mathbf{A}\mathbf{e}^{(n)} \tag{10.100}$$

Moving linearly in the direction of the steepest descent, the value of $\boldsymbol{\phi}$ at step $n + 1$ can be expressed in terms of the value of $\boldsymbol{\phi}$ at step n according to

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)} \tag{10.101}$$

The value of $\alpha^{(n)}$ that minimizes $\mathbf{Q}(\boldsymbol{\phi})$ should satisfy

$$\frac{d}{d\alpha^{(n)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right) = 0 \tag{10.102}$$

This can be expanded into

$$\frac{d}{d\alpha^{(n)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right) = 0 \Rightarrow \left[\frac{d}{d\boldsymbol{\phi}^{(n+1)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right)\right]^{\mathrm{T}}\frac{d\boldsymbol{\phi}^{(n+1)}}{d\alpha^{(n)}} = 0 \Rightarrow \left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \tag{10.103}$$

indicating that the new step should be in a direction normal to the old step. The value of $\alpha^{(n)}$ is calculated by using Eq. (10.103) as follows:

$$\begin{aligned}
&\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \Rightarrow \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \\
&\Rightarrow \left[\mathbf{b} - \mathbf{A}\left(\boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}\right)\right]^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \\
&\Rightarrow \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = \alpha^{(n)}\left(\mathbf{A}\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} \\
&\Rightarrow \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = \alpha^{(n)}\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)} \\
&\Rightarrow \alpha^{(n)} = \frac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)}}
\end{aligned} \tag{10.104}$$

The steepest descent algorithm can be summarized as follows:

$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose residual as starting direction)
iterate starting at (n) until convergence
$\mathbf{r}^{(n)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)}$ (Compute the residual vector)
$\alpha^{(n)} = \dfrac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)}}$ (Compute the factor in the orthogonal direction)
$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

As presented above, the algorithm necessitates performing two matrix-vector multiplications per iteration. One of them can be eliminated by multiplying both sides of Eq. (10.101) by $-\mathbf{A}$ and adding $\mathbf{b}$ to obtain

$$\begin{aligned}\boldsymbol{\phi}^{(n+1)} &= \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)} \Rightarrow \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n+1)} \\ &= \mathbf{b} - \mathbf{A}\left(\boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}\right) \Rightarrow \mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{r}^{(n)}\end{aligned} \tag{10.105}$$

The equation for $\mathbf{r}^{(n)}$ in step 1 is needed only to calculate $\mathbf{r}^{(0)}$, while Eq. (10.105) can be used afterwards. With this formulation there will be no need to compute $\mathbf{A}\boldsymbol{\phi}^{(n)}$, as it was replaced by $\mathbf{A}\mathbf{r}^{(n)}$. However a shortcoming of this approach, is the lack of feedback from the value of $\boldsymbol{\phi}^{(n)}$ into the residual, which may cause the solution to converge to a value different from the exact one due to accumulation of roundoff errors. This deficiency can be resolved by periodically computing the residual using the original equation.

10.3.13 The Conjugate Gradient Method

While the steepest descent method guarantees convergence, its rate of convergence is low. This slow convergence is caused by oscillations around local minima forcing the method to search in the same direction repeatedly. To avoid this undesirable behavior every new search should be in a direction different from the directions of previous searches [19]. This can be accomplished by selecting a set of search directions $\mathbf{d}^{(0)}, \mathbf{d}^{(1)}, \mathbf{d}^{(2)}, \ldots, \mathbf{d}^{(N-1)}$ that are **A**-orthogonal. Two vectors $\mathbf{d}^{(n)}$ and $\mathbf{d}^{(m)}$ are said to be **A**-orthogonal if they satisfy the following condition:

$$\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(m)} = 0 \tag{10.106}$$

If in each search direction the right step size is taken, the solution will be found after N steps. Step $n + 1$ is chosen such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)} \tag{10.107}$$

Subtracting $\boldsymbol{\phi}$ from both sides of the above equation, an equation for the error is obtained as

$$\mathbf{e}^{(n+1)} = \mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)} \tag{10.108}$$

Combining Eq. (10.100) with Eq. (10.108), an equation for the residual is found as

$$\begin{aligned}\mathbf{r}^{(n+1)} &= -\mathbf{A}\mathbf{e}^{(n+1)} \\ &= -\mathbf{A}\left(\mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}\right) \\ &= \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}\end{aligned} \tag{10.109}$$

Equation (10.109) shows that each new residual $\mathbf{r}^{(n+1)}$ is just a linear combination of the previous residual and $\mathbf{A}\mathbf{d}^{(n)}$.

It is further required that $\mathbf{e}^{(n+1)}$ be $\mathbf{A}$-orthogonal to $\mathbf{d}^{(n)}$. This new condition is equivalent to finding the minimum point along the search direction $\mathbf{d}^{(n)}$. Using this $\mathbf{A}$-orthogonality condition between $\mathbf{e}^{(n+1)}$ and $\mathbf{d}^{(n)}$ along with Eq. (10.108) an expression for $\alpha^{(n)}$ can be derived as

$$\begin{aligned}\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{e}^{(n+1)} = 0 \Rightarrow \left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\left(\mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}\right) \\ = 0 \Rightarrow \alpha^{(n)} = \frac{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}\end{aligned} \tag{10.110}$$

The above requirement also implies that

$$\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{e}^{(n+1)} = 0 \Rightarrow \left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)} = 0 \tag{10.111}$$

If the search directions are known then $\alpha^{(n)}$ can be calculated.

To derive the search direction, it is assumed to be governed by an equation of the form

$$\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n)}\mathbf{d}^{(n)} \tag{10.112}$$

The $\mathbf{A}$-orthogonality requirement of the $\mathbf{d}$ vectors implies that

$$\left(\mathbf{d}^{(n+1)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)} = 0 \tag{10.113}$$

Substituting the value of $\mathbf{d}^{(n+1)}$ from Eq. (10.112) in Eq. (10.113) yields

$$\beta^{(n)} = -\frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}} \tag{10.114}$$

From Eq. (10.109) an expression for $\mathbf{A}\mathbf{d}^{(n)}$ is obtained as

$$\mathbf{A}\mathbf{d}^{(n)} = -\frac{1}{\alpha^{(n)}}\left(\mathbf{r}^{(n+1)} - \mathbf{r}^{(n)}\right) \tag{10.115}$$

Combining Eqs. (10.110), (10.114), and (10.115) leads to

$$\begin{aligned}\beta^{(n)} &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\left(\mathbf{r}^{(n+1)}-\mathbf{r}^{(n)}\right)}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \\ &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)} - \underbrace{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}_{=0}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \\ &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}\end{aligned} \tag{10.116}$$

The denominator of the above equation can be further expressed as

$$\begin{aligned}\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} &= \left(\mathbf{r}^{(n)}+\beta^{(n-1)}\mathbf{d}^{(n-1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} \\ &= \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} + \beta^{(n-1)}\underbrace{\left(\mathbf{d}^{(n-1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}_{=0} \\ &= \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}\end{aligned} \tag{10.117}$$

Using Eqs. (10.116) and (10.117), the final expression for $\beta^{(n)}$ is obtained as

$$\beta^{(n)} = \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \tag{10.118}$$

The Conjugate Gradient algorithm becomes

$\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose residual as starting direction)
iterate starting at (n) until convergence
$\alpha^{(n)} = \dfrac{\mathbf{d}^{(n)T}\mathbf{r}^{(n)}}{\mathbf{d}^{(n)T}\mathbf{A}\mathbf{d}^{(n)}}$ (Choose factor in $\mathbf{d}$ direction)
$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)
$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new residual)
$\beta^{(n)} = \dfrac{\mathbf{r}^{(n+1)T}\mathbf{r}^{(n+1)}}{\mathbf{r}^{(n)T}\mathbf{r}^{(n)}}$ (Calculate coefficient to conjugate residual)
$\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n)}\mathbf{d}^{(n)}$ (obtain new conjugated search direction)

The convergence rate of the CG method may be increased by preconditioning. This can be done by multiplying the original system of equations by the inverse of

the preconditioned matrix $\mathbf{P}^{-1}$, where $\mathbf{P}$ is a symmetric positive-definite matrix, to yield Eq. (10.79). The problem is that $\mathbf{P}^{-1}\mathbf{A}$ is not necessarily symmetric even if $\mathbf{P}$ and $\mathbf{A}$ are symmetric. To circumvent this problem the Cholesky decomposition is used to write $\mathbf{P}$ in the form

$$\mathbf{P} = \mathbf{L}\mathbf{L}^{\mathrm{T}} \tag{10.119}$$

To guarantee symmetry, the system of equations is written as

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-\mathrm{T}}\mathbf{L}^{\mathrm{T}}\boldsymbol{\phi} = \mathbf{L}^{-1}\mathbf{b} \tag{10.120}$$

where $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-\mathrm{T}}$ is symmetric and positive-definite. The CG method can be used to solve for $\mathbf{L}^{\mathrm{T}}\boldsymbol{\phi}$, from which $\boldsymbol{\phi}$ is found. However, by variable substitutions, $\mathbf{L}$ can be eliminated from the equations without disturbing symmetry or affecting the validity of the method. Performing this step and adopting the terminology used with the CG method, the various steps in the preconditioned CG method are obtained.

The preconditioned CG method can be summarized as follows:

$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ and $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$ (choose starting direction)

iterate starting at (n) until convergence

$$\alpha^{(n)} = \frac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}$$ (Choose factor in $\mathbf{d}$ direction)

$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new residual)

$$\beta^{(n+1)} = \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}$$ (Calculate coefficient to conjugate residual)

$\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new conjugated search direction)

Many pre-conditioners have been developed with a wide spectrum of sophistication varying from a simple diagonal matrix whose elements are the diagonal elements of the original matrix $\mathbf{A}$ (Jacobi pre-conditioner) to more involved ones using incomplete Cholesky factorization. Nonetheless, the CG method should always be used with a pre-conditioner when solving large systems of equations.

10.3.14 The Bi-conjugate Gradient Method (BiCG) and Preconditioned BICG

The matrix of coefficients resulting from the discretization of the diffusion equation presented in Chap. 8 and some other equations like the incompressible pressure or

pressure correction equation that will be presented in Chap. 15 are symmetrical leading to symmetrical systems that can be solved using the CG method discussed above. However, the matrix $\mathbf{A}$ obtained from the discretization of the general conservation equation arising in CFD applications is unsymmetrical yielding an unsymmetrical system of equations. To be able to solve this system using the CG method, it should be transformed into a symmetrical one [20]. One way to do that is to rewrite Eq. (10.1) as

$$\begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\phi}} \\ \boldsymbol{\phi} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \tag{10.121}$$

where $\widehat{\boldsymbol{\phi}}$ is a dummy variable added in order to convert the original unsymmetrical system into a symmetrical one amenable to a solution by the CG method. When applied to this system, the CG method results in two sequences of CG-like vectors, the ordinary sequence based on the original system with the coefficient matrix $\mathbf{A}$, from which $\boldsymbol{\phi}$ is calculated, and the shadow sequence for the unneeded system with the coefficient matrix $\mathbf{A}^{\mathrm{T}}$, from which $\widehat{\boldsymbol{\phi}}$ can be calculated if desired. Because of the two series of vectors the name bi-conjugate gradient (BiCG) is coined to the method. The same terminology used with the CG method is used here. The series of ordinary vectors for the residuals and search directions are denoted by $\mathbf{r}$ and $\mathbf{d}$, respectively, and their shadow equivalent forms by $\widehat{\mathbf{r}}$ and $\widehat{\mathbf{d}}$. The bi-orthogonality of the residuals is guaranteed by forming them such that

$$\left(\widehat{\mathbf{r}}^{(m)}\right)^{\mathrm{T}} \mathbf{r}^{(n)} = \left(\widehat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(m)} = 0 \quad m < n \tag{10.122}$$

and the bi-conjugacy of the search directions is fulfilled by requiring that

$$\left(\widehat{\mathbf{d}}^{(n)}\right)^{T} \mathbf{A}\mathbf{d}^{(m)} = \left(\mathbf{d}^{(n)}\right)^{T} \mathbf{A}^{T}\widehat{\mathbf{d}}^{(m)} = 0 \quad m < n \tag{10.123}$$

Moreover, the sequences of residuals and search directions are constructed such that the ordinary form of one is orthogonal to the shadow form of the other. Mathematically this is written as

$$\left(\widehat{\mathbf{r}}^{(n)}\right)^{T} \mathbf{d}^{(m)} = \left(\mathbf{r}^{(n)}\right)^{T} \widehat{\mathbf{d}}^{(m)} \quad m < n \tag{10.124}$$

Several variants of the method, which has irregular convergence with the possibility of breaking down, have been developed and the algorithm described next is due to Lanczos [21, 22].

The BiCG algorithm of Lanczos can be summarized as follows:

$\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \bar{\mathbf{d}}^{(0)} = \hat{\mathbf{r}}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose starting directions)

iterate starting at (n) until convergence

$$\alpha^{(n)} = \frac{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(n)}}{\left(\bar{\mathbf{d}}^{(n)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{d}^{(n)}}$$ (Choose factor in $\mathbf{d}$ direction)

$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new $\mathbf{r}$ residual)

$\hat{\mathbf{r}}^{(n+1)} = \hat{\mathbf{r}}^{(n)} - \alpha^{(n)}\mathbf{A}^{\mathrm{T}}\bar{\mathbf{d}}^{(n)}$ (calculate new $\hat{\mathbf{r}}$ residual)

$$\beta^{(n+1)} = \frac{\left(\hat{\mathbf{r}}^{(n+1)}\right)^{\mathrm{T}} \mathbf{r}^{(n+1)}}{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(n)}}$$ (Calculate coefficient to conjugate residual)

$\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new search $\mathbf{d}$ direction)

$\bar{\mathbf{d}}^{(n+1)} = \hat{\mathbf{r}}^{(n+1)} + \beta^{(n+1)}\bar{\mathbf{d}}^{(n)}$ (obtain new search $\bar{\mathbf{d}}$ direction)

The BiCG method necessitates a multiplication with the coefficient matrix and with its transpose at each iteration resulting in almost double the computational effort required by the CG method per iteration.

Preconditioning may also be used with the BiCG method. For the same terminology as for the preconditioned CG method, a robust variant of the method developed by Fletcher [23] is summarized next.

The preconditioned algorithm for the BiCG of Fletcher is as follows ($\mathbf{P}$ representing the preconditioning matrix):

$\mathbf{r}^{(0)} = \hat{\mathbf{r}}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$, $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$, $\bar{\mathbf{d}}^{(0)} = \mathbf{P}^{-\mathrm{T}}\hat{\mathbf{r}}^{(0)}$ (choose starting directions)

iterate starting at (n) until convergence

$$\alpha^{(n)} = \frac{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\bar{\mathbf{d}}^{(n)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{d}^{(n)}}$$ (Choose factor in $\mathbf{d}$ direction)

$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new $\mathbf{r}$ residual)

$\hat{\mathbf{r}}^{(n+1)} = \hat{\mathbf{r}}^{(n)} - \alpha^{(n)}\mathbf{A}^{\mathrm{T}}\bar{\mathbf{d}}^{(n)}$ (calculate new $\hat{\mathbf{r}}$ residual)

$$\beta^{(n+1)} = \frac{\left(\hat{\mathbf{r}}^{(n+1)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n)}}$$ (Calculate coefficient to conjugate residual)

$\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new search $\mathbf{d}$ direction)

$\bar{\mathbf{d}}^{(n+1)} = \mathbf{P}^{-\mathrm{T}}\hat{\mathbf{r}}^{(n+1)} + \beta^{(n+1)}\bar{\mathbf{d}}^{(n)}$ (obtain new search $\bar{\mathbf{d}}$ direction)

Other variants of the BiCG method that are more stable and robust have been reported such as the conjugate gradient squared (CGS) method of Sonneveld [24], the bi-conjugate gradient stabilized (Bi-CGSTAB) method of Van Der Vorst [25] and the generalized minimal residual method GMRES [13, 26–29]. These methods are useful for solving large systems of equations arising in CFD applications as they are applicable to non-symmetrical matrices and to both structured and unstructured grids.

10.4 The Multigrid Approach

The rate of convergence of iterative methods drastically deteriorates as the size of the algebraic system increases, with the drop in convergence rate even observed in medium to large systems after the initial errors have been eliminated. This has constituted a severe limitation for iterative solvers. Luckily it was very quickly found that the combination of multigrid and iterative methods can practically remedy this weakness.

Developments in multigrid methods started with the work of Fedorenko [30] (Geometric Multigrid), Poussin [31] (Algebraic Multigrid), and Settari and Azziz [32], and gained more interest with the theoretical work of Brandt [33]. While high-frequency or oscillatory errors are easily eliminated with standard iterative solvers (Jacobi, Gauss-Seidel, ILU), these solution techniques cannot easily remove the smooth or low frequency error components [34]. Because of that these solution

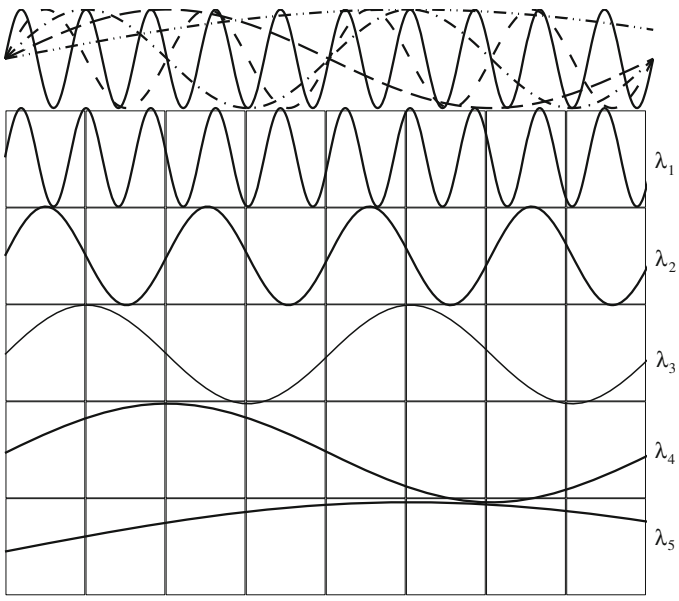


Fig. 10.4 Schematic of different error modes in a one dimensional grid

methods are denoted by smoothers in the context of multigrid methods. An illustration of error frequency is shown in Fig. 10.4 where the variations of error frequency modes for a one dimensional problem are plotted.

The error modes shown in Fig. 10.4 vary from high frequency of short wavelength λ_1 to low frequency of long wavelength λ_5 and are plotted collectively on the top of the figure. The one dimensional domain is discretized using the one dimensional grid shown and the various modes are separately plotted over the same grid. As can be seen, the high frequency error appears oscillatory over an element and is easily sensed by the iterative method. As the frequency of the error decreases or as the wavelength (λ) increases, the error becomes increasingly smoother over the grid as only a small portion of the wavelength lies within any cell. This gets worse as the grid is further refined, leading to a higher number of equations and explaining the degradation in the rate of convergence as the size of the system increases.

Multigrid methods improve the efficiency of iterative solvers by ensuring that the resulting low frequency errors that arise from the application of a smoother at any one grid level are transformed into higher frequency errors at a coarser grid level. By using a hierarchy of coarse grids (Fig. 10.5), multigrid methods are able to overcome the convergence degradation.

Generally the coarse mesh can be formed using either the topology and geometry of the finer mesh, this is akin to generating a new mesh for each coarse level on top of the finer level mesh or by direct agglomeration of the finer mesh elements [35–40]; this approach is also known as the Algebraic MultiGrid Method (AMG). In the AMG no geometric information is directly needed or used, and the agglomeration process is purely algebraic, with the equations at each coarse level reconstructed from those of the finer level, again through the agglomeration process. This approach can be used to build highly efficient and robust linear solvers

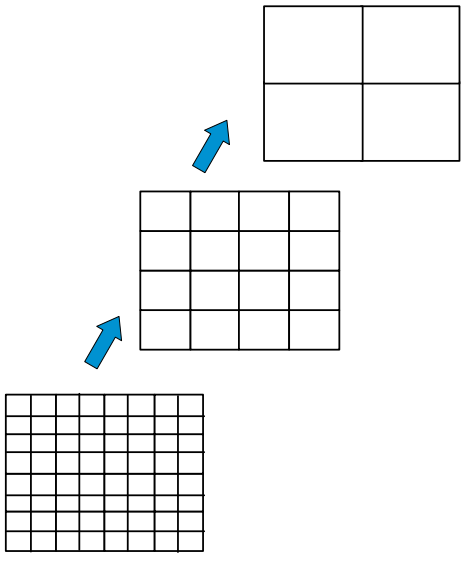


Fig. 10.5 A schematic of the hierarchy of grid systems used with the multi grid approach

for both highly anisotropic grids and/or problems with large changes in the coefficients of their equations.

In either approach, a multigrid cycling procedure is used to guide the traversal of the various grid hierarchies. Each traversal from a fine grid to a coarse one involves: (i) a restriction procedure, (ii) the setup or update of the system of equations for the coarse grid level, and (iii) the application of a number of smoother iterations. A traversal from a coarse grid to a finer one requires: (i) a prolongation procedure, (ii) the correction of the field values at the finer level, and (iii) the application of a number of smoother iterations on the equations constructed during restriction. The various steps needed are detailed next.

10.4.1 Element Agglomeration/Coarsening

The first step in the solution process is to generate the coarse/fine grid levels by an agglomeration/coarsening algorithm. Three different approaches can be adopted for that purpose. In the first approach, the coarse mesh is initially generated and the fine levels are obtained by refinement [41, 42]. This facilitates the definitions of the coarse-fine grid relations and is attractive in an adaptive grid setup [41–43]. A major drawback however, is the dependence of the fine grid distribution on the coarse grid. In the second method, non-nested grids are used [44] rendering the transfer of information between grid levels very expensive. In addition, both approaches do not allow good resolution of complex domains. In the third approach, recommended here, the process starts with the generation of the finest mesh that will be used in solving the problem. Then coarse grid levels are developed through agglomeration of the fine-grid elements [45, 46], as shown in Fig. 10.6, with the agglomeration process based either on the elements geometry or on a criterion to be satisfied by the coefficients of neighboring elements. The discussions to follow are pertinent to the third approach.

Coarse grid levels are generated by fusing fine grid elements through an agglomeration algorithm. For each coarse grid level, the algorithm is repeatedly applied until all grid cells of the finer level become associated with coarse grid cells.

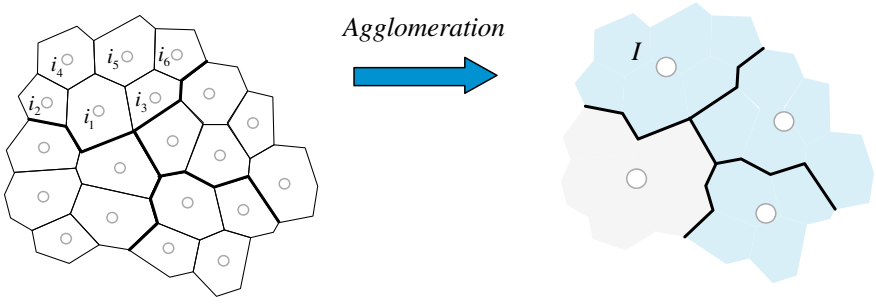


Fig. 10.6 Agglomeration of a fine grid level to form a coarse grid level

During this heuristic agglomeration process, fine grid points are individually visited. A cell is selected as the seed element into which a certain number of neighboring elements satisfying the set criteria are fused to form a coarse element. The maximum number of fine elements to be fused into a coarse element is decided a priori. If the chosen seed element fails to form a coarse element, it is added to the least populated coarse element among its neighbors.

An efficient agglomeration algorithm is the directional agglomeration (DA) algorithm developed by Mavriplis [47]. In the DA, agglomeration is performed by starting with a seed element and merging with it the neighboring fine grid elements based on the strength of their geometric connectivity. The procedure needs to be performed only once at the start of the solution.

10.4.2 The Restriction Step and Coarse Level Coefficients

The solution starts at the fine grid level. After performing few iterations, the error is transferred or restricted to a coarser grid level and the solution is found at that level. Then after performing few iterations at that level the error is restricted again to a higher level and the sequence of events repeated until the highest or coarsest grid level is reached. Let (k) denotes some level at which the solution has been found by solving the following system of equations in correction form:

$$\mathbf{A}^{(k)}\mathbf{e}^{(k)} = \mathbf{r}^{(k)} \tag{10.125}$$

The next coarser level is $(k+1)$ to which the error will be restricted. Let G_I represents the set of cells i on the fine grid level (k) that are agglomerated to form cell I of the coarse grid level $(k+1)$. Then, the system to be solved on the coarse grid at level $(k+1)$ is

$$\mathbf{A}^{(k+1)}\mathbf{e}^{(k+1)} = \mathbf{r}^{(k+1)} \tag{10.126}$$

with the residuals on the RHS of Eq. (10.126) computed as

$$\mathbf{r}^{(k+1)} = I_k^{k+1}\mathbf{r}^{(k)} \tag{10.127}$$

where I_k^{k+1} is the restriction operator (i.e., the interpolation matrix) from the fine grid to the coarse grid as defined by the agglomeration process. In AMG the restriction operator is defined in a linear manner to yield a summation of the fine grid residuals as

$$\mathbf{r}_I^{(k+1)} = \sum_{i \in G_I} \mathbf{r}_i^{(k)} \tag{10.128}$$

Moreover, the coefficients of the coarse element are constructed by adding the appropriate coefficients of the constituting fine elements. Recalling that a linear equation after discretization has the form

$$a_C \phi_C + \sum_{F=NB(C)} a_F \phi_F = b_C \tag{10.129}$$

which, for the current purpose, is written for the fine grid level in a more suitable form as

$$a_i^{(k)} \phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j^{(k)} = b_i^{(k)} \tag{10.130}$$

where $NB(i)$ refers to the neighbors of element i. Initially Eq. (10.130) is not satisfied resulting in the following residual:

$$r_i^{(k)} = b_i^{(k)} - \left(a_i^{(k)} \phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j^{(k)} \right) \tag{10.131}$$

Denoting by $\phi_I^{(k+1)}$ the solution on the coarse mesh element I that is parent to the fine mesh element i, the correction on the fine mesh from the coarse mesh can be written as

$$\phi_i'^{(k)} = \phi_I^{(k+1)} - \phi_i^{(k)} \tag{10.132}$$

It is desired for the correction to result in zero residuals over the coarse mesh element I. These new residuals denoted by $\tilde{r}_i^{(k)}$ are calculated as

$$\tilde{r}_i^{(k)} = b_i^{(k)} - \left(a_i^{(k)} \left(\phi_i^{(k)} + \phi_i'^{(k)} \right) + \sum_{j=NB(i)} a_{ij}^{(k)} \left(\phi_j^{(k)} + \phi_j'^{(k)} \right) \right) \tag{10.133}$$

or equivalently as

$$\begin{aligned} \tilde{r}_i^{(k)} &= \underbrace{b_i^{(k)} - \left(a_i^{(k)} \phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j^{(k)} \right)}_{r_i^{(k)}} - \left(a_i^{(k)} \phi_i'^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j'^{(k)} \right) \\ &= r_i^{(k)} - \left(a_i^{(k)} \phi_i'^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j'^{(k)} \right) \end{aligned} \tag{10.134}$$

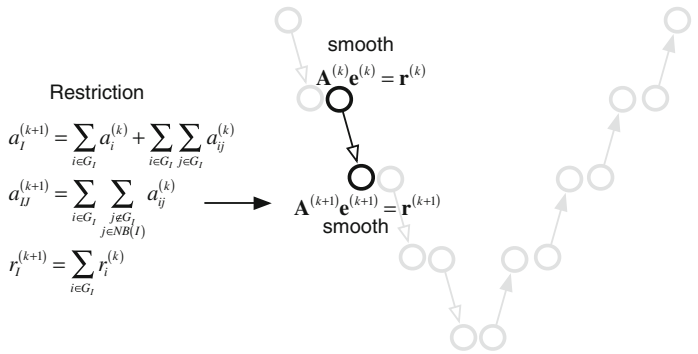


Fig. 10.7 The restriction step and assembly of coarse grid level coefficients

Enforcing the residual sum in I to zero, i.e.,

$$\sum_{i\in G_I} \tilde{\boldsymbol{r}}_i^{(k)} = 0 \tag{10.135}$$

and substituting Eq. (10.134) in Eq. (10.135) yields

$$0 = \sum_{i\in G_I} \boldsymbol{r}_i^{(k)} - \left(\sum_{i\in G_I} a_i^{(k)} \phi_i'^{(k)} + \sum_{i\in G_I}\sum_{j=NB(i)} a_{ij}^{(k)} \phi_j'^{(k)} \right) \tag{10.136}$$

Rewriting Eq. (10.136) using coarse mesh numbering, the coarse mesh correction equation becomes

$$a_I^{(k+1)} \phi_I'^{(k+1)} + \sum_{J=NB(I)} a_{IJ}^{(k+1)} \phi_J'^{(k+1)} = r_I^{(k+1)} \tag{10.137}$$

where $a_I^{(k+1)}, a_{IJ}^{(k+1)}$, and $r_I^{(k+1)}$ are derived directly from fine grid coefficients as

$$\begin{aligned} a_I^{(k+1)} &= \sum_{i\in G_I} a_i^{(k)} + \sum_{i\in G_I}\sum_{j\in G_I} a_{ij}^{(k)} \\ a_{IJ}^{(k+1)} &= \sum_{i\in G_I} \sum_{\substack{j\notin G_I \\ j\in NB(I)}} a_{ij}^{(k)} \\ r_I^{(k+1)} &= \sum_{i\in G_I} r_i^{(k)} \end{aligned} \tag{10.138}$$

This is illustrated in Fig. 10.7.

10.4.3 The Prolongation Step and Fine Grid Level Corrections

The prolongation operator is used to transfer the correction from a coarse to a fine grid level. Many options may be used. One possibility, depicted in Fig. 10.8, is a zero order prolongation operator that yields the same value of the error on the fine grid, i.e., the error at a coarse grid cell will be inherited by all the children of this cell on the fine grid level.

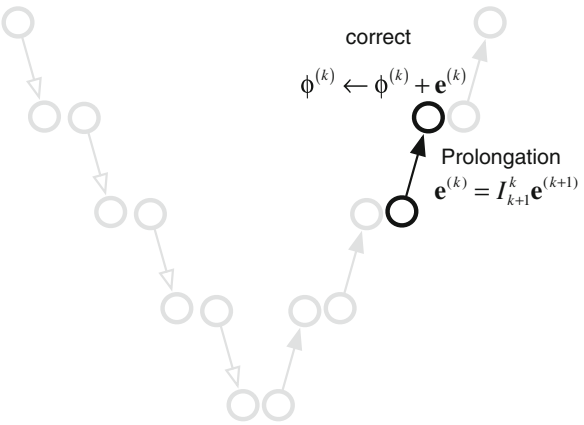


Fig. 10.8 The prolongation step and fine grid level corrections

The correction is basically obtained from the solution of the system of equations at the coarse grid. The interpolation or prolongation to the fine grid level is denote as

$$\mathbf{e}^{(k)} = I_{k+1}^{k}\mathbf{e}^{(k+1)} \tag{10.139}$$

where I_{k+1}^{k} is an interpolation matrix from the coarse grid to the fine grid. Finally, the fine grid solution is corrected as

$$\boldsymbol{\phi}^{(k)} \leftarrow \boldsymbol{\phi}^{(k)} + \mathbf{e}^{(k)} \tag{10.140}$$

The number of grid levels used depends on the size of the grid. For a larger number of grid levels, the procedure is the same as sketched in Fig. 10.8.

10.4.4 Traversal Strategies and Algebraic Multigrid Cycles

Traversal strategies refer to the way by which coarse grids are visited during the solution process, which are also known as multigrid cycles [48]. The usual cycles used in the AMG method are the V cycle, the W cycle, and the F cycle [35, 36, 49, 50] displayed in Fig. 10.9.

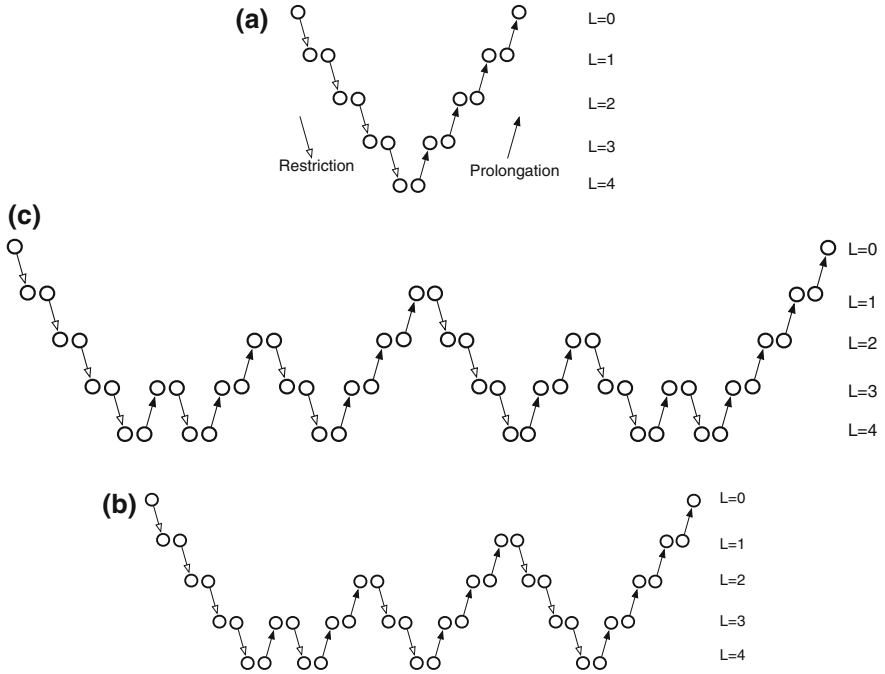


Fig. 10.9 **a** V, **b** W, and **c** F multigrid cycles

The simplest AMG cycle, illustrated in Fig. 10.9a, is the V cycle [49, 50] and consists of visiting each of the grid levels only once. The usual practice is to perform few iterative sweeps in the restriction phase and then to inject the residual to a coarser grid until reaching the coarsest level. For very stiff systems, the V cycle may not be sufficient for accelerating the solution and therefore, more iterations on the coarse level are required. The W cycle is based on applying smaller V cycles on each visited coarse grid level. In this manner, the W cycle (Fig. 10.9b) consists of nested coarse and fine grid level sweeps with the complexity increasing as the number of AMG levels increases. The F cycle is a variant of the W cycle and can be thought of as splitting the W cycle in half as shown in Fig. 10.9c. The F cycle requires less coarse level sweeps than the W cycle but more sweeps than the V cycle. Therefore, it lies in between the V and W cycling strategies.

10.5 Computational Pointers

10.5.1 uFVM

In uFVM, two linear algebraic solvers are implemented. The successive over-relaxation method (SOR) and the ILU(0) method. The implementation of these

methods follows the procedures described earlier. The SOR is located in the file "cfdSORSolver.m" while the ILU(0) implementation can be found in "cfdILUSolver.m".

10.5.2 OpenFOAM®

The organizational structure of iterative linear algebraic solvers in OpenFOAM® [51] follows the usual approach. It starts by defining the base classes from which each type of algebraic matrix solvers is established. These algebraic solvers are grouped under three main categories denoted by solvers, preconditioners, and smoothers. Smoothers and preconditioners are differentiated by relating to smoothers the fixed point relation and embedding them within the preconditioners framework. Recalling Eq. (10.81), the preconditioners classes implement the product $\mathbf{P}^{-1}\mathbf{r}$ while the smoothers classes advance the solution. Moreover, the solvers category collects the necessary information related to the implementation of the conjugate gradient and multigrid algorithms.

The source codes of the linear algebraic solvers reside within the lduMatrix folder located at "…/src/OpenFOAM/matrices/lduMatrix/" in the following three subfolders:

- *solvers*
- *preconditioners*
- *smoothers*

The name of each subfolder reflects its functionality. The folder *solvers* contains the main codes of the iterative solvers implemented in OpenFOAM®, which are

- **diagonalSolver**: a diagonal solver for both symmetric and asymmetric problems.
- **GAMG**: a geometric agglomerated algebraic multigrid solver (also named Generalized geometric- algebraic multi-grid in the manual).
- **ICC**: an incomplete Cholesky preconditioned conjugate gradient solver.
- **PBiCG**: a preconditioned bi-conjugate gradient solver for asymmetric matrices.
- **PCG**: a preconditioned conjugate gradient solver for symmetric matrices.
- **smoothSolver**: an iterative solver using smoother for symmetric and asymmetric matrices based on preconditioners.

The folder *preconditioners* contains various implementations of the diagonal ILU denoted by

- **diagonalPreconditioner**: a diagonal preconditioner.
- **DICPreconditioner, DILUPreconditioner**: a diagonal Incomplete Cholesky preconditioner for symmetric and asymmetric matrices respectively.
- **FDICPreconditioner**: a faster version of the DICPreconditioners diagonal-based incomplete Cholesky preconditioner for symmetric matrices in which the reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored.

- **GAMGPreconditioner**: a geometric agglomerated algebraic multigrid preconditioner. It uses a mutigrid cycle as preconditioner to execute the second part of Eq. (10.81).
- **noPreconditioner**: a null preconditioner for both symmetric and asymmetric matrices.

Finally the *smoothers* folder contains the following:

- **DIC**, **DILU**: a diagonal-based incomplete Cholesky smoother for symmetric and asymmetric matrices.
- **DICGaussSeidel**, **DILUGaussSeidel**: a combined DIC, DILU/Gauss-Seidel smoother for symmetric and asymmetric matrices in which DIC, DILU smoothing is followed by Gauss-Seidel to ensure that any "spikes" created by the DIC, DILU sweeps are smoothed out.
- **DILU**: a diagonal-based incomplete LU smoother for asymmetric matrices.
- **GaussSeidel**: The Gauss-Seidel method for both symmetric and asymmetric matrices.

Furthermore OpenFOAM® defines inside the lduMatrix class three additional base classes that wrap the three corresponding categories. Thus the lduMatrix.H file reads (Listing 10.1)

```
class lduMatrix
{
    // private data

        //- LDU mesh reference
        const lduMesh& lduMesh_;

        //- Coefficients (not including interfaces)
        scalarField *lowerPtr_, *diagPtr_, *upperPtr_;

…
public:

    //- Abstract base-class for lduMatrix solvers
    class solver
    {
    protected:
…

    class smoother
    {
    protected:
…
    class preconditioner
    {
    protected:
…
```

Listing 10.1 The three base classes (solver, smoother, and preconditioner) defined with the lduMatrix class

So each smoother, solver, and preconditioner has to be derived from these three base classes. For example, the DILU preconditioner is declared as shown in Listing 10.2,

```
class DILUPreconditioner
:
    public lduMatrix::preconditioner
{
…
```

Listing 10.2 Syntax used to declare the DILU preconditioner

while the Conjugate Gradient solver is declared as shown in Listing 10.3.

```
class PCG
:
    public lduMatrix::solver
```

Listing 10.3 Syntax used to declare the PCG solver

In either case the preconditioner or the solver is evidently derived from the base class defined under the lduMatrix class.

Having clarified the basic concepts and organizational structure of algebraic solvers in OpenFOAM®, an example investigating the details of implementing the preconditioned CG method is now provided. The files are located in the directory "**$FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/PCG**".

The class derived from the lduMatrix::solver class, as shown in Listing 10.3, defines the main member function "solve" using the script in Listing 10.4.

```
    // Member Functions

        //- Solve the matrix with this solver
        virtual solverPerformance solve
        (
            scalarField& psi,
            const scalarField& source,
            const direction cmpt=0
        ) const;
```

Listing 10.4 Script used to define the member function "solve"

The function "solve" implements the solution algorithm of the chosen linear algebraic solver in file "PCG.C". Recalling the preconditioned conjugate gradient algorithm, its sequence of events are given by

1. Calculate $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$
2. Calculate $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$
3. Calculate $\alpha^{(n)} = \dfrac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}$
4. Calculate $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$
5. Calculate $\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$
6. If solution has converged stop
7. Calculate $\beta^{(n+1)} = \dfrac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}$
8. Calculate $\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$
9. Go to step 3

The algorithm is directly implemented in "PCG.C" following the same procedure as described next.

In step 1, depicted in Listing 10.5, the residual is evaluated and stored in the rA variable while the wA variable stores the matrix-solution product $\mathbf{A}\boldsymbol{\phi}^{(0)}$.

```
// --- Calculate A.psi
matrix_.Amul(wA, psi, interfaceBouCoeffs_, interfaces_, cmpt);
// --- Calculate initial residual field
scalarField rA(source - wA);
```

Listing 10.5 Script used to calculate the residuals

Step 2 involves preconditioning. Thus, first the type of preconditioner used is defined with the object preconPtr as (Listing 10.6).

```
// --- Select and construct the preconditioner
autoPtr<lduMatrix::preconditioner> preconPtr =
lduMatrix::preconditioner::New
(
    *this,
    controlDict_
);
```

Listing 10.6 Defining the type of preconditioner used

The constructor used is a generic one based on the base class and the "New" constructor. The preconditioner type is chosen from the dictionary at run time. Then the preconditioning operation, $\mathbf{P}^{-1}\mathbf{r}^{(n)}$, is applied to the residual rA (according to the equation in step 2 of the algorithm). The result is stored in the same variable wA

to reduce memory usage and then used in the evaluation of $\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}$ by simply performing the scalar product of the two vectors wA and rA using the gSumProd function, as shown in Listing 10.7. Moreover the old value of wArA from the previous iteration $(n-1)$ is stored in the variable wArAold.

```
            wArAold = wArA;

            // --- Precondition residual
            preconPtr->precondition(wA, rA, cmpt);

            // --- Update search directions:
            wArA = gSumProd(wA, rA, matrix().mesh().comm());
```

Listing 10.7 Syntax used to calculate $\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}$ and to store its old value

Following the preconditioned conjugate gradient algorithm, steps 3, 4, and 5 are performed as displayed in Listing 10.8.

```
            // --- Update preconditioned residual
                matrix_.Amul(wA, pA, interfaceBouCoeffs_, interfaces_,
cmpt);

            scalar wApA = gSumProd(wA, pA, matrix().mesh().comm());


            // --- Update solution and residual:

            scalar alpha = wArA/wApA;

            for (register label cell=0; cell<nCells; cell++)
            {
                psiPtr[cell] += alpha*pAPtr[cell];
                rAPtr[cell] -= alpha*wAPtr[cell];
            }
```

Listing 10.8 Script used to calculate α and to update the values of the dependent variable and the residuals

Now the variable wA stores the product $\mathbf{A}\mathbf{d}^{(n)}$ while pA represents the $\mathbf{d}^{(n)}$ vector. Again the product $\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}$ is performed with the gSumProd function and stored in the wApA variable. Once alpha is evaluated, the update of residuals and solution of steps 4 and 5 is performed in the for loop with the variables psiPtr and rAPtr shown in Listing 10.8 representing $\boldsymbol{\phi}$ and $\mathbf{r}$, respectively.

The iteration in the algorithm is completed by executing steps 7 and 8 using the script in Listing 10.9.

```
scalar beta = wArA/wArAold;

for (register label cell=0; cell<nCells; cell++)
{
    pAPtr[cell] = wAPtr[cell] + beta*pAPtr[cell];
}
```

Listing 10.9 Script used for calculating new values for β and **d** to be used in the next iteration

For practical use in test cases, the definitions of the linear solver are made inside the *system* directory in the *fvSolution* under the syntax *solvers* {} as shown in Listing 10.10.

```
solvers
{
    T
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
}
```

Listing 10.10 Definition of the linear solver

The meaning of the various entries in Listing 10.10 are

- *solver*: defines the type of solver (in this case PCG is the preconditioned conjugate gradient for symmetric matrices), with the various options being as follows:
  - "PCG": preconditioned conjugate gradient (for symmetric matrices only).
  - "PBiCG": preconditioned biconjugate gradient (for asymmetric matrices only).
  - "smoothSolver": solver used only as a smoother to reduce residuals.
  - "GAMG": generalised geometric algebraic multigrid. It should be used for the pressure equation on large grids.

- preconditioner: defines the type of the preconditioner to be used. The two available options are
  - "DIC": diagonal incomplete-cholesky preconditioner for symmetric matrices.
  - "DILU": diagonal incomplete-lower-upper preconditioner for asymmetric matrices.
- tolerance: the maximum allowable value of the absolute residual for the linear solver to stop iterating.
- relTol: the ratio between the initial residual and the actual residual for the linear solver to stop iterating.

Other examples are shown in Listing 10.11.

```
solvers
{
    T
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }
}
solvers
{
    T
    {
        solver           smoothSolver;
        smoother         GaussSeidel;
        tolerance        1e-8;
        relTol           0.1;
        nSweeps          1;
    }
    T
    {
        solver           GAMG;
        tolerance        1e-7;
        relTol           0.01;
        smoother         GaussSeidel;
        nPreSweeps       0;
        nPostSweeps      2;
        cacheAgglomeration on;
        agglomerator     faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels      1;
    }
}
```

Listing 10.11 Examples of defining the linear solver

10.6 Closure

The chapter introduced the direct and iterative approaches for solving algebraic systems of equations. In each category a number of methods were described. The algebraic multigrid technique was also discussed. The next chapter will proceed with the discretization of the conservation equation and will detail the discretization of the convection term.

10.7 Exercises

Exercise 1
In each of the following cases obtain the LU factorization of matrix **A** and use it to solve the system of equation **Ax** = **b** by performing the backward and forward substitution, i.e., **Ly** = **b**, **Ux** = **y**:

$$(a) \qquad \mathbf{A} = \begin{pmatrix} 13.0 & 5.0 & 6.0 & 7.0 & 5.0 \\ 2.0 & 12.0 & 6.0 & 3.0 & 4.0 \\ 4.0 & 2.0 & 15.0 & 4.0 & 5.0 \\ 3.0 & 3.0 & 5.0 & 9.0 & 5.0 \\ 4.0 & 6.0 & 0 & 5.0 & 13.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 3.0 \\ 12.0 \\ 13.0 \\ 21.0 \\ 13.0 \end{pmatrix}$$

$$(b) \qquad \mathbf{A} = \begin{pmatrix} 14.0 & 1.0 & 6.0 & 7.0 & 3.0 \\ 7.0 & 10.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 1.0 & 10.0 & 7.0 & 6.0 \\ 4.0 & 7.0 & 6.0 & 11.0 & 1.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 14.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 13.0 \\ 13.0 \\ 16.0 \\ 13.0 \\ 1.0 \end{pmatrix}$$

$$(c) \qquad \mathbf{A} = \begin{pmatrix} 12.0 & 6.0 & 1.0 & 5.0 & 4.0 \\ 4.0 & 11.0 & 1.0 & 0 & 0 \\ 3.0 & 6.0 & 14.0 & 1.0 & 6.0 \\ 6.0 & 2.0 & 7.0 & 10.0 & 7.0 \\ 4.0 & 1.0 & 1.0 & 6.0 & 12.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 13.0 \\ 19.0 \\ 7.0 \\ 22.0 \\ 30.0 \end{pmatrix}$$

Exercise 2
Using the Gauss-Seidel and Jacobi methods and starting with a zero initial guess solve the following systems of equations while adopting as a stopping criteria $\max(\|\mathbf{Ax} - \mathbf{b}\|) < 0.1$:

(*a*)
$$\mathbf{A} = \begin{pmatrix} 6 & 1 & 0 & 3 & 3 \\ 0 & 6 & 0 & 4 & 3 \\ 0 & 5 & 6 & 1 & 0 \\ 5 & 5 & 0 & 6 & 0 \\ 2 & 0 & 2 & 4 & 10 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 22 \\ 21 \\ 20 \\ 10 \\ 22 \end{pmatrix}$$

(*b*)
$$\mathbf{A} = \begin{pmatrix} 36 & 4 & 5 & 4 & 0 & 2 \\ 0 & 40 & 4 & 0 & 0 & 3 \\ 0 & 0 & 37 & 0 & 2 & 4 \\ 3 & 2 & 0 & 36 & 1 & 5 \\ 3 & 3 & 1 & 0 & 36 & 0 \\ 5 & 0 & 2 & 0 & 2 & 40 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 24 \\ 14 \\ 19 \\ 29 \\ 20 \\ 8 \end{pmatrix}$$

(*c*)
$$\mathbf{A} = \begin{pmatrix} 27 & 0 & 0 & 1 \\ 0 & 27 & 1 & 0 \\ 0 & 0 & 26 & 1 \\ 1 & 2 & 5 & 26 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 21 \\ 3 \\ 10 \\ 29 \end{pmatrix}$$

Exercise 3

For the systems of equations in Exercise 2 find the preconditioned matrix **P** for both the Gauss-Seidel and Jacobi methods. Use Eq. (10.81) with a zero initial guess to resolve the systems of equations subject to the same stopping criteria. Compare solutions with those obtained in Exercise 2.

Exercise 4

Perform the ILU(0) factorization of the following M matrices:

(*a*)
$$\mathbf{M} = \begin{pmatrix} 6 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 3 & 1 & 3 \\ 0 & 0 & 6 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 1 & 1 & 6 & 0 \\ 0 & 0 & 5 & 0 & 3 & 0 & 6 \end{pmatrix}$$

(*b*)
$$\mathbf{M} = \begin{pmatrix} 6 & 0 & 0 & 0 & 0 & 3 & 4 & 0 \\ 3 & 7 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 6 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 \\ 4 & 1 & 0 & 0 & 1 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 6 \end{pmatrix}$$

(*c*)
$$\mathbf{M} = \begin{pmatrix} 6 & 0 & 0 & 2 & 0 \\ 0 & 10 & 0 & 1 & 4 \\ 0 & 1 & 6 & 0 & 4 \\ 3 & 0 & 5 & 6 & 5 \\ 0 & 4 & 0 & 5 & 10 \end{pmatrix}$$

Exercise 5
Based on the factorizations in Exercise 4 and knowing that the lower and upper parts of the factorization correspond to $\overline{\mathbf{L}}$ and $\overline{\mathbf{U}}$, respectively, find for each of the cases the error matrix defined as $\mathbf{R} = \mathbf{M} - \mathbf{P}$.

Exercise 6
Perform the DILU factorization of the following M matrices:

(*a*)
$$\mathbf{M} = \begin{pmatrix} 9 & 4 & 0 & 0 & 2 & 5 & 0 \\ 3 & 7 & 0 & 0 & 1 & 2 & 1 \\ 0 & 5 & 10 & 0 & 5 & 5 & 4 \\ 0 & 1 & 1 & 6 & 3 & 0 & 0 \\ 4 & 0 & 0 & 1 & 6 & 1 & 1 \\ 0 & 5 & 2 & 0 & 0 & 6 & 3 \\ 4 & 0 & 0 & 0 & 0 & 2 & 8 \end{pmatrix}$$

(*b*)
$$\mathbf{M} = \begin{pmatrix} 6 & 2 & 2 & 3 & 4 & 3 & 4 & 0 \\ 1 & 6 & 0 & 0 & 1 & 4 & 3 & 0 \\ 1 & 0 & 10 & 0 & 0 & 3 & 4 & 0 \\ 2 & 1 & 4 & 8 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 4 & 8 & 2 & 0 & 0 \\ 1 & 0 & 3 & 0 & 1 & 8 & 0 & 1 \\ 2 & 2 & 1 & 1 & 1 & 0 & 6 & 0 \\ 4 & 0 & 4 & 0 & 0 & 4 & 1 & 6 \end{pmatrix}$$

(*c*)
$$\mathbf{M} = \begin{pmatrix} 6 & 2 & 3 & 0 & 5 & 0 \\ 1 & 6 & 0 & 3 & 0 & 1 \\ 3 & 3 & 8 & 3 & 4 & 0 \\ 4 & 4 & 1 & 7 & 1 & 0 \\ 3 & 4 & 0 & 0 & 8 & 0 \\ 3 & 0 & 4 & 0 & 0 & 6 \end{pmatrix}$$

Exercise 7
Based on the diagonal factorizations in exercise 6 and using Eq. (10.91), find for each of the cases the error matrix defined as $\mathbf{R} = \mathbf{M} - \mathbf{P}$.

Exercise 8
Solve the following systems of equations using the ILU(0) and DILU methods. Perform three iterations only starting with a zero initial guess.

(a)

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 0 & 0 & 0 & 0 \\ 5 & 5 & 0 & 0 & 0 & 2 \\ 3 & 2 & 7 & 0 & 1 & 2 \\ 4 & 1 & 2 & 7 & 0 & 1 \\ 4 & 2 & 0 & 0 & 4 & 0 \\ 3 & 3 & 2 & 0 & 2 & 10 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 26 \\ 28 \\ 18 \\ 26 \\ 11 \\ 26 \end{pmatrix}$$

(b)

$$\mathbf{A} = \begin{pmatrix} 12 & 3 & 1 & 4 & 4 & 1 \\ 4 & 7 & 1 & 0 & 3 & 0 \\ 4 & 5 & 7 & 0 & 0 & 0 \\ 3 & 2 & 0 & 6 & 2 & 0 \\ 3 & 0 & 2 & 1 & 7 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 25 \\ 24 \\ 4 \\ 4 \\ 2 \\ 25 \end{pmatrix}$$

Exercise 9

Solve the systems of equations in Exercise 8 using the Gauss-Seidel and Jacobi methods (starting with a zero initial guess) and compare the errors after three iterations with the errors obtained in exercise 8 and with the exact solution. Comment on the results.

Exercise 10

Solve the following symmetric systems of equations by performing two iterations of the preconditioned conjugate gradient method (start with a zero field):

(a)

$$\mathbf{A} = \begin{pmatrix} 6 & 0 & -1 & 0 & -1 \\ 0 & 6 & 1 & 1 & -1 \\ -1 & 1 & 6 & 0 & 0 \\ 0 & 1 & 0 & 4 & -1 \\ -1 & -1 & 0 & -1 & 6 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 7 \\ 9 \\ 27 \\ 23 \\ 1 \end{pmatrix}$$

(b)

$$\mathbf{A} = \begin{pmatrix} 4 & 1 & 0 & -1 & 0 \\ 1 & 5 & 2 & -1 & 3 \\ 0 & 2 & 5 & 0 & 0 \\ -1 & -1 & 0 & 6 & 0 \\ 0 & 3 & 0 & 0 & 5 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 8 \\ 2 \\ 12 \\ 11 \\ 30 \end{pmatrix}$$

(c)

$$\mathbf{A} = \begin{pmatrix} 3 & 1 & -1 & 2 & -1 & 0 \\ 1 & 7 & 1 & 1 & 2 & -2 \\ -1 & 1 & 7 & 0 & 0 & 0 \\ 2 & 1 & 0 & 6 & 0 & 2 \\ -1 & 2 & 0 & 0 & 6 & -1 \\ 0 & -2 & 0 & 2 & -1 & 7 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 29 \\ 9 \\ 3 \\ 13 \\ 28 \\ 27 \end{pmatrix}$$

Exercise 11
List all the available linear solvers inside OpenFOAM®.
After choosing the smoothSolver in OpenFOAM®, list all implemented smoothers.

Exercise 12
Find in OpenFOAM® the implementation of the BiCG linear solver (PBiCG) and compare it with the BiCG algorithm of Lanczos.

Exercise 13
Find in OpenFOAM® the implementation of the multigrid V-cycle ($FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/GAMG/GAMGSolverSolve.C) and compare it with the theoretical V-cycle algorithm.

Exercise 14
Verify the correct implementation in OpenFOAM® of the diagonal version of the ILU(0) developed by Pommerell.

References

1. Duff I, Erisman A, Reid J (1986) Direct methods for sparse matrices. Clarendon Press, Oxford
2. Westlake JR (1968) A handbook of numerical matrix inversion and solution of linear equations. Wiley, New York
3. Stoer J, Bulirsch R (1980) Introduction to numerical analysis. Springer, New York
4. Press WH (2007) Numerical recipes, 3rd edn. The art of scientific computing. Cambridge University Press, Cambridge
5. Thomas LH (1949) Elliptic problems in linear differential equations over a network. Watson Sci. Comput. Lab Report, Columbia University, New York
6. Conte SD, deBoor C (1972) Elementary numerical analysis. McGraw-Hill, New York
7. Pozrikidis C (1998) Numerical computation in science and engineering. Oxford University Press, Oxford
8. Sebben S, Baliga BR (1995) Some extensions of tridiagonal and pentadiagonal matrix algorithms. Numer Heat Transfer, Part B, 28:323–351
9. Zhao X-L, Huang T-Z (2008) On the inverse of a general pentadiagonal matrix. Appl Math Comput 202(2):639–646
10. Karawia AA (2010) Two algorithms for solving general backward pentadiagonal linear systems. Int J Comput Math 87(12):2823–2830
11. Hageman L, Young D (1981) Applied iterative methods. Academic Press, New York
12. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics
13. Golub G, Van Loan C (2012) Matrix computations, 4th edn. The Johns Hopkins University Press, Baltimore
14. Dongarra J, Van Der Vorst H (1993) Performance of various computers using standard sparse linear equations solving techniques. In: Computer benchmarks. Elsevier Science Publishers BV, New York, pp 177–188
15. Van Der Vorst H (1981) Iterative solution methods for certain sparse linear systems with a nonsymmetric matrix arising from PDEProblems. J Comput Phys 44:1–19
16. Meijerink J, Van Der Vorst H (1977) An iterative solution method for linear systems of which the coefficient matrix is a symmetric M matrix. Math Comput 31:148–162

17. Beauwens R, Quenon L (1976) Existence criteria for partial matrix factorizations in iterative methods. SIAM J Numer Anal 13:615–643
18. Pommerell C (1992) Solution of large unsymmetric systems of linear equations. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland
19. Van Der Sluis A, Van Der Vorst H (1986) The rate of convergence of conjugate gradients. Numer Math 48(5):543–560
20. Faber V, Manteuffel T (1984) Necessary and sufficient conditions for the existence of a conjugate gradient method. SIAM J Numer Anal 21:315–339
21. Lanczos C (1950) An iteration method for the solution of eigenvalue problem of linear differential and integral operators. J Res Natl Bur Stand 45:255–282 (RP 2133)
22. Lanczos C (1952) Solution of systems of linear equations by minimized iterations. J Res Natl Bur Stand 49(1):33–53 (RP 2341)
23. Fletcher R (1976) Conjugate gradient methods for indefinite systems. In: Watson G (ed) Numerical analysis Dundee 1975. Springer, Berlin, pp 73–89
24. Sonneveld P (1989) CGS, AFast Lanczostype solver for nonsymmetric linear systems. SIAM J Sci Stat Comput 10:36–52
25. Van Der Vorst H (1992) BiCGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. SIAM J Sci Stat Comput 13:631–644
26. Van Der Vorst H, Vuik C (1991) GMRESR: a family of nested GMRES methods. Technical report 9180, Delft University of Technology, Faculty of Tech. Math, Delft, The Netherlands
27. Southwell R (1946) Relaxation methods in theoretical physics. Clarendon Press, Oxford
28. Demmel J, Heath M, Van Der Vorst H (1993) Parallel numerical linear algebra. Acta Numerica 2:111–198
29. Saad Y, Schultz M (1986) A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J Sci Stat Comput 7:856–869
30. Fedorenko P (1962) A relaxation method for solving elliptic difference equations. USSR Comput Math Math Phys 1(4):1092–1096
31. Poussin FV (1968) An accelerated relaxation algorithm for iterative solution of elliptic equations. SIAM J Numer Anal 5:340–351
32. Settari A, Aziz K (1973) A generalization of the additive correction methods for the iterative solution of matrix equations. SIAM J Numer Anal 10(3):506–521
33. Brandt A (1977) Multi-level adaptive solutions to boundary value problems. Math Comput 31 (138):333–390
34. Briggs WL (1987) A multigrid tutorial. Society of Industrial and Applied Mathematics, Philadelphia, PA
35. Shome B (2006) An enhanced additive correction multigrid method. Numer Heat Transfer B 49:395–407
36. Hutchinson BR, Raithby GD (1986) A multigrid method based on the additive correction strategy. Numer Heat Transfer 9:511–537
37. Elias SR, Stubley GD, Raithby GD (1997) An adaptive agglomeration method for additive correction multigrid. Int J Numer Meth Eng 40:887–903
38. Mavriplis D, Venkatakfrishnan V (1994) Agglomeration multigrid for viscous turbulent flows. AIAA paper 94-2332
39. Phillips RE, Schmidt FW (1985) A multilevel-multigrid technique for recirculating flows. Numer Heat Transfer 8:573–594
40. Lonsdale RD (1991) An algebraic multigrid scheme for solving the Navier-Stokes equations on unstructured meshes. In: Taylor C, Chin JH, Homsy GM (eds) Numerical methods in laminar and turbulent flow, vol 7(2). Pineridge Press, Swansea, pp 1432–1442
41. Perez E (1985) A 3D finite element multigrid solver for the euler equations. INRIA report 442
42. Connell SD, Braaten DG (1994) A 3D unstructured adaptive multigrid scheme for the Euler equations. AIAA J 32:1626–1632
43. Parthasarathy V, Kallinderis Y (1994) New multigrid approach for three-dimensional unstructured adaptive grids. AIAA J 32:956–963

44. Mavriplis DJ (1995) Three-dimensional multigrid reynolds-averaged Navier-Stokes solver for unstructured meshes. AIAA J 33(12):445–453
45. Qinghua W, Yogendra J (2006) Algebraic multigrid preconditioned Krylov subspace methods for fluid flow and heat transfer on unstructured meshes. Numer Heat Transfer B 49:197–221
46. Lallemand M, Steve H, Dervieux A (1992) Unstructured multigridding by volume agglomeration: current status. Comput Fluids 21:397–433
47. Mavriplis DJ (1999) Directional agglomeration multigrid techniques for high Reynolds Number viscous flow solvers. AIAA J 37:393–415
48. Trottenberg U, Oosterlee C, Schüller A (2001) Multigrid. Academic Press, London
49. Phillips RE, Schmidt FW (1985) Multigrid techniques for the solution of the passive scalar advection-diffusion equation. Numer Heat Transfer 8:25–43
50. Ruge JW, Stüben K (1987) Algebraic multigrid (AMG). In: McCormick SF (ed) Multigrid methods, frontiers in applied mathematics, vol 3. SIAM, Philadelphia, pp 73–130
51. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org

Chapter 11
Discretization of the Convection Term

Abstract So far, the discretization of the general steady diffusion equation has been formulated on orthogonal, non-orthogonal, structured, and unstructured grids. Another important term, the convection term represented by the divergence operator, is the focus of this chapter. Initially this term is discretized using a symmetrical linear profile similar to the one adopted for the discretization of the diffusion term. The shortcomings of this profile are delineated and a remedy is suggested through the use of an upwind profile. Even though it leads to physically plausible predictions, the upwind profile is shown to be highly diffusive generating results that are first order accurate. To increase accuracy, higher order profiles that are upwind biased are introduced. While reducing the discretization error, higher order profiles are shown to give rise to another type of error known as the dispersion error. Methods dealing with this error will be dealt with in the next chapter. Moreover, the flow field, which represents the driving catalyst of the convection term, is assumed to be known. The computation of the flow field will be the subject of later chapters.

11.1 Introduction

Although the convection term looks simple, its discretization presents a number of difficulties that have occupied researchers for more than three decades now. Their work has shed light on the problems that hindered its discretization and resulted in the development of a large number of convection schemes. The body of literature is so large that two chapters are devoted for the analysis of this term. In this chapter the basic concepts are introduced and High Order (HO) [1–3] upwind biased schemes are discussed. The bounding of the convective flux, which made possible the development of non-oscillatory convection schemes of high order of accuracy, denoted by High Resolution (HR) schemes, will be discussed in the next chapter.

For clarity, the presentation of new concepts will be initially introduced using a one dimensional grid, and then extended to multi dimensional non-orthogonal grids. The chapter starts with the discretization of the one-dimensional convection-diffusion

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_11

problem to highlight, through a stability criterion, the shortcomings of the central difference scheme [4, 5]. The upwind scheme [6] is shown to pass the stability test. The numerical diffusion error associated with low order schemes and the numerical dispersion error accompanying high order schemes are also explained. The treatment of the class of HR schemes will be discussed in the next chapter.

11.2 Steady One Dimensional Convection and Diffusion

Initially the derivations are performed for a very simple, one dimensional, steady convection-diffusion problem to learn as much as possible without being distracted by the complexity of the calculations. The governing equation for the case studied can be written as

$$\frac{d(\rho u\phi)}{dx} - \frac{d}{dx}\left(\Gamma^{\phi}\frac{d\phi}{dx}\right) = 0 \tag{11.1}$$

Luckily an analytical solution for the problem is available, and is used as a reference with which various numerical solutions are compared.

11.2.1 Analytical Solution

The continuity equation for this steady-state one dimensional problem of constant cross sectional area is given by

$$\frac{d(\rho u)}{dx} = 0 \tag{11.2}$$

implying that ρu is constant. Having this in mind and integrating with respect to x, Eq. (11.1) becomes

$$\rho u\phi - \Gamma^{\phi}\frac{d\phi}{dx} = c_1 \tag{11.3}$$

where c_1 is a constant of integration the value of which depends on the boundary conditions used. Rearranging, Eq. (11.3) is rewritten as

$$\frac{d\phi}{dx} = \frac{\rho u}{\Gamma^{\phi}}\phi - \frac{c_1}{\Gamma^{\phi}} \tag{11.4}$$

Through a change of variable, Eq. (11.4) is transformed to

$$\frac{d\Phi}{dx} = \frac{\rho u}{\Gamma^{\phi}}\Phi \tag{11.5}$$

where

$$\Phi = \frac{\rho u}{\Gamma^{\phi}}\phi - \frac{c_1}{\Gamma^{\phi}} \tag{11.6}$$

Separating variables and integrating, the solution to Eq. (11.5) is found to be

$$\frac{d\Phi}{\Phi} = \frac{\rho u}{\Gamma^{\phi}}dx \Rightarrow Ln(\Phi) = \frac{\rho u}{\Gamma^{\phi}}x + c_3 \Rightarrow \Phi = c_2 e^{\frac{\rho u}{\Gamma^{\phi}}x} \tag{11.7}$$

where c_2 is another constant of integration. Reverting back to the original variable, the general solution for ϕ is given by

$$\phi = \frac{c_2\Gamma^{\phi}e^{\frac{\rho u}{\Gamma^{\phi}}x} + c_1}{\rho u} \tag{11.8}$$

Thus the analytical solution between the two points W and E shown in Fig. 11.1 and subject to

$$\begin{cases} \phi = \phi_W \; at \; x = x_W \\ \phi = \phi_E \; at \; x = x_E \end{cases} \tag{11.9}$$

is obtained as

$$\frac{\phi - \phi_W}{\phi_E - \phi_W} = \frac{e^{Pe_L\frac{x-x_W}{L}} - 1}{e^{Pe_L} - 1} \tag{11.10}$$

where Pe_L is the Péclet number (based on the length L), which represents the ratio of the advective transport rate of ϕ to its diffusive transport rate, and is given by

$$Pe_L = \frac{\rho u L}{\Gamma^{\phi}} \quad \text{and} \quad L = x_E - x_W. \tag{11.11}$$

Equation (11.10) is evaluated for different values of Pe_L and results are displayed in Fig. 11.2. As shown, variations in ϕ between W and E change from a linear profile for pure diffusion problems to almost a step profile at high values of Pe_L.

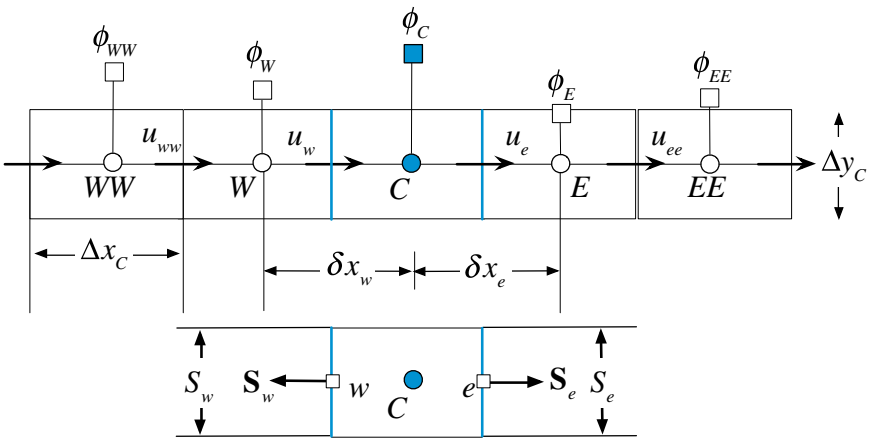


Fig. 11.1 Notation for a one dimensional grid system

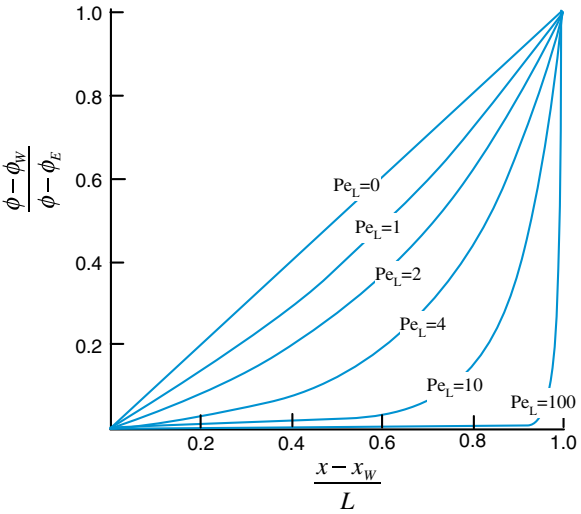


Fig. 11.2 Analytical solutions of the one dimensional convection and diffusion problem for various Pe_L

11.2.2 Numerical Solution

The discretization of Eq. (11.1) starts by integrating the conservation equation over the one dimensional element shown in Fig. 11.1 to yield

$$\int_{V_c} \left[\nabla \cdot (\rho \mathbf{v}\phi) - \nabla \cdot \left(\Gamma^{\phi} \nabla \phi\right)\right] dV = 0 \tag{11.12}$$

where $\mathbf{v} = u\mathbf{i}$ is the velocity vector. The conservation equation can be written in terms of the convection and diffusion fluxes as

$$\int_{V_c} \nabla \cdot \left(\mathbf{J}^{\phi,C} + \mathbf{J}^{\phi,D}\right) dV = 0 \quad \text{where} \quad \mathbf{J}^{\phi,C} = \rho \mathbf{v} \phi \quad \text{and} \quad \mathbf{J}^{\phi,D} = -\Gamma^{\phi} \nabla \phi. \tag{11.13}$$

Then, using the divergence theorem, the volume integral is transformed into a surface integral giving

$$\int_{V_c} \nabla \cdot \left(\mathbf{J}^{\phi,C} + \mathbf{J}^{\phi,D}\right) dV = \int_{\partial V_c} \left(\mathbf{J}^{\phi,C} + \mathbf{J}^{\phi,D}\right) \cdot d\mathbf{S} = \int_{\partial V_c} \left[\rho u \phi \mathbf{i} - \Gamma^{\phi} \frac{d\phi}{dx} \mathbf{i}\right] \cdot d\mathbf{S} = 0. \tag{11.14}$$

Replacing the surface integral by a summation of fluxes over the element faces, Eq. (11.14) becomes

$$\sum_{f \sim nb(C)} \left(\rho u \phi \mathbf{i} - \Gamma^{\phi} \frac{d\phi}{dx} \mathbf{i}\right)_f \cdot \mathbf{S}_f = 0. \tag{11.15}$$

Noting that the surface vectors on opposite sides of the element have opposite signs, the expanded form of Eq. (11.15) for a constant cross section is obtained as

$$\left[(\rho u \Delta y \phi)_e - \left(\Gamma^{\phi} \frac{d\phi}{dx} \Delta y\right)_e\right] - \left[(\rho u \Delta y \phi)_w - \left(\Gamma^{\phi} \frac{d\phi}{dx} \Delta y\right)_w\right] = 0 \tag{11.16}$$

The values of u at the cell faces are known, and those of the gradient can be discretized in the way previously described. The question is how to proceed in the discretization of the face values ϕ_e and ϕ_w in terms of the values at adjacent nodes. The method of specifying these face values is denoted in the literature by the "advection scheme".

11.2.3 A Preliminary Derivation: The Central Difference (CD) Scheme

At first sight the "obvious" answer would be a linear interpolation profile similar to the one used for the diffusion term. Hence, the value of ϕ at a given face, say the right face e, will be computed as

$$\phi(x) = k_0 + k_1 (x - x_C) \tag{11.17}$$

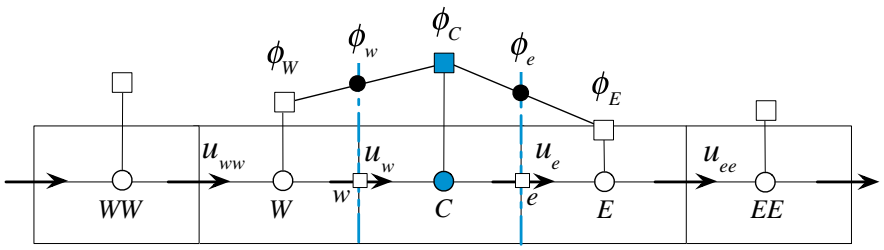


Fig. 11.3 The Central Difference scheme profile

where k_0 and k_1 are constants evaluated using the two nodes straddling face e. Thus using the fact that $\phi = \phi_E$ at $x = x_E$ and $\phi = \phi_C$ at $x = x_C$, Eq. (11.17) evaluated at $x = x_e$ gives

$$\phi_e = \phi_C + \frac{(\phi_E - \phi_C)}{(x_E - x_C)}(x_e - x_C). \tag{11.18}$$

This is basically the central difference scheme that can be obtained by a Taylor series expansion where terms involving derivatives of the second order and higher are neglected, which means it is second order accurate.

For the uniform grid shown in Fig. 11.3, Eq. (11.18) becomes

$$\phi_e = \frac{\phi_C + \phi_E}{2} \tag{11.19}$$

Example 1

Derive the value of ϕ_e for a central difference scheme using a Taylor expansion approach

Solution

The Taylor expansion about e can *be written as*

$$\phi_C = \underbrace{\phi(-\Delta x_C/2)}_{=\phi_e} + \left.\frac{\partial \phi}{\partial x}\right|_e (-\Delta x_C/2) + O\left(\Delta x_C^2\right)$$

$$= \phi_e - \frac{\phi_E - \phi_C}{\delta x_e}\frac{\Delta x_C}{2}$$

thus

$$\phi_e = \phi_C + \frac{\Delta x_C}{\delta x_e}\frac{\phi_E - \phi_C}{2}$$

for a uniform grid $\delta x_e = \Delta x_C$ *yielding*

$$\phi_e = \frac{\phi_C + \phi_E}{2}$$

Thus after the discretization of the diffusion term using a linear profile (Fig. 11.3), the term in the first square bracket of Eq. (11.16) becomes

$$\begin{aligned}(\rho u \Delta y \phi)_e - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_e &= (\rho u \Delta y)_e \frac{(\phi_E + \phi_C)}{2} - \left(\Gamma^\phi \frac{\Delta y}{\delta x}\right)_e (\phi_E - \phi_C) \\ &= FluxC_e\phi_C + FluxF_e\phi_E + FluxV_e\end{aligned} \tag{11.20}$$

where

$$\begin{aligned}FluxC_e &= \Gamma_e^\phi \frac{\Delta y_e}{\delta x_e} + \frac{(\rho u \Delta y)_e}{2} \\ FluxF_e &= -\Gamma_e^\phi \frac{\Delta y_e}{\delta x_e} + \frac{(\rho u \Delta y)_e}{2} \\ FluxV_e &= 0\end{aligned} \tag{11.21}$$

A similar expression for the term in the second square bracket of Eq. (11.16) can also be derived and is given by

$$\begin{aligned}-\left[(\rho u \Delta y \phi)_w - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_w\right] &= -\left[(\rho u \Delta y)_w \frac{(\phi_W + \phi_C)}{2} - \left(\Gamma^\phi \frac{\Delta y}{\delta x}\right)_w (\phi_C - \phi_W)\right] \\ &= FluxC_w\phi_C + FluxF_w\phi_W + FluxV_w\end{aligned} \tag{11.22}$$

where now

$$\begin{aligned}FluxC_w &= \Gamma_w^\phi \frac{\Delta y_w}{\delta x_w} - \frac{(\rho u \Delta y)_w}{2} \\ FluxF_w &= -\Gamma_w^\phi \frac{\Delta y_w}{\delta x_w} - \frac{(\rho u \Delta y)_w}{2} \\ FluxV_w &= 0\end{aligned} \tag{11.23}$$

Substitution of these values into the convection-diffusion equation yields

$$a_C\phi_C + a_E\phi_E + a_W\phi_W = 0 \tag{11.24}$$

where

$$\begin{aligned} a_E &= FluxF_e = -\Gamma_e^{\phi}\frac{\Delta y_e}{\delta x_e} + \frac{(\rho u \Delta y)_e}{2} \\ a_W &= FluxF_w = -\Gamma_w^{\phi}\frac{\Delta y_w}{\delta x_w} - \frac{(\rho u \Delta y)_w}{2} \\ a_C &= FluxC_e + FluxC_w = \left(\frac{(\rho u \Delta y)_e}{2} + \Gamma_e^{\phi}\frac{\Delta y_e}{\delta x_e}\right) + \left(-\frac{(\rho u \Delta y)_w}{2} + \Gamma_w^{\phi}\frac{\Delta y_w}{\delta x_w}\right) \end{aligned} \tag{11.25}$$

As the problem is one dimensional $\Delta y_e = \Delta y_w$ and, without loss of generality, can be set equal to 1. Moreover, from continuity u is constant and thus $(\rho u \Delta y)_e - (\rho u \Delta y)_w = 0$. Assuming a uniform diffusion coefficient, the coefficients of the discretized equation can be simplified to

$$\begin{aligned} a_E &= -\frac{\Gamma^{\phi}}{x_E - x_C} + \frac{(\rho u)_e}{2} \\ a_W &= -\frac{\Gamma^{\phi}}{x_C - x_W} - \frac{(\rho u)_w}{2} \\ a_C &= -(a_E + a_W) \end{aligned} \tag{11.26}$$

Substituting back in Eq. (11.24), the value for ϕ_C is found as

$$\frac{\phi_C - \phi_W}{\phi_E - \phi_W} = \frac{a_E}{a_E + a_W} \tag{11.27}$$

If the grid is assumed to be uniform, then the above equation can be written in terms of Pe_L as

$$\frac{\phi_C - \phi_W}{\phi_E - \phi_W} = \frac{1}{2}\left(1 - \frac{Pe_L}{2}\right) \tag{11.28}$$

where L is $(x_E - x_W)$, which is the size of two elements. The analytical solution for the problem can be obtained from Eq. (11.10) by setting $(x - x_W)/L$ to 0.5 and is given by

$$\frac{\phi_C - \phi_W}{\phi_E - \phi_W} = \frac{e^{\frac{Pe_L}{2}} - 1}{e^{Pe_L} - 1} \tag{11.29}$$

The two solutions are compared in Fig. 11.4 as Pe_L varies from −10 to +10. Figure 11.4 demonstrates that at low values of Pe_L the numerical and analytical

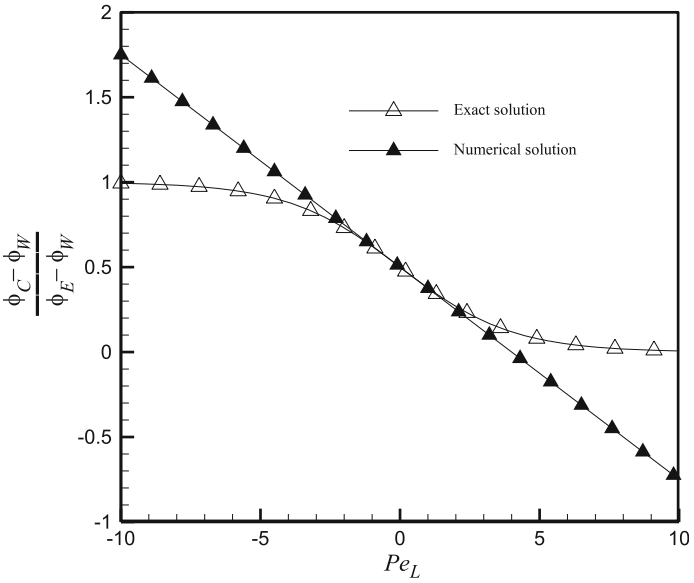


Fig. 11.4 Comparison of numerical (obtained with the CD scheme) and analytical solutions for the one dimensional convection and diffusion problem

solutions are very close to each other. However as Pe_L is increased beyond a certain value the central difference (CD) numerical solution greatly departs from the analytical solution and becomes unbounded experiencing unphysical behavior. Whereas the analytical solution approaches asymptotically the values 0 and 1 for positive and negative values of Pe_L, respectively, the CD solution decreases linearly from $+\infty$ to $-\infty$ as Pe_L increases from $-\infty$ to $+\infty$. This solution indicates that some of the assumptions used in the discretization of the equation are unrealistic or unphysical. What is the reason for this behavior?

As depicted in Fig. 11.5, whereas diffusion at point C is equally affected by conditions upstream and downstream of C (Fig. 11.5a), the advection process is a highly directional process transporting properties only in the direction of the flow (Fig. 11.5b). Therefore the linear profile approximation, which assigns equal weight to both the upwind and downwind nodes, is a good approximation for the diffusion term (Fig. 11.5a). However it cannot describe the directional preference of convection, for which a step profile is more appropriate (Fig. 11.5b), and is the cause of the problem.

The combined convection-diffusion zone of influence and the more relevant profile in this case is schematically depicted in Fig. 11.5c. This zone of influence approaches the diffusion region displayed in Fig. 11.5a and the advection region depicted in Fig. 11.5b at low and high values of the Péclet number, respectively.

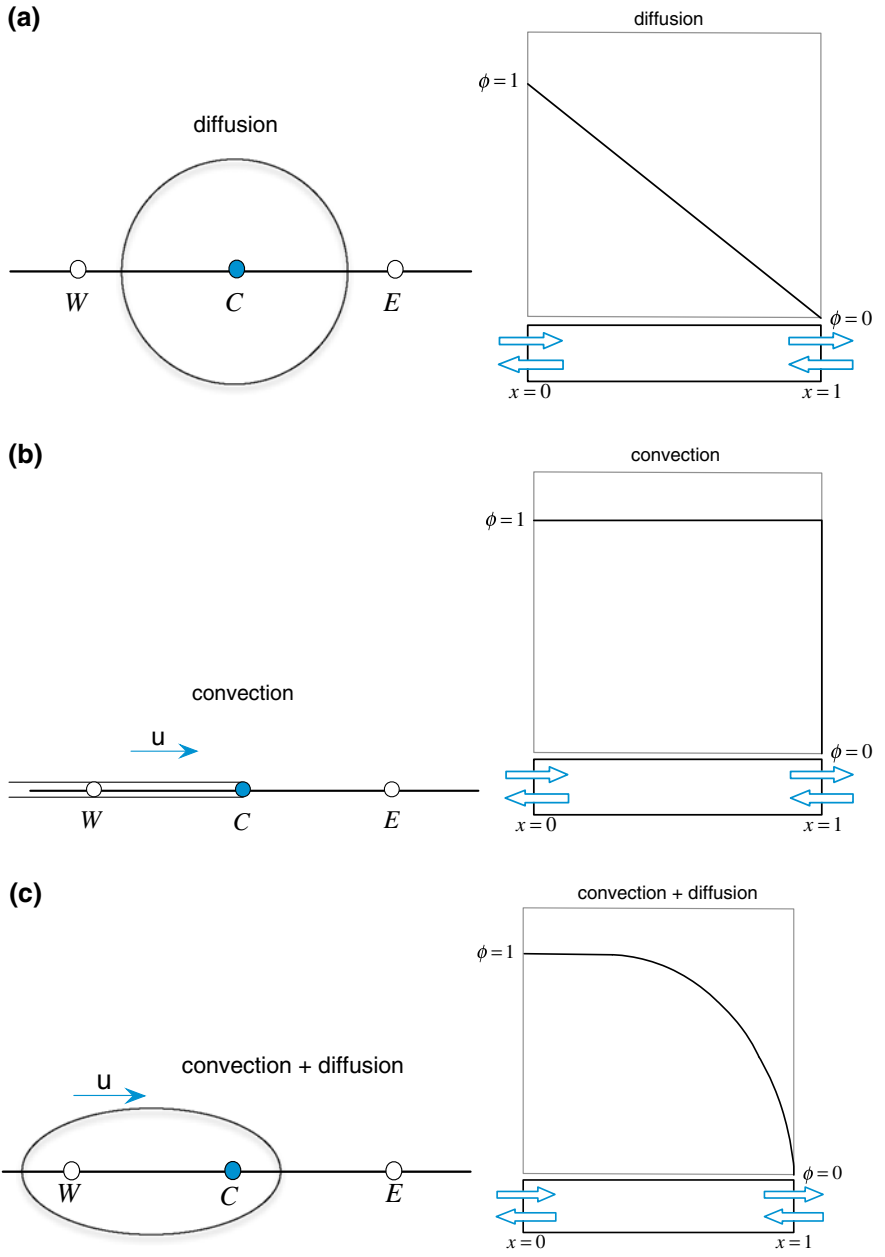


Fig. 11.5 Zone of influence of **a** diffusion, **b** convection, and **c** combined convection and diffusion terms and expected problem solution

Therefore, as long as diffusion is the dominant transfer mechanism, the use of a linear profile yields physical results. However, once convection overwhelms diffusion, unphysical results are obtained. The value of Péclet number at which this occurs can be easily calculated. Assuming the flow to be in the positive x direction, it is noted that there is a possibility for the a_E coefficient to become positive, thus leading to unphysical results (if the flow is in the negative x direction, then a_W may become positive) when,

$$-\Gamma_e^{\phi}\frac{\Delta y_e}{\delta x_e}+\frac{(\rho u\Delta y)_e}{2}>0\Rightarrow\frac{(\rho u)_e\delta x_e}{\Gamma_e^{\phi}}>2. \tag{11.30}$$

Defining the cell Péclet number as

$$Pe=\frac{\rho u\delta x}{\Gamma^{\phi}}. \tag{11.31}$$

which, for a uniform grid, is half Pe_L, then Eq. (11.30) can be rewritten as

$$Pe>2. \tag{11.32}$$

Thus for a cell Péclet number (Pe) larger than 2 the discretization process becomes inconsistent as now an increase in the neighboring value will lead to a decrease in the value at C. This in turn will lead to a further increase in the neighboring value, and the error is magnified.

This situation can be avoided by decreasing the grid size so that the cell Péclet number is smaller than 2. For many practical situations, however, the increase in storage and computational requirements may be too large to afford. Moreover for purely convected flows (e.g., Euler flows) this is simply not feasible. Therefore a remedy is needed.

11.2.4 The Upwind Scheme

When examining the discretization procedure described above, it is noticed that the reason for obtaining these positive coefficients is because of the adopted linear symmetric profile. A linear symmetric profile gives equal weights to the two nodes sharing the face with no directional preference, which is appropriate for non-directional phenomena with an elliptic type term, such as the diffusion term. It is simply not adequate for the convection term [4].

A scheme that is more compatible with the advection process is the upwind scheme [4, 6] schematically displayed in Fig. 11.6. The upwind scheme basically mimics the basic physics of advection in that the cell face value is made dependent on the upwind nodal value, i.e., dependent on the flow direction. In this case the cell face values for the configuration displayed in Fig. 11.6 are given by

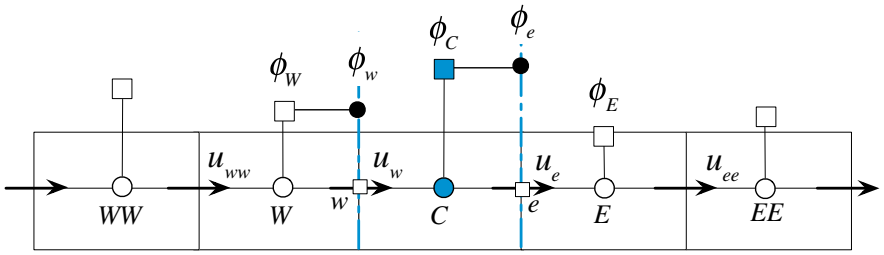


Fig. 11.6 The upwind scheme profile

$$\phi_e = \begin{cases} \phi_C & if \ \dot{m}_e > 0 \\ \phi_E & if \ \dot{m}_e < 0 \end{cases} \quad \text{and} \quad \phi_w = \begin{cases} \phi_C & if \ \dot{m}_w > 0 \\ \phi_W & if \ \dot{m}_w < 0 \end{cases} \tag{11.33}$$

where $\dot{m}_e$ and $\dot{m}_w$ are the mass flow rates at faces e and w given by

$$\begin{aligned} \dot{m}_e &= (\rho \mathbf{v} \cdot \mathbf{S})_e = (\rho u S)_e = (\rho u \Delta y)_e \\ \dot{m}_w &= (\rho \mathbf{v} \cdot \mathbf{S})_w = -(\rho u S)_w = -(\rho u \Delta y)_w \end{aligned} \tag{11.34}$$

Thus, the advection flux at face e can be written as

$$\begin{aligned} \dot{m}_e \phi_e &= \|\dot{m}_e, 0\| \phi_C - \|-\dot{m}_e, 0\| \phi_E \\ &= FluxC_e^{Conv} \phi_C + FluxF_e^{Conv} \phi_E + FluxV_e^{Conv} \end{aligned} \tag{11.35}$$

where

$$\begin{aligned} FluxC_e^{Conv} &= \|\dot{m}_e, 0\| \\ FluxF_e^{Conv} &= -\|-\dot{m}_e, 0\| \\ FluxV_e^{Conv} &= 0 \end{aligned} \tag{11.36}$$

In Eqs. (11.35) and (11.36), the term $\|a, b\|$ represents the maximum of a and b. Moreover, a similar relation can be derived for the advection flux at face w and is given by

$$\begin{aligned} \dot{m}_w \phi_w &= \|\dot{m}_w, 0\| \phi_C - \|-\dot{m}_w, 0\| \phi_W \\ &= FluxC_w^{Conv} \phi_C + FluxF_w^{Conv} \phi_W + FluxV_w^{Conv} \end{aligned} \tag{11.37}$$

where now

$$\begin{aligned} FluxC_w^{Conv} &= \|\dot{m}_w, 0\| \\ FluxF_w^{Conv} &= -\|-\dot{m}_w, 0\| \\ FluxV_w^{Conv} &= 0 \end{aligned} \tag{11.38}$$

Denoting the contribution of the diffusion flux with a superscript *Diff*, then substitution into Eq. (11.15) yields

$$\begin{aligned}\left(FluxC_e^{Conv} + FluxC_e^{Diff} + FluxC_w^{Conv} + FluxC_w^{Diff}\right)\phi_C \\ + \left(FluxF_e^{Conv} + FluxF_e^{Diff}\right)\phi_E + \left(FluxF_w^{Conv} + FluxF_w^{Diff}\right)\phi_W = 0\end{aligned} \tag{11.39}$$

which can be modified into the form

$$a_C\phi_C + a_E\phi_E + a_W\phi_W = 0 \tag{11.40}$$

with

$$\begin{aligned} a_E &= FluxF_e^{Conv} + FluxF_e^{Diff} \\ &= -\|-\dot{m}_e, 0\| - \Gamma_e^{\phi}\frac{S_e}{\delta x_e} \\ a_W &= FluxF_w^{Conv} + FluxF_w^{Diff} \\ &= -\|-\dot{m}_w, 0\| - \Gamma_w^{\phi}\frac{S_w}{\delta x_w} \\ a_C &= \sum_f \left(FluxC_f^{Conv} + FluxC_f^{Diff}\right) \\ &= \|\dot{m}_e, 0\| + \|\dot{m}_w, 0\| + \Gamma_e^{\phi}\frac{S_e}{\delta x_e} + \Gamma_w^{\phi}\frac{S_w}{\delta x_w} \\ &= -(a_E + a_W) + \underbrace{(\dot{m}_e + \dot{m}_w)}_{=0} \\ b_C &= -\sum_f \left(FluxV_f^{Conv} + FluxV_f^{Diff}\right) \\ &= 0 \end{aligned} \tag{11.41}$$

It is easily seen that the upwind scheme yields negative neighbor coefficients, and provided continuity is satisfied, the coefficient at the main node is given by,

$$a_C = -(a_W + a_E) \tag{11.42}$$

which guarantees the boundedness property.

Invoking the continuity constraint in Eq. (11.41) and assuming uniform grid and constant diffusion coefficient, the value for ϕ_C in terms of ϕ_E and ϕ_W is obtained from Eq. (11.40) and Eq. (11.41) as

$$\frac{\phi_C - \phi_W}{\phi_E - \phi_W} = \frac{2 + ||-Pe_L, 0||}{4 + ||-Pe_L, 0|| + ||Pe_L, 0||} = \frac{2 + ||-Pe_L, 0||}{4 + |Pe_L|} \tag{11.43}$$

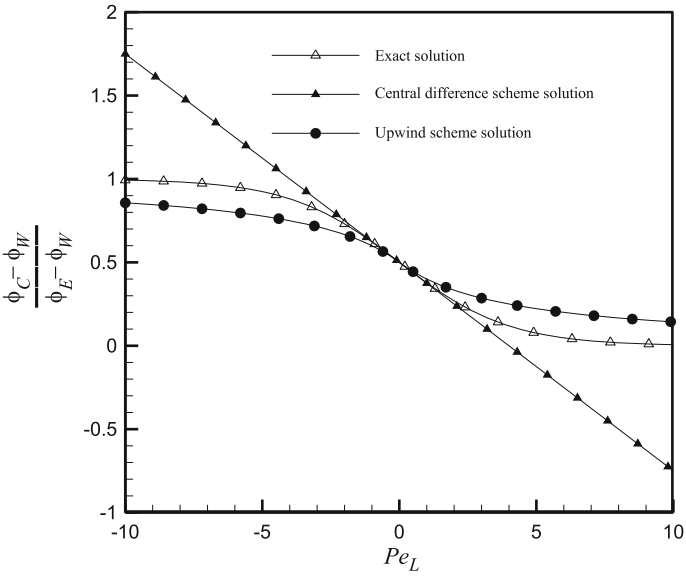


Fig. 11.7 Comparison of solutions obtained analytically and numerically using the upwind and central difference schemes for the one dimensional convection and diffusion problem

The ϕ_C profile generated using the upwind scheme (Eq. 11.43) is compared in Fig. 11.7 with similar ones obtained analytically (Eq. 11.29) and numerically via the central difference scheme (Eq. 11.28). At low Pe_L values, profiles indicate that the upwind scheme is not as accurate as central differencing. This is expected since the upwind profile is first order accurate while the linear profile is second order accurate. At high Pe_L values, the central difference scheme is unstable as its solution is unbounded and physically incorrect. On the other hand, even though the upwind scheme is not particularly accurate, it is physically correct.

Thus there appears to be a tradeoff between accuracy and stability. Using the upwind scheme, solutions are better behaved and bounded even at high Péclet numbers. This is however achieved at the cost of low accuracy. On the other hand, the second order central difference scheme becomes unstable beyond a certain value of Pe_L resulting in physically erroneous solutions. Both schemes seem to be infected by errors, one affecting accuracy while the other affecting stability. What are these errors? This will be discussed after introducing the downwind scheme.

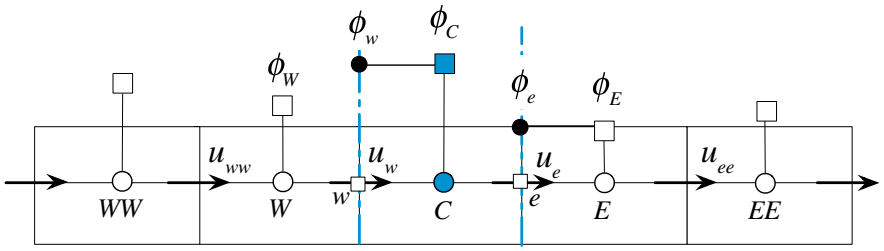


Fig. 11.8 The downwind scheme profile

11.2.5 The Downwind Scheme

It is interesting to see what happens if a scheme opposite to the upwind scheme is used, i.e., the downwind scheme [7, 8]. For this interpolation profile, displayed in Fig. 11.8, the value at the node on the downwind side of the interface is taken to represent the value at the interface. Thus, the values at faces e and w are calculated as

$$\phi_e = \begin{cases} \phi_E & if \quad \dot{m}_e > 0 \\ \phi_C & if \quad \dot{m}_e < 0 \end{cases} \quad \text{and} \quad \phi_w = \begin{cases} \phi_W & if \quad \dot{m}_w > 0 \\ \phi_C & if \quad \dot{m}_w < 0 \end{cases} \tag{11.44}$$

Using Eq. (11.44), the advection fluxes at the faces can be written as

$$\begin{aligned} \dot{m}_e\phi_e &= -\|-\dot{m}_e, 0\|\phi_C + \|\dot{m}_e, 0\|\phi_E \\ &= FluxC_e^{Conv}\phi_C + FluxF_e^{Conv}\phi_E + FluxV_e^{Conv} \\ \dot{m}_w\phi_w &= -\|-\dot{m}_w, 0\|\phi_C + \|\dot{m}_w, 0\|\phi_W \\ &= FluxC_w^{Conv}\phi_C + FluxF_w^{Conv}\phi_W + FluxV_w^{Conv} \end{aligned} \tag{11.45}$$

Substitution of the above values in the discretization equation, Eq. (11.15), yields

$$\begin{aligned} &\left(FluxC_e^{Conv} + FluxC_e^{Diff} + FluxC_w^{Conv} + FluxC_w^{Diff}\right)\phi_C \\ &\quad + \left(FluxF_e^{Conv} + FluxF_e^{Diff}\right)\phi_E + \left(FluxF_w^{Conv} + FluxF_w^{Diff}\right)\phi_W = 0 \end{aligned} \tag{11.46}$$

which can be modified into the form

$$a_C\phi_C + a_E\phi_E + a_W\phi_W = 0 \tag{11.47}$$

with

$$\begin{aligned}
a_E &= FluxF_e^{Conv} + FluxF_e^{Diff} \\
&= \|\dot{m}_e, 0\| - \Gamma_e^{\phi} \frac{S_e}{\delta x_e} \\
a_W &= FluxF_w^{Conv} + FluxF_w^{Diff} \\
&= \|\dot{m}_w, 0\| - \Gamma_w^{\phi} \frac{S_w}{\delta x_w} \\
a_C &= \sum_f \left(FluxC_f^{Conv} + FluxC_f^{Diff} \right) \\
&= - \|-\dot{m}_e, 0\| + \Gamma_e^{\phi} \frac{S_e}{\delta x_e} - \|-\dot{m}_w, 0\| + \Gamma_w^{\phi} \frac{S_w}{\delta x_w} \\
&= - (a_E + a_W) + \underbrace{(\dot{m}_e + \dot{m}_w)}_{=0} \\
b_C &= - \sum_f \left(FluxV_f^{Conv} + FluxV_f^{Diff} \right) \\
&= 0
\end{aligned} \tag{11.48}$$

Invoking the continuity constraint in Eq. (11.48) and assuming uniform grid and constant diffusion coefficient, the value for ϕ_C in terms of ϕ_E and ϕ_W is obtained as

$$\frac{\phi_C - \phi_W}{\phi_E - \phi_W} = \frac{2 - ||Pe_L, 0||}{4 - ||-Pe_L, 0|| - ||Pe_L, 0||} = \frac{2 - ||Pe_L, 0||}{4 - |Pe_L|} \tag{11.49}$$

Without plotting Eq. (11.49) it is clear that as $|Pe_L| \rightarrow 4$ the solution becomes completely unbounded confirming once more the analysis presented above. The downwind scheme may be beneficial when blended with other schemes to predict sharp interfaces [7, 8]. Nevertheless its introduction here will be exploited in the next section to give better insight into stability.

11.3 Truncation Error: Numerical Diffusion and Anti-Diffusion

Truncation error occurs due to the approximate nature of the discretization process and is more easily analyzed for one dimensional situations on Cartesian meshes. The diffusion and anti-diffusion of the upwind, downwind, and central difference schemes are presented next.

11.3.1 The Upwind Scheme

Considering the equation discretized via the upwind scheme, the intention is to recover the integral equation while accounting for the truncation error. To do so, ϕ_C and ϕ_W are written as functions of ϕ_e and ϕ_w, respectively, for the case when the flow is assumed to be in the positive x direction. In this case the upwind scheme results in

$$\phi_e = \phi_C \quad \text{and} \quad \phi_w = \phi_W. \tag{11.50}$$

Thus the one dimensional convection diffusion equation discretized using the upwind scheme simplifies to

$$(\rho u \Delta y)_e \phi_C - (\rho u \Delta y)_w \phi_W - \left[\left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_e - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_w\right] = 0. \tag{11.51}$$

The one dimensional Taylor series expansion of ϕ_C with respect to its value at cell face e is given by

$$\begin{aligned}\phi_C &= \phi_e + \left(\frac{d\phi}{dx}\right)_e (x_C - x_e) + \frac{1}{2}\left(\frac{d^2\phi}{dx^2}\right)_e (x_C - x_e)^2 + \cdots \\ &= \phi_e - \left(\frac{d\phi}{dx}\right)_e (x_e - x_C) + \cdots\end{aligned} \tag{11.52}$$

and for a uniform grid as

$$\phi_C = \phi_e - \left(\frac{d\phi}{dx}\right)_e \frac{(\delta x)_e}{2} + \frac{1}{2}\left(\frac{d^2\phi}{dx^2}\right)_e \left(\frac{(\delta x)_e}{2}\right)^2 \cdots \tag{11.53}$$

A similar expression can be obtained for ϕ_W and is given by

$$\phi_W = \phi_w - \left(\frac{d\phi}{dx}\right)_w \frac{(\delta x)_w}{2} + \frac{1}{2}\left(\frac{d^2\phi}{dx^2}\right)_w \left(\frac{(\delta x)_w}{2}\right)^2 \cdots \tag{11.54}$$

Truncating second order terms and higher and substituting into the advection term, the left hand side of the discretized equation becomes

$$\begin{aligned}&(\rho u \Delta y)_e \phi_C - (\rho u \Delta y)_w \phi_W - \left[\left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_e - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_w\right] \\ &= (\rho u \Delta y)_e \left[\phi_e - \left(\frac{d\phi}{dx}\right)_e \frac{\delta x_e}{2}\right] - (\rho u \Delta y)_w \left[\phi_w - \left(\frac{d\phi}{dx}\right)_w \frac{\delta x_w}{2}\right] \\ &\quad - \left[\left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_e - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_w\right]\end{aligned} \tag{11.55}$$

which can be rearranged into

$$\begin{aligned}&(\rho u\Delta y)_e\phi_C-(\rho u\Delta y)_w\phi_W-\left[\left(\Gamma^\phi\frac{d\phi}{dx}\Delta y\right)_e-\left(\Gamma^\phi\frac{d\phi}{dx}\Delta y\right)_w\right]\\&=(\rho u\Delta y)_e\phi_e-(\rho u\Delta y)_w\phi_w\\&\quad-\left[\left(\Gamma^\phi+\rho u\frac{\delta x}{2}\right)_e\left(\frac{d\phi}{dx}\Delta y\right)_e-\left(\Gamma^\phi+\rho u\frac{\delta x}{2}\right)_w\left(\frac{d\phi}{dx}\Delta y\right)_w\right]\end{aligned}\tag{11.56}$$

It is clear now that the equation being solved has an added component of diffusion, which is called truncation error. The value of the numerical diffusion is equal to

$$\Gamma^\phi_{truncation}=\rho u\frac{\delta x}{2}\tag{11.57}$$

This truncation error, also known as stream wise diffusion, reduces the accuracy of the solution by altering the magnitude of the diffusion coefficient and consequently the equation to be solved. Thus the convection and diffusion equation has an effective modified value of the diffusion effects. On the other hand, this additional stream wise numerical diffusion is desirable as it stabilizes the solution by keeping it bounded and physically correct.

It is obvious that to reduce stream wise numerical diffusion a higher order approximation of the convection term is needed. However, as will be explained in a later section, this should be done in such a way that the solution remains bounded.

11.3.2 The Downwind Scheme

As with the upwind scheme, ϕ_C and ϕ_W are written as functions of ϕ_e and ϕ_w, respectively, for the case when the flow is assumed to be in the positive x direction. In this case the downwind scheme results in

$$\phi_e=\phi_E\quad\text{and}\quad\phi_w=\phi_C\tag{11.58}$$

and the discretized equation becomes

$$(\rho u\Delta y)_e\phi_E-(\rho u\Delta y)_w\phi_C-\left[\left(\Gamma^\phi\frac{d\phi}{dx}\Delta y\right)_e-\left(\Gamma^\phi\frac{d\phi}{dx}\Delta y\right)_w\right]=0\tag{11.59}$$

For a uniform grid, the one dimensional Taylor series expansions of ϕ_E and ϕ_C with respect to their values at cell faces e and w are given by

$$\begin{aligned}\phi_E &= \phi_e + \left(\frac{d\phi}{dx}\right)_e \frac{(\delta x)_e}{2} + \frac{1}{2}\left(\frac{d^2\phi}{dx^2}\right)_e \left(\frac{(\delta x)_e}{2}\right)^2 + \cdots \\ \phi_C &= \phi_w + \left(\frac{d\phi}{dx}\right)_w \frac{(\delta x)_w}{2} + \frac{1}{2}\left(\frac{d^2\phi}{dx^2}\right)_w \left(\frac{(\delta x)_w}{2}\right)^2 + \cdots\end{aligned} \tag{11.60}$$

Truncating second order terms and higher and substituting into the advection term, the left hand side of the discretized equation becomes

$$\begin{aligned}&(\rho u \Delta y)_e \phi_E - (\rho u \Delta y)_w \phi_C - \left[\left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_e - \left(\Gamma^\phi \frac{d\phi}{dx}\Delta y\right)_w\right] \\ &= (\rho u \Delta y)_e \phi_e - (\rho u \Delta y)_w \phi_w \\ &\quad - \left[\left(\Gamma^\phi - \rho u \frac{\delta x}{2}\right)_e \left(\frac{d\phi}{dx}\Delta y\right)_e - \left(\Gamma^\phi - \rho u \frac{\delta x}{2}\right)_w \left(\frac{d\phi}{dx}\Delta y\right)_w\right]\end{aligned} \tag{11.61}$$

For this profile, numerical diffusion has a negative sign and is equal to

$$\Gamma^\phi_{truncation} = -\rho u \frac{\delta x}{2} \tag{11.62}$$

which acts at decreasing the diffusion coefficient and in effect is an anti-diffusion error. Predictions using the downwind scheme are found to cause clipping of the advected profiles. In fact solutions to the one dimensional convection-diffusion problem generated using this scheme are more oscillatory than the CD scheme.

11.3.3 The Central Difference (CD) Scheme

The truncation error for the CD scheme is a little more involved to obtain with the finite volume method as computation of the gradient requires interpolated values at the element faces and not at nodes where they are available. Assuming the velocity is known everywhere and the grid is uniform with a size of Δx, the approximation is simply the one introduced in the calculation of $(\phi_e - \phi_w)$, which can be written as

$$\underbrace{(\phi_e - \phi_w)}_{Interpolated} = \frac{1}{2}(\phi_E + \phi_C) - \frac{1}{2}(\phi_C + \phi_W) = \underbrace{(\phi_e - \phi_w)}_{Exact} + TE \tag{11.63}$$

where *TE* refers to truncation error. To calculate this truncation error, the following Taylor series expansions with respect to the exact ϕ_e and ϕ_w values are needed:

$$\begin{aligned}
\phi_W &= \phi_w - \frac{\Delta x}{2}\phi'_w + \frac{\Delta x^2}{8}\phi''_w - \frac{\Delta x^3}{48}\phi'''_w + \frac{\Delta x^4}{384}\phi^{iv}_w - \frac{\Delta x^5}{3840}\phi^{v}_w + \cdots \\
\phi_C &= \phi_w + \frac{\Delta x}{2}\phi'_w + \frac{\Delta x^2}{8}\phi''_w + \frac{\Delta x^3}{48}\phi'''_w + \frac{\Delta x^4}{384}\phi^{iv}_w + \frac{\Delta x^5}{3840}\phi^{v}_w + \cdots \\
\phi_C &= \phi_e - \frac{\Delta x}{2}\phi'_e + \frac{\Delta x^2}{8}\phi''_e - \frac{\Delta x^3}{48}\phi'''_e + \frac{\Delta x^4}{384}\phi^{iv}_e - \frac{\Delta x^5}{3840}\phi^{v}_e + \cdots \\
\phi_E &= \phi_e + \frac{\Delta x}{2}\phi'_e + \frac{\Delta x^2}{8}\phi''_e + \frac{\Delta x^3}{48}\phi'''_e + \frac{\Delta x^4}{384}\phi^{iv}_e + \frac{\Delta x^5}{3840}\phi^{v}_e + \cdots
\end{aligned} \tag{11.64}$$

Using these expansions, the average values at the faces are obtained as

$$\begin{aligned}
\frac{1}{2}(\phi_E + \phi_C) &= \phi_e + \frac{\Delta x^2}{8}\phi''_e + \frac{\Delta x^4}{384}\phi^{iv}_e + \cdots \\
&\Rightarrow \frac{\Delta x^2}{4}\phi''_e = (\phi_C - 2\phi_e + \phi_E) - \frac{\Delta x^4}{192}\phi^{iv}_e + \cdots \\
\frac{1}{2}(\phi_C + \phi_W) &= \phi_w + \frac{\Delta x^2}{8}\phi''_w + \frac{\Delta x^4}{384}\phi^{iv}_w + \cdots \\
&\Rightarrow \frac{\Delta x^2}{4}\phi''_w = (\phi_W - 2\phi_w + \phi_C) - \frac{\Delta x^4}{192}\phi^{iv}_w + \cdots
\end{aligned} \tag{11.65}$$

Subtracting the above two terms given in Eq. (11.65), their difference is found to be

$$\begin{aligned}
\frac{1}{2}(\phi_E + \phi_C) - \frac{1}{2}(\phi_C + \phi_W) = (\phi_e - \phi_w) &+ \frac{\Delta x^2}{8}\left(\phi''_e - \phi''_w\right) \\
&+ \frac{\Delta x^4}{384}\left(\phi^{iv}_e - \phi^{iv}_w\right) + \cdots
\end{aligned} \tag{11.66}$$

Further, the expanded forms of the exact ϕ_e and ϕ_w with respect to ϕ_C are given by

$$\begin{aligned}
\phi_e &= \phi_C + \frac{\Delta x}{2}\phi'_C + \frac{\Delta x^2}{8}\phi''_C + \frac{\Delta x^3}{48}\phi'''_C + \frac{\Delta x^4}{384}\phi^{iv}_C + \frac{\Delta x^5}{3840}\phi^{v}_C + \cdots \\
\phi_w &= \phi_C - \frac{\Delta x}{2}\phi'_C + \frac{\Delta x^2}{8}\phi''_C - \frac{\Delta x^3}{48}\phi'''_C + \frac{\Delta x^4}{384}\phi^{iv}_C - \frac{\Delta x^5}{3840}\phi^{v}_C + \cdots
\end{aligned} \tag{11.67}$$

In addition, ϕ_E and ϕ_W can be expanded in terms of ϕ_C as

$$\begin{aligned}
\phi_E &= \phi_C + \Delta x\phi'_C + \frac{\Delta x^2}{2}\phi''_C + \frac{\Delta x^3}{6}\phi'''_C + \frac{\Delta x^4}{24}\phi^{iv}_C + \frac{\Delta x^5}{120}\phi^{v}_C + \cdots \\
\phi_W &= \phi_C - \Delta x\phi'_C + \frac{\Delta x^2}{2}\phi''_C - \frac{\Delta x^3}{6}\phi'''_C + \frac{\Delta x^4}{24}\phi^{iv}_C - \frac{\Delta x^5}{120}\phi^{v}_C + \cdots
\end{aligned} \tag{11.68}$$

Using the above two sets of equations, the following is obtained:

$$\begin{aligned}\phi_C - 2\phi_e + \phi_E &= \phi_C - 2\phi_C - \Delta x\phi'_C - \frac{\Delta x^2}{4}\phi''_C - \frac{\Delta x^3}{24}\phi'''_C - \frac{\Delta x^4}{192}\phi^{iv}_C - \frac{\Delta x^5}{1920}\phi^{v}_C + \cdots \\ &\quad + \phi_C + \Delta x\phi'_C + \frac{\Delta x^2}{2}\phi''_C + \frac{\Delta x^3}{6}\phi'''_C + \frac{\Delta x^4}{24}\phi^{iv}_C + \frac{\Delta x^5}{120}\phi^{v}_C + \cdots \\ &= \frac{\Delta x^2}{4}\phi''_C + \frac{\Delta x^3}{8}\phi'''_C + \frac{7\Delta x^4}{192}\phi^{iv}_C + \frac{15\Delta x^5}{1920}\phi^{v}_C + \cdots \\ \phi_W - 2\phi_w + \phi_C &= \frac{\Delta x^2}{4}\phi''_C - \frac{\Delta x^3}{8}\phi'''_C + \frac{7\Delta x^4}{192}\phi^{iv}_C - \frac{15\Delta x^5}{1920}\phi^{v}_C + \cdots\end{aligned} \tag{11.69}$$

Then $\left(\phi''_e - \phi''_w\right)$ is computed as

$$\frac{\Delta x^2}{8}\left(\phi''_e - \phi''_w\right) = \frac{\Delta x^3}{8}\phi'''_C + \frac{\Delta x^5}{128}\phi^{v}_C + \cdots \tag{11.70}$$

Substituting back, Eq. (11.66) is transformed to

$$\frac{1}{2}(\phi_E + \phi_C) - \frac{1}{2}(\phi_C + \phi_W) = \phi_e - \phi_w + \frac{\Delta x^3}{8}\phi'''_C + \frac{\Delta x^5}{128}\phi^{v}_C + \cdots \tag{11.71}$$

Dividing throughout by Δx, the truncation error associated with the calculation of the gradient is obtained as

$$TE = \frac{\Delta x^2}{8}\phi'''_C + \frac{\Delta x^4}{128}\phi^{v}_C + \cdots \tag{11.72}$$

which indicates that the method is second order accurate.

11.4 Numerical Stability

The confusion related to truncation error (first order for the physical solution and second order for the unphysical one) has led many workers to extrapolate that since central differencing of the diffusion term is so accurate, then central differencing of the convection term should also be similarly accurate. Of course, as seen in the previous sections, central differencing of the convection term can lead to unphysical solutions. The cause for this behavior is that the central difference scheme has no inherent *convective stability* when applied to derivatives of *odd* order like the convection term.

Leonard [9] advanced the *convective stability* concept as an explanation to oscillation in numerical solutions generated by applying the central difference scheme to convection dominated flows. Leonard used the general one dimensional unsteady convection-diffusion equation with constant velocity given by

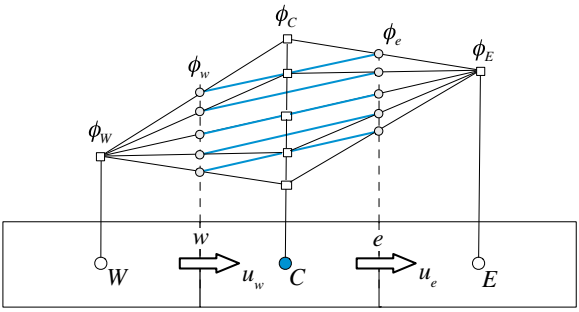


Fig. 11.9 Insensitivity of CD convection term to the values of $\boldsymbol{\phi}_C$

$$\frac{\partial(\rho\phi)}{\partial t} = -\frac{\partial(\rho u\phi)}{\partial x} + \frac{\partial}{\partial x}\left(\Gamma^{\phi}\frac{\partial\phi}{\partial x}\right) + Q^{\phi} \tag{11.73}$$

to describe this concept. When applying this equation over the element of centroid C shown in Fig. 11.9, its left hand side (*LHS*) represents the rate of change of ϕ_C within the control cell per unit time, and its right hand side (*RHS*) represents the net influx across the element surface and source terms within the element that are affecting the value of ϕ_C. If there were numerical errors in the *RHS* then the value of ϕ_C calculated from Eq. (11.73) would either increase or decrease depending on the scheme used in the discretization process. In an unstable scheme, a small deviation from the correct value of ϕ_C gives a corresponding increase/decrease in the net influx represented by the *RHS*. When an iterative procedure is used as part of the solution mechanism, an increase/decrease in the net influx will further increase/decrease the value of ϕ_C at each subsequent step of the iterative process. In a stable scheme this change in ϕ_C due to errors in the *RHS* should feed back negatively into the *RHS* as a self correction device. Clearly, for this kind of numerical stability, the *RHS* should satisfy

$$\frac{\partial(RHS)}{\partial\phi_C} < 0 \tag{11.74}$$

indicating that the sensitivity to ϕ_C of the combination of the modeled terms on the right hand side of Eq. (11.73) should be negative. In this case, an increase/decrease in ϕ_C will correspond to a decrease/increase in the influx, which will in turn pushes ϕ_C downward/upward towards its correct value. However, stability should not be confused with boundedness or accuracy. A stable numerical scheme could actually be unbounded giving rise to over/under shoots and oscillations/wiggles or be very diffusive and give results that are of low accuracy. The stability here refers to controlling the numerical error to remain bounded in order for it not to increase indefinitely as was the case with the central difference scheme where the normalized

value of ϕ_C (Fig. 11.4) was found to vary from $+\infty$ to $-\infty$ while varying Pe from $-\infty$ to $+\infty$, even though the correct value of ϕ_C should vary between 1 and 0. It so happens that the upwind scheme possesses both the boundedness and stability characteristics.

With this in mind, the general discretized form of the *RHS* of Eq. (11.73) is given by

$$RHS = -(\rho u \Delta y)_e \phi_e + (\rho u \Delta y)_w \phi_w + \left[\left(\Gamma^\phi \frac{\Delta y}{\delta x} \right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{\Delta y}{\delta x} \right)_w (\phi_C - \phi_W) \right] + Q_C^\phi V_C \tag{11.75}$$

Thus using the central difference scheme, Eq. (11.75) becomes

$$RHS_{CD} = -(\rho u \Delta y)_e \frac{(\phi_E + \phi_C)}{2} + (\rho u \Delta y)_w \frac{(\phi_C + \phi_W)}{2} + \left[\left(\Gamma^\phi \frac{\Delta y}{\delta x} \right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{\Delta y}{\delta x} \right)_w (\phi_C - \phi_W) \right] + Q_C^\phi V_C \tag{11.76}$$

Analyzing the central difference scheme using the above criteria, it is found that for the diffusion term the sensitivity is given by

$$\frac{\partial \left(RHS_{CD}^{Diff} \right)}{\partial \phi_C} = -2\Gamma^\phi \frac{\Delta y}{\delta x} \tag{11.77}$$

which is negative since Γ^ϕ is positive, indicating a stable scheme. However, for the convective term the sensitivity equation gives

$$\frac{\partial \left(RHS_{CD}^{Conv} \right)}{\partial \phi_C} = -\frac{1}{2}(\dot{m}_e + \dot{m}_w) \tag{11.78}$$

which is equal to zero for steady flows but not necessarily for unsteady flows. For unsteady situations, its value will be positive for decelerating flows. In a general flow, such regions act as wiggle sources and can easily lead to a total numerical catastrophe when the Péclet number is large enough. Even for steady flows, as its value is zero, it cannot feed back into the equation to act as a self correction device. Equation (11.78) also indicates that for steady flow the net convective flux computed with the CD scheme is independent of the value of ϕ_C. Therefore, as shown in Fig. 11.9, the different possible values of ϕ_C will result in the same net convective flux over the element of centroid C.

With the upwind scheme, the *RHS* can be obtained from Eq. (11.36) as

$$RHS_{Upwind} = -\|\dot{m}_e, 0\|\phi_C + \|-\dot{m}_e, 0\|\phi_E - \|\dot{m}_w, 0\|\phi_C + \|-\dot{m}_w, 0\|\phi_W + \left(\Gamma^\phi \frac{S}{\delta x}\right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{S}{\delta x}\right)_w (\phi_C - \phi_W) + Q_C^\phi \tag{11.79}$$

The sensitivity is expected to be negative since the scheme was found to be stable for all Péclet number, indeed

$$\frac{\partial\left(RHS_{Upwind}^{Conv}\right)}{\partial\phi_C} = -\|\dot{m}_e, 0\| - \|\dot{m}_w, 0\| \tag{11.80}$$

which is negative or equal to zero for all flows. It will be equal to zero when both mass flow rates are negative, a situation that does not arise in a one dimensional situation of constant cross-sectional area. When added to the false diffusion introduced by the first order approximation, Eq. (11.80) indicates that the scheme is very stable. However, this stability is achieved at the expense of accuracy, as was demonstrated in the previous section.

For the downwind scheme the *RHS* is given by

$$RHS_{Downwind} = -\|\dot{m}_e, 0\|\phi_E + \|-\dot{m}_e, 0\|\phi_C - \|\dot{m}_w, 0\|\phi_W + \|-\dot{m}_w, 0\|\phi_C + \left(\Gamma^\phi \frac{S}{\delta x}\right)_e (\phi_E - \phi_C) + \left(\Gamma^\phi \frac{S}{\delta x}\right)_w (\phi_W - \phi_C) \tag{11.81}$$

The sensitivity is given by

$$\frac{\partial\left(RHS_{Downwind}^{Conv}\right)}{\partial\phi_C} = \|-\dot{m}_e, 0\| + \|-\dot{m}_w, 0\| \tag{11.82}$$

which is always positive or equal to zero for all flows. When this is added to the anti-diffusion effects it gives a highly unstable scheme.

11.5 Higher Order Upwind Schemes

The previous sections demonstrated that both the upwind and central difference schemes have severe limitations, the former because of its poor accuracy due to numerical diffusion, and the latter because of its instability also known as numerical dispersion error. These shortcomings have provoked a great deal of research to improve the accuracy and stability of advection schemes by using higher order

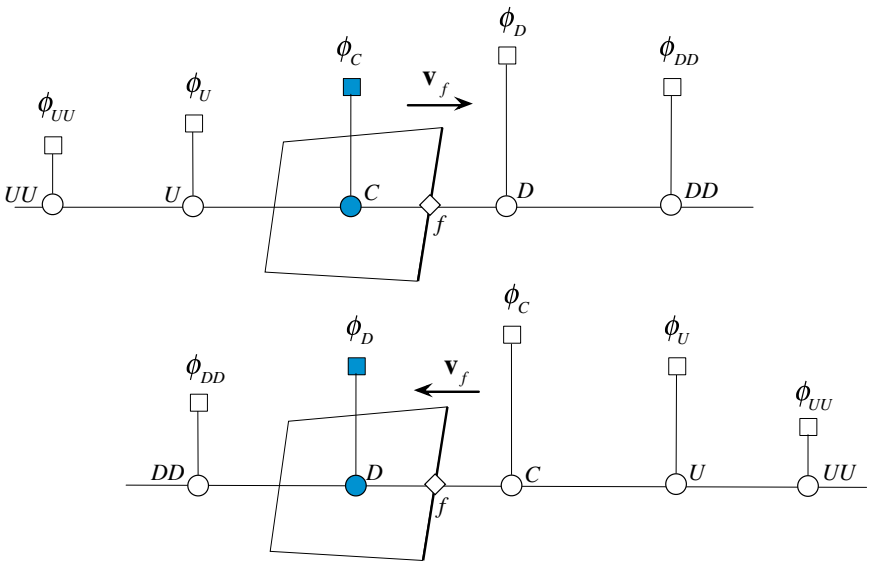


Fig. 11.10 A schematic showing the *UU*, *U*, *C*, *D* and *DD* node locations used in describing convection schemes

upwind biased interpolation profiles. These higher-order schemes aim at producing at least a second order accurate solutions, while being unconditionally stable.

To underline the conservation property which associates fluxes with cell faces, not nodes, in what follows *D*, *C*, and *U* will be used to denote the Downwind, Upwind, and far Upwind nodes at any particular face. In addition, *DD* and *UU* denote the nodes downstream and upstream of *D* and *U*, respectively. The corresponding values at these locations are denoted by $\phi_{DD}, \phi_D, \phi_C, \phi_U$ and ϕ_{UU}, respectively. This notation is displayed graphically in Fig. 11.10 for the cases when the velocity at the face is either positive ($\rightarrow$) or negative ($\leftarrow$).

11.5.1 Second Order Upwind Scheme

A second order scheme, requires the use of a linear profile, as was the case with the central difference scheme. However, instead of a symmetric profile, an upwind biased stencil is used [10]. As depicted in Fig. 11.11, the linear profile is constructed by employing the ϕ values at nodes *C* and *U*. Therefore the value at the face is actually calculated by extrapolation rather than interpolation.

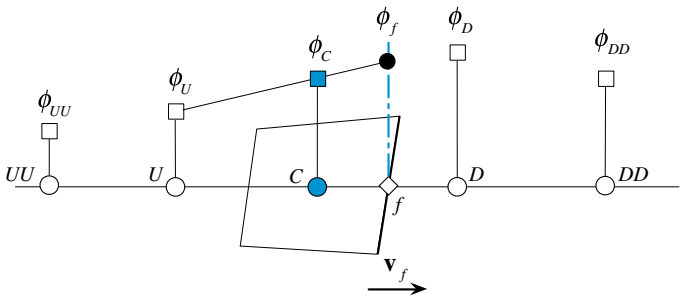


Fig. 11.11 The Second Order Upwind (SOU) scheme profile

11.5.2 The Interpolation Profile

Starting with the linear profile

$$\phi(x) = k_0 + k_1(x - x_C) \tag{11.83}$$

and fitting it to the nodal values at x_C and x_U where the ϕ values are ϕ_C and ϕ_U, respectively, the profile becomes

$$\phi(x) = \phi_C + \frac{\phi_C - \phi_U}{x_C - x_U}(x - x_C) \tag{11.84}$$

Using the above equation, the ϕ value at face f, shown in Fig. 11.11, is given by

$$\phi_f = \phi(x_f) = \phi_C + \frac{\phi_C - \phi_U}{x_C - x_U}(x_f - x_C) \tag{11.85}$$

which, for a uniform grid reduces to

$$\phi_f = \frac{3}{2}\phi_C - \frac{1}{2}\phi_U \tag{11.86}$$

11.5.3 The Discretized Equation

Using this profile to approximate the interface values in the discretized one dimensional convection diffusion equation (Eq. 11.16), the fluxes at the faces are obtained as

$$\begin{aligned}\dot{m}_e\phi_e &= \left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_W\right)\|\dot{m}_e, 0\| - \left(\frac{3}{2}\phi_E - \frac{1}{2}\phi_{EE}\right)\|-\dot{m}_e, 0\| \\ \dot{m}_w\phi_w &= \left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_E\right)\|\dot{m}_w, 0\| - \left(\frac{3}{2}\phi_W - \frac{1}{2}\phi_{WW}\right)\|-\dot{m}_w, 0\|\end{aligned} \tag{11.87}$$

Substitution of these fluxes in Eq. (11.16), the discretized form is transformed to

$$\begin{aligned}&\left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_W\right)\|\dot{m}_e, 0\| - \left(\frac{3}{2}\phi_E - \frac{1}{2}\phi_{EE}\right)\|-\dot{m}_e, 0\| + \left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_E\right)\|\dot{m}_w, 0\| \\ &- \left(\frac{3}{2}\phi_W - \frac{1}{2}\phi_{WW}\right)\|-\dot{m}_w, 0\| - \left[\left(\Gamma^\phi \frac{S}{\delta x}\right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{S}{\delta x}\right)_w (\phi_C - \phi_W)\right] = 0\end{aligned} \tag{11.88}$$

which can be modified into the form

$$a_C\phi_C + a_E\phi_E + a_W\phi_W + a_{EE}\phi_{EE} + a_{WW}\phi_{WW} = 0 \tag{11.89}$$

where

$$\begin{aligned}a_E &= FluxF_e = -\Gamma_e^\phi \frac{S_e}{\delta x_e} - \frac{3}{2}\|-\dot{m}_e, 0\| - \frac{1}{2}\|\dot{m}_w, 0\| \quad a_{EE} = FluxF_{ee} = \frac{1}{2}\|-\dot{m}_e, 0\| \\ a_W &= FluxF_w = -\Gamma_w^\phi \frac{S_w}{\delta x_w} - \frac{3}{2}\|-\dot{m}_w, 0\| - \frac{1}{2}\|\dot{m}_e, 0\| \quad a_{WW} = FluxF_{ww} = \frac{1}{2}\|-\dot{m}_w, 0\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f \\ &= \Gamma_e^\phi \frac{S_e}{\delta x_e} + \Gamma_w^\phi \frac{S_w}{\delta x_w} + \frac{3}{2}\|\dot{m}_e, 0\| + \frac{3}{2}\|\dot{m}_w, 0\| \\ &= -(a_E + a_W + a_{EE} + a_{WW}) + (\dot{m}_e + \dot{m}_w)\end{aligned} \tag{11.90}$$

11.5.4 Truncation Error

Following a procedure similar to the one used with the central difference scheme, the truncation error is found to be

$$TE = -\frac{3}{8}\Delta x^2\phi_C''' - \frac{1}{4}\Delta x^3\phi_C^{iv} + \cdots \tag{11.91}$$

indicating second order accuracy.

11.5.5 Stability Analysis

To check the stability of the SOU scheme, the discretized convective fluxes are substituted in Eq. (11.75) resulting in the following $RHS_{convection}$ term:

$$RHS_{convection} = -\left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_W\right)\|\dot{m}_e, 0\| + \left(\frac{3}{2}\phi_E - \frac{1}{2}\phi_{EE}\right)\|-\dot{m}_e, 0\| - \left(\frac{3}{2}\phi_C - \frac{1}{2}\phi_E\right)\|\dot{m}_w, 0\| + \left(\frac{3}{2}\phi_W - \frac{1}{2}\phi_{WW}\right)\|-\dot{m}_w, 0\| \tag{11.92}$$

The rate of change of this term with respect to ϕ_C is found as

$$\frac{\partial(RHS_{convection})}{\partial\phi_C} = -\frac{3}{2}\|\dot{m}_e, 0\| - \frac{3}{2}\|\dot{m}_w, 0\| \tag{11.93}$$

which is always negative indicating a stable scheme, i.e., the solution error will always remain under control. It should be kept in mind that this is true for the conditions stated in the derivations and not for the general case when the velocity is not constant.

11.5.6 The QUICK Scheme

The Quadratic Upstream Interpolation for Convective Kinematics (QUICK) scheme was developed by Leonard [2]. The method is based on interpolating the value of the dependent variable at each face of the element by using a quadratic polynomial biased toward the upstream direction, as shown in Fig. 11.12. The interpolated value is used to calculate the convective term in the governing equations for the dependent variable. The calculation of the values of the dependent variable at a cell face is detailed next.

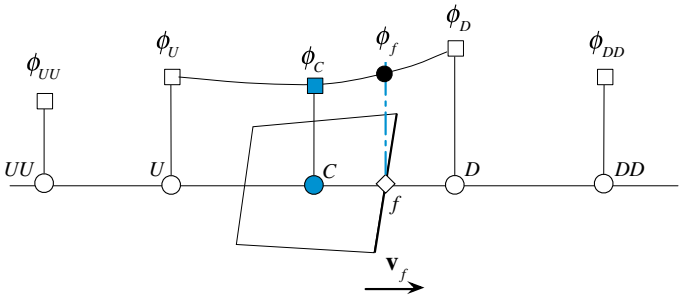


Fig. 11.12 The QUICK scheme profile

11.5.7 The Interpolation Profile

In its original form, the QUICK scheme required the use of a general multi dimensional second degree polynomial for the calculation of the unknown variable ϕ. However it is generally sufficient to treat the flow as locally one dimensional [11] and to use a second order one dimensional profile in each coordinate direction. For the one dimensional convection-diffusion problem under discussion, the value of ϕ is computed using

$$\phi = k_0 + k_1 x + k_2 x^2, \tag{11.94}$$

subject to

$$\phi = \begin{cases} \phi_U & at\ x = x_U \\ \phi_C & at\ x = x_C \\ \phi_D & at\ x = x_D \end{cases} \tag{11.95}$$

Applying the conditions given by Eq. (11.95), the profile for ϕ is found to be

$$\phi = \phi_U + \frac{(x - x_U)(x - x_C)}{(x_D - x_U)(x_D - x_C)}(\phi_D - \phi_U) + \frac{(x - x_U)(x - x_D)}{(x_C - x_U)(x_C - x_D)}(\phi_C - \phi_U) \tag{11.96}$$

For the case of a uniform grid, the value of ϕ at the cell face f reduces to

$$\phi_f = \frac{\phi_C + \phi_D}{2} - \frac{\phi_D - 2\phi_C + \phi_U}{8} \tag{11.97}$$

Thus, the convective fluxes at the element faces can be computed as

$$\begin{aligned} \dot{m}_e\phi_e &= \left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_W + \frac{3}{8}\phi_E\right)\|\dot{m}_e, 0\| - \left(\frac{3}{4}\phi_E - \frac{1}{8}\phi_{EE} + \frac{3}{8}\phi_C\right)\|-\dot{m}_e, 0\| \\ \dot{m}_w\phi_w &= \left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_E + \frac{3}{8}\phi_W\right)\|\dot{m}_w, 0\| - \left(\frac{3}{4}\phi_W - \frac{1}{8}\phi_{WW} + \frac{3}{8}\phi_C\right)\|-\dot{m}_w, 0\| \end{aligned} \tag{11.98}$$

Substituting back in the one dimensional convection-diffusion equation [Eq. (11.16)], the discretized form becomes

$$\begin{aligned} &\left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_W + \frac{3}{8}\phi_E\right)\|\dot{m}_e, 0\| - \left(\frac{3}{4}\phi_E - \frac{1}{8}\phi_{EE} + \frac{3}{8}\phi_C\right)\|-\dot{m}_e, 0\| \\ &+ \left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_E + \frac{3}{8}\phi_W\right)\|\dot{m}_w, 0\| - \left(\frac{3}{4}\phi_W - \frac{1}{8}\phi_{WW} + \frac{3}{8}\phi_C\right)\|-\dot{m}_w, 0\| \\ &- \left[\left(\Gamma^\phi \frac{S}{\delta x}\right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{S}{\delta x}\right)_w (\phi_C - \phi_W)\right] = 0 \end{aligned} \tag{11.99}$$

which can be modified into the form

$$a_C\phi_C + a_E\phi_E + a_W\phi_W + a_{EE}\phi_{EE} + a_{WW}\phi_{WW} = 0 \tag{11.100}$$

with the coefficients given by

$$\begin{aligned}
a_E &= FluxF_e = -\Gamma_e^\phi \frac{S_e}{\delta x_e} - \frac{3}{4}\|-\dot{m}_e, 0\| + \frac{3}{8}\|\dot{m}_e, 0\| - \frac{1}{8}\|\dot{m}_w, 0\| \\
a_W &= FluxF_w = -\Gamma_w^\phi \frac{S_w}{\delta x_w} - \frac{3}{4}\|-\dot{m}_w, 0\| + \frac{3}{8}\|\dot{m}_w, 0\| - \frac{1}{8}\|\dot{m}_e, 0\| \\
a_{EE} &= FluxF_{ee} = \frac{1}{8}\|-\dot{m}_e, 0\| \quad a_{WW} = FluxF_{ww} = \frac{1}{8}\|-\dot{m}_w, 0\| \\
a_C &= \sum_{f\sim nb(C)} FluxC_f \\
&= \Gamma_e^\phi \frac{S_e}{\delta x_e} + \Gamma_w^\phi \frac{S_w}{\delta x_w} + \frac{3}{4}\|\dot{m}_e, 0\| - \frac{3}{8}\|-\dot{m}_e, 0\| + \frac{3}{4}\|\dot{m}_w, 0\| - \frac{3}{8}\|-\dot{m}_w, 0\| \\
&= -\left(a_E + a_W + a_{WW} + a_{EE}\right) + \left(\dot{m}_e + \dot{m}_w\right)
\end{aligned} \tag{11.101}$$

11.5.8 Truncation Error

Again following the procedure described above, the truncation error is found to be

$$TE = \frac{1}{16}\Delta x^3 \phi_C^{iv} - \frac{3}{128}\Delta x^4 \phi_C^{v} + \cdots \tag{11.102}$$

which is clearly third order accurate.

11.5.9 Stability Analysis

To check the stability of the QUICK scheme, the discretized convective fluxes are substituted in Eq. (11.75) resulting in the following $RHS_{convection}$ term:

$$\begin{aligned}
RHS_{convection} = &-\left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_W + \frac{3}{8}\phi_E\right)\|\dot{m}_e, 0\| + \left(\frac{3}{4}\phi_E - \frac{1}{8}\phi_{EE} + \frac{3}{8}\phi_C\right)\|-\dot{m}_e, 0\| \\
&-\left(\frac{3}{4}\phi_C - \frac{1}{8}\phi_E + \frac{3}{8}\phi_W\right)\|\dot{m}_w, 0\| + \left(\frac{3}{4}\phi_W - \frac{1}{8}\phi_{WW} + \frac{3}{8}\phi_C\right)\|-\dot{m}_w, 0\|
\end{aligned} \tag{11.103}$$

The rate of change of this term with respect to ϕ_C is found to be given by

$$\frac{\partial(RHS_{convection})}{\partial\phi_C} = -\frac{3}{8}\|\dot{m}_e, 0\| - \frac{3}{8}\|\dot{m}_w, 0\| - \frac{3}{8}(\dot{m}_e + \dot{m}_w) \tag{11.104}$$

which for a uniform velocity is always negative indicating a stable scheme. However this does not guarantee solution boundedness especially in the general case of nonuniform velocity.

11.5.10 The FROMM Scheme

The FROMM scheme [12] fits a linear profile between the far Upwind (U) and Downwind (D) nodes straddling the interface. As depicted in Fig. 11.13, instead of a symmetric profile, an upwind biased stencil is used to calculate the value of the dependent variable ϕ at face f. Based on the adopted profile, ϕ_U, ϕ_C, and ϕ_D are assumed to be collinear.

11.5.11 The Interpolation Profile

Starting with the linear profile

$$\phi(x) = k_0 + k_1(x - x_C) \tag{11.105}$$

and fitting it to the nodal values at x_D and x_U where the ϕ values are ϕ_D and ϕ_U, respectively, the profile becomes

$$\phi(x) = \phi_U + \frac{\phi_D - \phi_U}{x_D - x_U}(x - x_U) \tag{11.106}$$

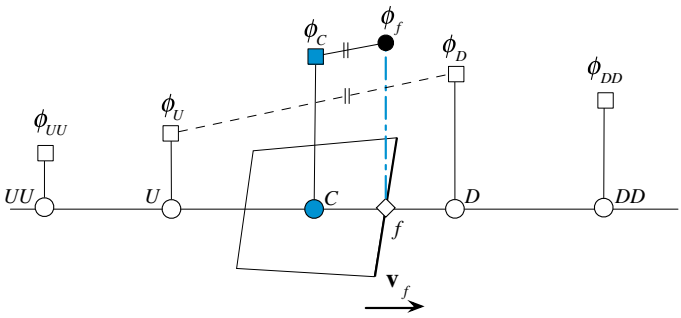


Fig. 11.13 The FROMM scheme profile

Using the above equation, the ϕ value at upwind node C is obtained as

$$\phi_C = \phi_U + \frac{\phi_D - \phi_U}{x_D - x_U}(x_C - x_U) \tag{11.107}$$

For the case of a uniform grid the above equation becomes

$$\phi_C = \frac{\phi_D + \phi_U}{2} \tag{11.108}$$

The required value at the face f, shown in Fig. 11.13, is given by

$$\phi_f = \phi(x_f) = \phi_U + \frac{\phi_D - \phi_U}{x_D - x_U}(x_f - x_U) = \phi_C + \frac{x_f - x_C}{x_D - x_U}(\phi_D - \phi_U) \tag{11.109}$$

which, for a uniform grid reduces to

$$\phi_f = \phi_C + \frac{\phi_D - \phi_U}{4} \tag{11.110}$$

The final expression for ϕ_f given in Eq. (11.109) was obtained by invoking Eq. (11.107).

11.5.12 The Discretized Equation

Using this profile to approximate the interface values in the discretized one dimensional convection diffusion equation (Eq. 11.16), the fluxes at the faces are obtained as

$$\begin{aligned}
\dot{m}_e\phi_e &= \left(\phi_C - \frac{1}{4}\phi_w + \frac{1}{4}\phi_E\right)\|\dot{m}_e, 0\| - \left(\phi_E - \frac{1}{4}\phi_{EE} + \frac{1}{4}\phi_C\right)\|-\dot{m}_e, 0\| \\
\dot{m}_w\phi_w &= \left(\phi_C - \frac{1}{4}\phi_E + \frac{1}{4}\phi_W\right)\|\dot{m}_w, 0\| - \left(\phi_W - \frac{1}{4}\phi_{WW} + \frac{1}{4}\phi_C\right)\|-\dot{m}_w, 0\|
\end{aligned} \tag{11.111}$$

Substitution of these fluxes in Eq. (11.16), the discretized form is transformed to

$$\begin{aligned}
&\left(\phi_C - \frac{1}{4}\phi_W + \frac{1}{4}\phi_E\right)\|\dot{m}_e, 0\| - \left(\phi_E - \frac{1}{4}\phi_{EE} + \frac{1}{4}\phi_C\right)\|-\dot{m}_e, 0\| \\
&+ \left(\phi_C - \frac{1}{4}\phi_E + \frac{1}{4}\phi_W\right)\|\dot{m}_w, 0\| - \left(\phi_W - \frac{1}{4}\phi_{WW} + \frac{1}{4}\phi_C\right)\|-\dot{m}_w, 0\| \\
&- \left[\left(\Gamma^\phi \frac{S}{\delta x}\right)_e (\phi_E - \phi_C) - \left(\Gamma^\phi \frac{S}{\delta x}\right)_w (\phi_C - \phi_W)\right] = 0
\end{aligned} \tag{11.112}$$

which can be modified into the form

$$a_C\phi_C + a_E\phi_E + a_W\phi_W + a_{EE}\phi_{EE} + a_{WW}\phi_{WW} = 0 \tag{11.113}$$

where

$$\begin{aligned}
a_E &= FluxF_e = -\Gamma_e^\phi \frac{S_e}{\delta x_e} + \frac{1}{4}\|\dot{m}_e, 0\| - \|-\dot{m}_e, 0\| - \frac{1}{4}\|\dot{m}_w, 0\| \\
a_W &= FluxF_w = -\Gamma_w^\phi \frac{S_w}{\delta x_w} + \frac{1}{4}\|\dot{m}_w, 0\| - \|-\dot{m}_w, 0\| - \frac{1}{4}\|\dot{m}_e, 0\| \\
a_{EE} &= FluxF_{ee} = \frac{1}{4}\|-\dot{m}_e, 0\| \quad a_{WW} = FluxF_{ww} = \frac{1}{4}\|-\dot{m}_w, 0\| \\
a_C &= \sum_{f\sim nb(C)} FluxC_f \\
&= \Gamma_e^\phi \frac{S_e}{\delta x_e} + \Gamma_w^\phi \frac{S_w}{\delta x_w} + \|\dot{m}_e, 0\| - \frac{1}{4}\|-\dot{m}_e, 0\| + \|\dot{m}_w, 0\| - \frac{1}{4}\|-\dot{m}_w, 0\| \\
&= -(a_E + a_W + a_{EE} + a_{WW}) + (\dot{m}_e + \dot{m}_w)
\end{aligned} \tag{11.114}$$

11.5.13 Truncation Error

Following a procedure similar to the one used with the central difference scheme, the truncation error can be derived to be of $O(\Delta x^2)$, i.e.,

$$TE = O\left(\Delta x^2\right) \tag{11.115}$$

indicating second order accuracy.

11.5.14 Stability Analysis

To check the stability of the FROMM scheme, the discretized convective fluxes are substituted in Eq. (11.75) resulting in the following $RHS_{convection}$ term:

$$\begin{aligned}
RHS_{convection} = &-\left(\phi_C - \frac{1}{4}\phi_W + \frac{1}{4}\phi_E\right)\|\dot{m}_e, 0\| + \left(\phi_E - \frac{1}{4}\phi_{EE} + \frac{1}{4}\phi_C\right)\|-\dot{m}_e, 0\| \\
&- \left(\phi_C - \frac{1}{4}\phi_E + \frac{1}{4}\phi_W\right)\|\dot{m}_w, 0\| + \left(\phi_W - \frac{1}{4}\phi_{WW} + \frac{1}{4}\phi_C\right)\|-\dot{m}_w, 0\|
\end{aligned} \tag{11.116}$$

The rate of change of this term with respect to ϕ_C is found as

$$\frac{\partial(RHS_{convection})}{\partial\phi_C} = -\frac{3}{4}\|\dot{m}_e, 0\| - \frac{3}{4}\|\dot{m}_w, 0\| - \frac{1}{4}(\dot{m}_e + \dot{m}_w) \tag{11.117}$$

which, for a constant velocity field, is always negative indicating a stable scheme, but not for the general case when the velocity is varying.

11.5.15 Comparison of the Various Schemes

By comparing the stability of the various schemes presented so far, it is clear that the most negative (with a coefficient of −3/2) is the one associated with the SOU scheme. This is followed by the upwind (with a coefficient of −1), then FROMM (with a coefficient of −3/4), after that QUICK(with a coefficient of −3/8), and finally the central difference scheme (with a coefficient of zero, i.e., a neutral sensitivity to changes in ϕ_C). The self corrective action discussed above is the sum of both convection and diffusion contributions. The false diffusion produced by the upwind scheme adds to its stability and even though its coefficient is −1, is the most stable scheme. This is demonstrated by the profiles presented in Fig. 11.14, which represent the solutions using the various numerical schemes of the convection-diffusion equation over a domain of length L = 1 and subject to the Dirichlet conditions of $\phi = 1$ at x = 0 and $\phi = 0$ at x = 1. While all solutions are stable at low Péclet number (i.e., Pe = 1, Fig. 11.14a), the CD, FROMM, and QUICK scheme solutions shown in Fig. 11.14b are seen to be wiggly at Pe = 10. At low Pe, the accuracy of the CD and QUICK schemes is comparable and their solutions are very close to the exact solution. The least accurate is the solution produced by the first order upwind scheme. The solution of the FROMM scheme is

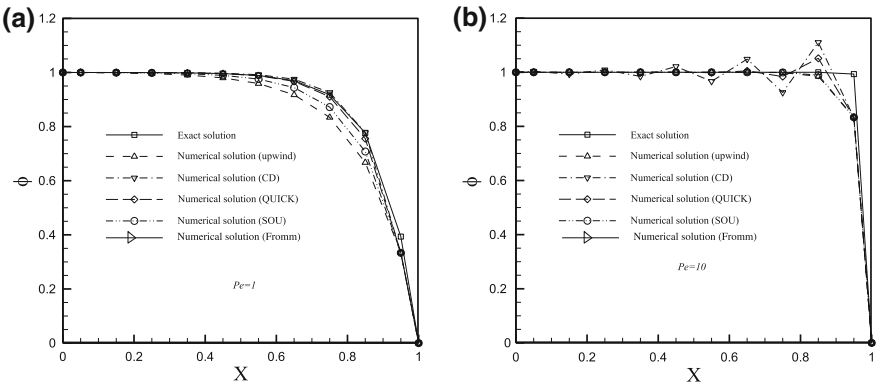


Fig. 11.14 Stability of solutions generated using several convective schemes at two values of cell Péclet number of **a** 1 and **b** 10

more accurate than the SOU scheme, which, in turn, is more accurate than the upwind scheme solution, while both the FROMM and SOU scheme solutions are less accurate than the QUICK scheme solution.

At high *Pe*, the behavior of the schemes changes. The only stable solutions are the ones obtained by the upwind and SOU schemes with their profiles being almost identical indicating that the SOU scheme is still highly diffusive. The CD scheme is wiggly over most of the domain. The FROMM and QUICK schemes, on the other hand, show wiggles but of smaller amplitudes. The reason for these wiggles is the imposed boundary condition at exit from the domain. As the phenomenon is convection dominated, it is affected by upstream values. At exit from the domain, the solution faces an imposed value that it has to satisfy, resulting in a large unexpected gradient causing the over and under shoots to occur. The diffusive upwind and SOU schemes being based on upstream values only, are smooth and do not show any sign of under/overshoots for all *Pe* values as the solution is independent of the imposed value at exit. However the SOU scheme is expected to give rise to oscillations (over/under shoots) in the presence of a high gradient in the domain like in the presence of a shock wave.

11.5.16 Functional Relationships for Uniform and Non-uniform Grids

The various interpolation profiles presented so far have been derived on a one dimensional Cartesian grid. Along a curvilinear coordinate axis, the same functional relationships can be used by replacing the Cartesian x-axis by a curvilinear ζ-axis, as shown in Fig. 11.15. For uniform grid, the functional relationships remain exactly the same, independent of whether a Cartesian or a curvilinear grid system is used. For non-uniform grid, the independent variable x appearing in the functional relationships should be replaced by the more general independent variable ζ, which represents distance along the coordinate axis. If O is the origin (Fig. 11.15), then ζ_U for example is calculated as

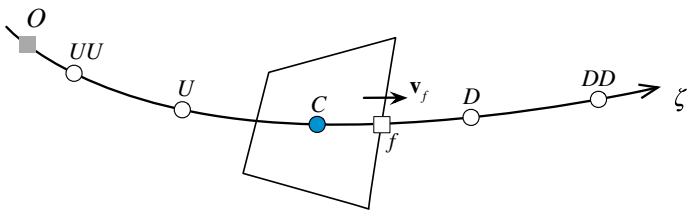


Fig. 11.15 Notation on a curvilinear coordinate axis ζ

Table 11.1 Functional relationships for several convection schemes on uniform and non-uniform grids

Scheme	Uniform grid	Non-uniform grid
Upwind	$\phi_f = \phi_C$	$\phi_f = \phi_C$
Downwind	$\phi_f = \phi_D$	$\phi_f = \phi_D$
CD	$\phi_f = 0.5(\phi_C + \phi_D)$	$\phi_f = \phi_C + \left(\frac{\phi_D - \phi_C}{\zeta_D - \zeta_C}\right)(\zeta_f - \zeta_C)$
SOU	$\phi_f = \frac{3}{2}\phi_C - \frac{1}{2}\phi_U$	$\phi_f = \phi_C + \left(\frac{\phi_C - \phi_U}{\zeta_C - \zeta_U}\right)(\zeta_f - \zeta_C)$
QUICK	$\phi_f = \frac{3}{4}\phi_C - \frac{1}{8}\phi_U + \frac{3}{8}\phi_D$	$\phi_f = \phi_U + \frac{(\zeta_f - \zeta_U)(\zeta_f - \zeta_C)}{(\zeta_D - \zeta_U)(\zeta_D - \zeta_C)}(\phi_D - \phi_U) + \frac{(\zeta_f - \zeta_U)(\zeta_f - \zeta_D)}{(\zeta_C - \zeta_U)(\zeta_C - \zeta_D)}(\phi_C - \phi_U)$
FROMM	$\phi_f = \phi_C + \frac{1}{4}(\phi_D - \phi_U)$	$\phi_f = \phi_C + \frac{\zeta_f - \zeta_C}{\zeta_D - \zeta_U}(\phi_D - \phi_U)$

$$\begin{aligned} \zeta_U &= \zeta_{UU} + (\zeta_U - \zeta_{UU}) \\ \zeta_{UU} &= \sqrt{(x_{UU} - x_O)^2 + (y_{UU} - y_O)^2 + (z_{UU} - z_O)^2} \\ \zeta_U - \zeta_{UU} &= \sqrt{(x_U - x_{UU})^2 + (y_U - y_{UU})^2 + (z_U - z_{UU})^2} \end{aligned} \tag{11.118}$$

Therefore distances are calculated by subdividing the curvilinear line into a number of straight lines and in general the following applies:

$$\zeta_1 - \zeta_2 = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{11.119}$$

Adopting a general coordinate system, the functional relationships for the various schemes presented so far on uniform and non-uniform grids can be derived to be as shown in Table 11.1.

11.6 Steady Two Dimensional Advection

The steady two dimensional advection equation is given by

$$\nabla \cdot (\rho \mathbf{v} \phi) = 0. \tag{11.120}$$

Integrating over the two-dimensional element of volume V_C shown in Fig. 11.16, using the divergence theorem, and replacing the surface integral by a summation over the element faces, Eq. (11.120) becomes

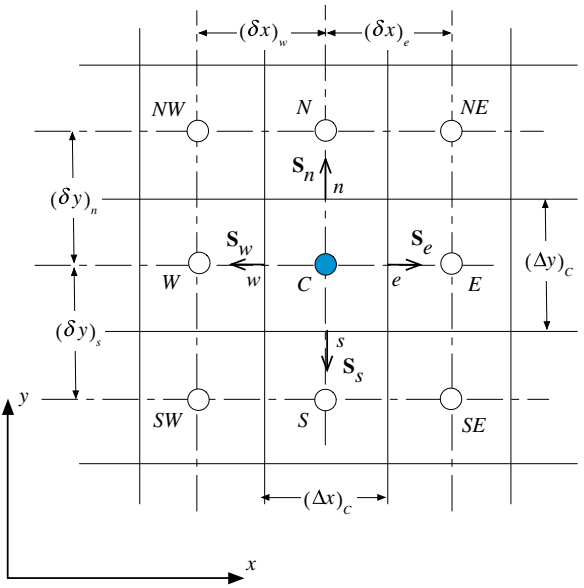


Fig. 11.16 Notation for a two dimensional Cartesian grid system

$$\sum_{f\sim nb(C)} \left(\int_f \mathbf{J}^{\phi,C} \cdot d\mathbf{S} \right) = 0 \tag{11.121}$$

Using a single Gaussian point for the face integral, the left hand side of Eq. (11.121) is transformed to

$$\sum_{f\sim nb(C)} \left(\int_f \mathbf{J}^{\phi,C} \cdot d\mathbf{S} \right) = \sum_{f\sim nb(C)} \left(\mathbf{J}_f^{\phi,C} \cdot \mathbf{S}_f \right) = \sum_{f\sim nb(C)} (\rho \mathbf{v}\phi)_f \cdot \mathbf{S}_f \tag{11.122}$$

Substitution of Eq. (11.122) in Eq. (11.121) yields

$$\sum_{f\sim nb(C)} (\rho \mathbf{v}\phi)_f \cdot \mathbf{S}_f = 0 \tag{11.123}$$

The full discretized form of Eq. (11.120) over a Cartesian grid is obtained as

$$\left(\overbrace{\rho u \Delta y \phi}^{\dot{m}_e} \right)_e - \left(\overbrace{\rho u \Delta y \phi}^{-\dot{m}_w} \right)_w + \left(\overbrace{\rho v \Delta x \phi}^{\dot{m}_n} \right)_n - \left(\overbrace{\rho v \Delta x \phi}^{-\dot{m}_s} \right)_s = 0 \tag{11.124}$$

Adopting an upwind scheme in each coordinate direction by treating the flow as locally one dimensional, Eq. (11.124) becomes

$$a_C\phi_C + a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S = 0 \tag{11.125}$$

with the coefficients given by

$$\begin{aligned}
a_E &= FLuxF_e = -\|-\dot{m}_e, 0\| \qquad a_W = FLuxF_w = -\|-\dot{m}_w, 0\| \\
a_N &= FLuxF_n = -\|-\dot{m}_n, 0\| \qquad a_S = FLuxF_s = -\|-\dot{m}_s, 0\| \\
a_C &= \sum_{f\sim nb(C)} FluxC_f = \|\dot{m}_e, 0\| + \|\dot{m}_w, 0\| + \|\dot{m}_n, 0\| + \|\dot{m}_n, 0\| \\
&= -\underbrace{(a_E + a_W + a_N + a_S)}_{\sum_{F\sim NB(C)} a_F} + \dot{m}_e + \dot{m}_w + \dot{m}_n + \dot{m}_s
\end{aligned} \tag{11.126}$$

If the QUICK scheme is used, the discretized equation is changed into

$$\begin{aligned}
a_C\phi_C + a_E\phi_E + a_W\phi_W + a_{EE}\phi_{EE} + a_{WW}\phi_{WW} \\
+ a_N\phi_N + a_S\phi_S + a_{NN}\phi_{NN} + a_{SS}\phi_{SS} = 0
\end{aligned} \tag{11.127}$$

with its coefficients computed as

$$\begin{aligned}
a_E &= FluxF_e = -\frac{3}{4}\|-\dot{m}_e, 0\| + \frac{3}{8}\|\dot{m}_e, 0\| - \frac{1}{8}\|\dot{m}_w, 0\| \\
a_W &= FluxF_w = -\frac{3}{4}\|-\dot{m}_w, 0\| + \frac{3}{8}\|\dot{m}_w, 0\| - \frac{1}{8}\|\dot{m}_e, 0\| \\
a_{EE} &= FluxF_{ee} = \frac{1}{8}\|-\dot{m}_e, 0\| \qquad a_{WW} = FluxF_{ww} = \frac{1}{8}\|-\dot{m}_w, 0\| \\
a_N &= FluxF_n = -\frac{3}{4}\|-\dot{m}_n, 0\| + \frac{3}{8}\|\dot{m}_n, 0\| - \frac{1}{8}\|\dot{m}_s, 0\| \\
a_S &= FluxF_s = -\frac{3}{4}\|-\dot{m}_s, 0\| + \frac{3}{8}\|\dot{m}_s, 0\| - \frac{1}{8}\|\dot{m}_n, 0\| \\
a_{NN} &= FluxF_{nn} = \frac{1}{8}\|-\dot{m}_n, 0\| \qquad a_{SS} = FluxF_{ss} = \frac{1}{8}\|-\dot{m}_s, 0\| \\
a_C &= \sum_{f\sim nb(C)} FluxC_f = -\sum_{F\sim NB(C)} a_F + (\dot{m}_e + \dot{m}_w + \dot{m}_n + \dot{m}_s)
\end{aligned} \tag{11.128}$$

Both discretized equations are used to solve the pure advection of a step profile in an oblique velocity field problem. The physical domain is schematically depicted in Fig. 11.17a. It represents a square domain with a property ϕ being convected in a field with a velocity $\mathbf{v}(1,1)$. The value of ϕ is 1 on the left side of the domain and 0

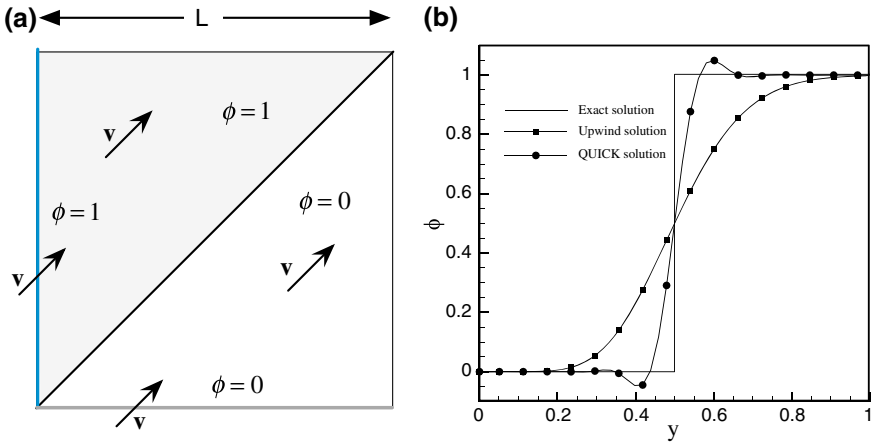


Fig. 11.17 **a** Physical domain and **b** ϕ profiles along the vertical centerline of the domain for pure advection of a step profile in an oblique velocity field

on the bottom. In the absence of any diffusion, the exact solution is $\phi = 1$ above the diagonal shown in Fig. 11.17a and $\phi = 0$ below it. Figure 11.17b compares the exact profile at $x = 0.5$ with similar ones obtained numerically using the upwind and QUICK schemes.

In comparison with the exact solution, the profile generated by the upwind scheme is smeared and highly inaccurate but very smooth. This inaccuracy is due to a new type of error known as cross-stream diffusion, which is caused by the one-dimensional interpolation profile used, i.e., it is due to treating the flow as locally one dimensional. The origin of cross-flow diffusion was identified by Patankar [4] and Stubley [13] as being a ***multi-dimensional*** phenomenon. It occurs only when the velocity field is not aligned with the grid. An approximate expression for the cross flow diffusion has been given by de Vahl Davis and Mallinson [14] for two dimensional flows as

$$\Gamma^{\phi}_{false} = \frac{\rho|\mathbf{v}|\Delta x \Delta y \sin(2\theta)}{4\left(\Delta y \sin^3(\theta) + \Delta x \cos^3(\theta)\right)} \tag{11.129}$$

where $|\mathbf{v}|$ is the velocity magnitude and θ the angle made by the velocity vector with the x coordinate axis. This error can be reduced by using higher order interpolation schemes as demonstrated by the profile generated with the QUICK scheme. The QUICK scheme profile is shown to be much sharper and more accurate than the upwind profile however it is infected with over/undershoots near the sharp gradient. As mentioned earlier this error is called the dispersion error, which causes the generation of maxima/minima in the solution domain and is a characteristic of all High Order (HO) schemes.

11.6.1 Error Sources

Based on the discussions in the previous sections the sources of numerical error in the discretization of the convective flux can be divided into numerical diffusion and numerical dispersion.

Numerical diffusion, which causes smearing of sharp gradients (Fig. 11.18a) can also be divided into stream wise and cross stream numerical diffusion. Stream wise numerical diffusion can be reduced by increasing the order of the interpolation profile, as shown in Fig. 11.18b, resulting in sharper profiles but inducing under/overshoots in the presence of large gradients. Cross-stream numerical diffusion, shown in Fig. 11.17b, is caused by the one dimensional nature of the assumed profile and can be reduced either by interpolating in the direction of the flow (i.e., multi-dimensional profiles) [15, 16] or by using one-dimensional higher order interpolation profiles (Fig. 11.18b).

Numerical dispersion error manifests itself through oscillation in the generated profile in the presence of large gradients rendering the solution unbounded. As shown in Figs. 11.18b and 11.19 and the results reported in Fig. 11.14b, it exists with all assumed interpolation profiles except the upwind scheme. It is the result of the unphysical behavior of the assumed interpolation profile.

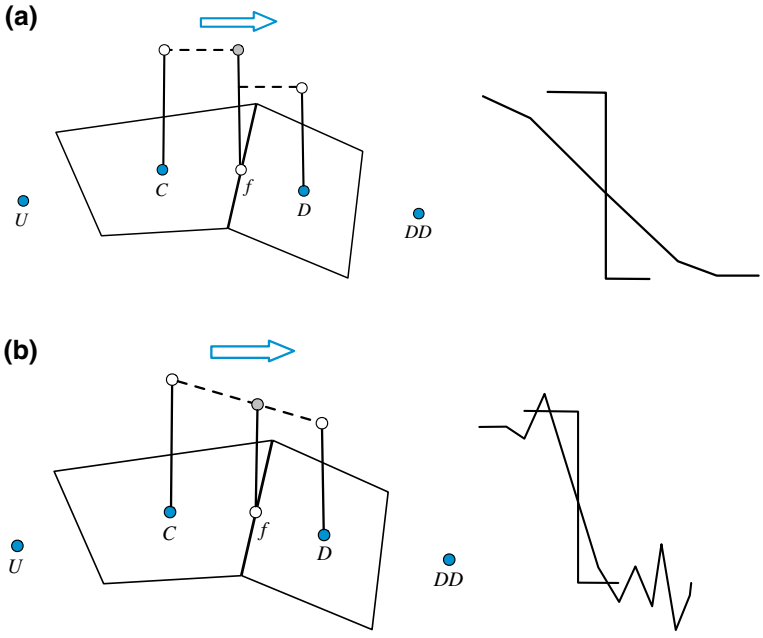


Fig. 11.18 **a** Smearing of sharp profiles by numerical diffusion and **b** wiggles in the computed profile due to numerical dispersion

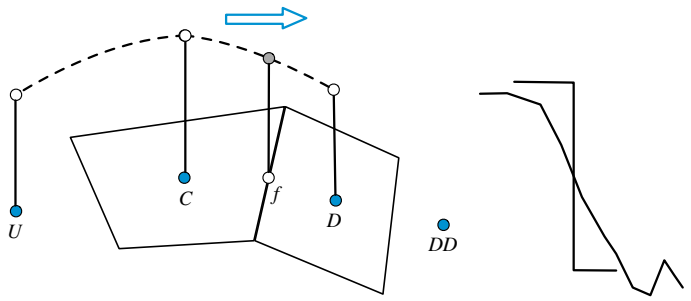


Fig. 11.19 Numerical dispersion error causing oscillations in the presence of a large gradient

An evaluation of this error can be obtained by using a simplified version of Eq. (11.73) in which diffusion and sources are neglecting. If further, velocity and density fields are considered constants, Eq. (11.73) simplifies to

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0 \tag{11.130}$$

Assuming an exact solution of the form

$$\phi(x,t) = \phi(t)e^{jkx} \tag{11.131}$$

where j is the imaginary number defined by $j^2 = -1$. Then the exact value of the gradient becomes

$$\frac{\partial}{\partial x}\phi(x,t) = jk\phi(t)e^{jkx} = jk\phi(x,t). \tag{11.132}$$

With an interpolation profile, the numerical approximation of the gradient is written in terms of ϕ values at locations $-M\Delta x$, $(-M+1)\Delta x$, …, Δx, $2\Delta x$, …, $N\Delta x$ as

$$\frac{\partial}{\partial x}\phi(x,t) \approx \frac{1}{\Delta x}\sum_{n=-M}^{N} a_n \phi(x+n\Delta x,\, t) = \frac{1}{\Delta x}\sum_{n=-M}^{N} a_n \phi(t)\mathrm{e}^{jk(x+n\Delta x)} \tag{11.133}$$

which, upon substitution of the assumed solution, becomes

$$\frac{\partial}{\partial x}\phi(x,t) \approx \frac{1}{\Delta x}\sum_{n=-M}^{N} a_n \phi(t)e^{jk(x+n\Delta x)} = \frac{1}{\Delta x}\sum_{n=-M}^{N} a_n \mathrm{e}^{jkn\Delta x}\phi(x,t). \tag{11.134}$$

By comparing the exact and numerical solutions, it is found that

$$k = \frac{-j}{\Delta x}\sum_{n=-M}^{N} a_n \mathrm{e}^{jkn\Delta x}. \tag{11.135}$$

In general k is an imaginary number that can be written as

$$k = \mathrm{Re}(k) + j\,\mathrm{Im}(k) \tag{11.136}$$

where Re and Im refer to the real and imaginary part, respectively. Inserting k in the exact solution, the approximate solution is obtained as

$$\begin{aligned}\phi(x,t) &= \phi(t)e^{jkx} \\ &\approx \phi(t)e^{j[\mathrm{Re(k)+jIm(k)}]x} = \phi(t)\ \underbrace{\mathrm{e}^{j\mathrm{Re}(k)x}}_{\substack{\textit{Phase}\\ \textit{Dispersion}}}\ \underbrace{\mathrm{e}^{-\mathrm{Im}(k)x}}_{\substack{\textit{Amplitude}\\ \textit{Dissipation}}}\end{aligned} \tag{11.137}$$

Therefore, the numerical solution may include both diffusion (or dissipative) and dispersion errors. If k is real, only dispersion error occurs. However if k is complex, then both types of errors will arise. Based on this analysis, the value of k for the upwind and CD schemes can be checked. For the upwind scheme, the gradient is computed as

$$\begin{aligned}\frac{\partial\phi}{\partial x} &\simeq \frac{\phi_e - \phi_w}{\Delta x} = \frac{\phi_C - \phi_W}{\Delta x} = \frac{e^{jkx} - e^{jk(x-\delta x)}}{\Delta x}\phi(x,t) \\ &= \frac{1-\cos(k\delta x) + j\sin(k\delta x)}{\Delta x}\phi(x,t) \\ &= jk\phi(x,t) \Rightarrow k = \frac{\sin(k\delta x)}{\Delta x} - j\frac{1-\cos(k\delta x)}{\Delta x}\end{aligned} \tag{11.138}$$

It is clear that the upwind scheme gives rise to both types of errors. For the central difference scheme, the gradient is given by

$$\begin{aligned}\frac{\partial\phi}{\partial x} &\simeq \frac{\phi_e - \phi_w}{\Delta x} = \frac{\phi_E - \phi_W}{2\Delta x} = \frac{e^{jk(x+\delta x)} - e^{jk(x-\delta x)}}{2\Delta x}\phi(x,t) \\ &= \frac{1}{\Delta x} j\,\sin(k\delta x)\phi(x,t) = jk\phi(x,t) \Rightarrow k = \frac{\sin(k\delta x)}{\Delta x}\end{aligned} \tag{11.139}$$

Since k is imaginary, then only numerical dispersion error arises. This dispersion error causes oscillations and under/overshoots in the solution.

Having developed a better understanding of numerical dispersion, it is desirable to develop convective non-oscillatory schemes of high order of accuracy. This has kept researchers busy for an extended period of time until the bounding of the convective flux became understood. The development of such schemes will be detailed in the next chapter.

11.7 High Order Schemes on Unstructured Grids

As for structured grids, the functional relationships of HO schemes are defined as functions of the values at the U, C, and D nodes. While the C and D nodes are readily available for any interior face (Fig. 11.20a), defining the U node is not

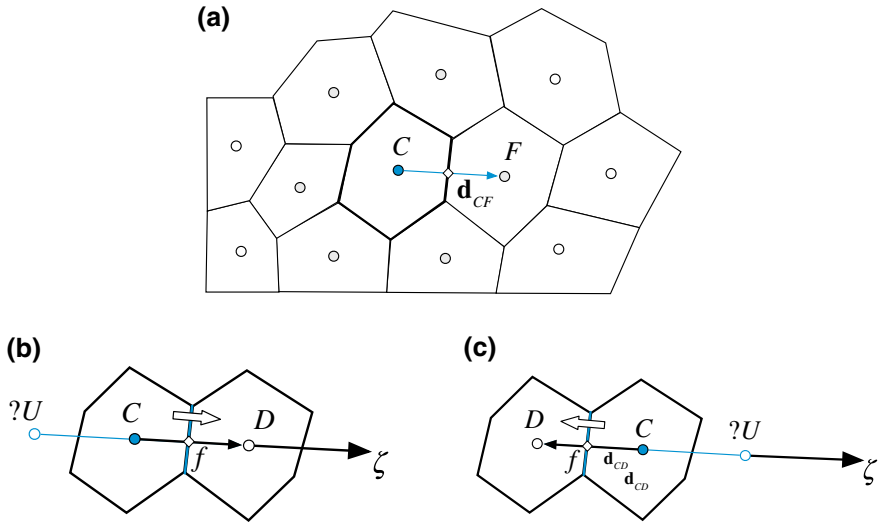


Fig. 11.20 **a** An element in an unstructured grid; the upwind node for HO and HR convection schemes in unstructured grids when the velocity at the element face is **b** negative or **c** positive

straightforward (Fig. 11.20b, c) in an unstructured grid. A simple way around this difficulty is to simply redefine HO schemes in terms of the gradients at the C and D nodes, or a combination of thereof. Another approach is to reconstruct a pseudo U node, which will be detailed in the next chapter and used in developing HR schemes.

11.7.1 Reformulating HO Schemes in Terms of Gradients

This approach relies on profiles developed over structured grids and is best explained by rewriting the QUICK [2] scheme using the new terminology and then generalizing results for any second order profile developed using three points. The functional relationship of the QUICK [2] scheme can be written as

$$\phi_f = \phi_C + \frac{1}{4}\left(\frac{\phi_D - \phi_U}{2}\right) + \frac{1}{4}(\phi_D - \phi_C) \tag{11.140}$$

By approximating the gradients at the locations C and f in the $\mathbf{d}_{CF}$ or ζ direction as shown in Fig. 11.19, then the gradients at the centroids C and f are computed as

$$\frac{\partial \phi_C}{\partial \zeta} = \frac{\phi_D - \phi_U}{2\Delta\zeta} \quad \frac{\partial \phi_f}{\partial \zeta} = \frac{\phi_D - \phi_C}{\Delta\zeta} \tag{11.141}$$

Using Eq. (11.141), Eq. (11.140) can be recast into

$$\phi_f = \phi_C + \frac{1}{2}\left(\frac{\phi_D - \phi_C}{2\Delta\zeta}\right)\frac{\Delta\zeta}{2} + \frac{1}{2}\left(\frac{\phi_D - \phi_C}{\Delta\zeta}\right)\frac{\Delta\zeta}{2} \tag{11.142}$$

or

$$\phi_f = \phi_C + \frac{1}{2}\frac{\partial\phi_C}{\partial\zeta}\frac{\Delta\zeta}{2} + \frac{1}{2}\frac{\partial\phi_f}{\partial\zeta}\frac{\Delta\zeta}{2} \tag{11.143}$$

Denoting the vector between C and f by $\mathbf{d}_{Cf}$, in vector form the above equation becomes

$$\phi_f = \phi_C + \frac{1}{2}\nabla\phi_C \cdot \mathbf{d}_{Cf} + \frac{1}{2}\nabla\phi_f \cdot \mathbf{d}_{Cf} \tag{11.144}$$

which is quite suitable for use in the context of unstructured grids since it only requires information related to gradient at the C and f locations. As long as the computation of these gradients is second order accurate, the way they are calculated becomes immaterial. This gives higher flexibility, as compared to the original formulation, over unstructured grids. Furthermore, Eq. (11.144) suggests that a profile based on three points can be written as

$$\phi_f = a\phi_C + b\nabla\phi_C \cdot \mathbf{d}_{Cf} + c\nabla\phi_f \cdot \mathbf{d}_{Cf} \tag{11.145}$$

where the constants a, b, and c are determined by equating ϕ_f to the profile obtained over structured grids. The general discretized form of Eq. (11.145) can be found once and then used in all subsequent derivations. This is derived by first substituting the approximate values of the gradients using Eq. (11.141) to yield

$$\begin{aligned}\phi_f &= a\phi_C + b\nabla\phi_C \cdot \mathbf{d}_{Cf} + c\nabla\phi_f \cdot \mathbf{d}_{Cf} \\ &= a\phi_C + b\frac{\phi_D - \phi_U}{2\Delta\zeta}\frac{\Delta\zeta}{2} + c\frac{\phi_D - \phi_C}{2\Delta\zeta}\frac{\Delta\zeta}{2}\end{aligned} \tag{11.146}$$

and then after some algebraic manipulations the final form is obtained as

$$\phi_f = \left(a - \frac{c}{2}\right)\phi_C + \left(\frac{b}{4} + \frac{c}{4}\right)\phi_D - \frac{b}{4}\phi_U \tag{11.147}$$

Using the above equation, the calculation of the a, b, and c coefficients is easily accomplished. For example the SOU profile in terms of the gradients can be found as follows:

$$\left.\begin{aligned}\phi_f &= \left(a-\frac{c}{2}\right)\phi_C + \left(\frac{b}{4}+\frac{c}{2}\right)\phi_D - \frac{b}{4}\phi_U \\ &= \frac{3}{2}\phi_C - \frac{1}{2}\phi_U\end{aligned}\right\} \Rightarrow \begin{cases}\dfrac{b}{4}=\dfrac{1}{2} \Rightarrow b=2 \\ \dfrac{b}{4}+\dfrac{c}{2}=0 \Rightarrow c=-1 \\ a-\dfrac{c}{2}=\dfrac{3}{2} \Rightarrow a=1\end{cases} \tag{11.148}$$

Thus the equivalent form of the SOU scheme is given by

$$\phi_f = \phi_C + \left(2\nabla\phi_C - \nabla\phi_f\right)\cdot \mathbf{d}_{Cf} \tag{11.149}$$

Following the same procedure, the gradient forms of the schemes presented earlier are found to be

Upwind scheme : $$\phi_f = \phi_C \tag{11.150}$$

Central difference : $$\phi_f = \phi_C + \nabla\phi_f \cdot \mathbf{d}_{Cf} \tag{11.151}$$

SOU scheme : $$\phi_f = \phi_C + \left(2\nabla\phi_C - \nabla\phi_f\right)\cdot \mathbf{d}_{Cf} \tag{11.152}$$

FROMM scheme : $$\phi_f = \phi_C + \nabla\phi_C \cdot \mathbf{d}_{Cf} \tag{11.153}$$

QUICK scheme : $$\phi_f = \phi_C + \frac{1}{2}\left(\nabla\phi_C + \nabla\phi_f\right)\cdot \mathbf{d}_{Cf} \tag{11.154}$$

Downwind scheme : $$\phi_f = \phi_C + 2\nabla\phi_f \cdot \mathbf{d}_{Cf} \tag{11.155}$$

A full chapter was devoted to the calculation of the gradient at the element centroid and faces and the reader is referred to Chap. 9 for the calculation of $\nabla\phi_C$ and $\nabla\phi_f$.

11.8 The Deferred Correction Approach

The Deferred Correction (DC) procedure of Khosla and Rubin [17] is a compacting technique that enables the use of HO schemes in codes initially written for low order schemes without violating any of the stability rules. The approach is applicable on any type of structured or unstructured grid systems. The method is based on writing the convection flux at a cell face f calculated using a HO scheme as

$$\dot{m}_f \phi_f^{HO} = \underbrace{\dot{m}_f \phi_f^U}_{implicit} + \underbrace{\dot{m}_f\left(\phi_f^{HO} - \phi_f^U\right)}_{explicit} \tag{11.156}$$

where the superscripts U and HO refer to the Upwind and High Order scheme, respectively. By expressing the convection flux in this way, the first term on the right hand side (RHS) is implicitly evaluated by expressing it in terms of nodal

values, while the second term on the RHS is evaluated explicitly using the latest available ϕ values, i.e., values from the previous iteration in an iterative solution procedure. In terms of nodal values, Eq. (11.156) is expressed as

$$\begin{aligned}\dot{m}_f\phi_f^{HO} &= \| \dot{m}_f, 0 \| \phi_C - \| -\dot{m}_f, 0 \| \phi_F + \left(\dot{m}_f\phi_f^{HR} - \| \dot{m}_f, 0 \| \phi_C + \| -\dot{m}_f, 0 \| \phi_F\right) \\ &= \underbrace{FluxC_f\phi_C + FluxF_f\phi_F}_{implicit} + \underbrace{FluxV_f}_{explicit}\end{aligned} \tag{11.157}$$

where

$$\begin{aligned}FluxC_f &= \| \dot{m}_f, 0 \| \\ FluxF_f &= - \| -\dot{m}_f, 0 \| \\ FluxV_f &= \dot{m}_f\phi_f^{HR} - FluxC_f\phi_C - FluxF_f\phi_F\end{aligned} \tag{11.158}$$

Substituting the convection flux given by Eqs. (11.157) and (11.158) in Eq. (11.156) and rearranging, the final form of the algebraic equation is obtained as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{11.159}$$

where

$$\begin{aligned}a_F &= FluxF_f = - \| -\dot{m}_f, 0 \| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \sum_{f\sim nb(C)} \| \dot{m}_f, 0 \| = \sum_{f\sim nb(C)} \left(\dot{m}_f + \| -\dot{m}_f, 0 \|\right) \\ &= - \sum_{F\sim NB(C)} a_F + \sum_{f\sim nb(C)} \dot{m}_f\end{aligned} \tag{11.160}$$

and

$$\begin{aligned}b_C &= Q_C^{\phi}V_C - \underbrace{\sum_{f\sim nb(C)} FluxV_f}_{b_C^{DC}} \\ &= Q_C^{\phi}V_C - \underbrace{\sum_{f\sim nb(C)} \dot{m}_f\left(\phi_f^{HO} - \phi_f^{U}\right)}_{b_C^{DC}}\end{aligned} \tag{11.161}$$

the b_C^{DC} represents the extra source term arising due to the DC procedure. Moreover, the DC technique results in an equation for which the coefficient matrix is always diagonally dominant since it is formed using the upwind scheme.

Example 2
Derive the coefficients for a deferred correction implementation of the QUICK scheme

Solution
The conservation equation is discretized into the following algebraic equation in every cell:

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C$$

For the quick scheme the value at a cell face is computed as

$$\phi_f = \frac{3}{4}\phi_C - \frac{1}{8}\phi_U + \frac{3}{8}\phi_D$$

The coefficients of the algebraic system of equations in a deferred correction approach are based on the UPWIND convection scheme. As such these coefficients are given by

$$a_F = -\|-\dot{m}_f, 0\|$$
$$a_C = -\sum_{F\sim NB(C)} a_F + \sum_{f\sim nb(C)} \dot{m}_f$$

The difference between the UPWIND and QUICK schemes is explicitly accounted for in the source term which is modified to

$$b_C = S_C^{\phi}V_C - \sum_{f\sim nb(C)} \dot{m}_f\left(\frac{3}{4}\phi_C - \frac{9}{8}\phi_U + \frac{3}{8}\phi_D\right)$$

This compacting procedure is simple to implement, however as the difference between the cell face values calculated with the upwind scheme and that calculated with the HO scheme becomes larger, the convergence rate diminishes.

11.9 Computational Pointers

11.9.1 uFVM

The assembly of the convection term in uFVM is performed in cfdAssembleConvectionTerm, where the assembly for the upwind scheme is performed first for interior faces (cfdAssembleConvectionTermInterior), then for boundary faces by looping over the various boundary patches (cfdAssembleConvectionTermInletBC, cfdAssembleConvectionTermOutletBC, etc.).

The core of the interior faces assembly routine is shown in Listing 11.1. The owner and neighbor indices for the interior faces are first retrieved and an upwind index is defined (pos). Then the coefficients for the upwind scheme are computed.

```
theMdotName = ['Mdot' theFluidTag];
mdotField = cfdGetMeshField(theMdotName,'Faces');

mdot_f = mdotField.phi(iFaces);

iOwners = [theMesh.faces(iFaces).iOwner];
iNeighbours = [theMesh.faces(iFaces).iNeighbour];
pos = zeros(size(mdot_f));
pos((mdot_f>0))=1;

theFluxes.FLUXC1f(iFaces,1) = mdot_f.*pos;
theFluxes.FLUXC2f(iFaces,1) = mdot_f.*(1-pos);
theFluxes.FLUXVf(iFaces,1) = 0;
```

Listing 11.1 Convection scheme class declaration

The implementation of High Order schemes is performed after the upwind assembly using the deferred correction method. The assembly of the QUICK scheme (cfdAssembleConvectionTermDCQUICK) is shown in Listing 11.2.

```
iUpwind = pos.*iOwners + (1-pos).*iNeighbours;
%get the upwind gradient at the interior faces
phiGradCf = phiGrad(iUpwind,:,iComponent);
%interpolated gradient to interior faces
iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';
pos = zeros(size(mdot_f));
pos(mdot_f>0)=1;
%
phiGradf                                                                    =
cfdInterpolateGradientsFromElementsToInteriorFaces('Average:Corrected'
,phiGrad,phi);
rc = [theMesh.elements(iUpwind).centroid]';
rf = [theMesh.faces(iFaces).centroid]';
rCf = rf-rc;
%
corr = mdot_f .* dot(phiGradCf'+phiGradf',rCf')'*0.5;
%
theFluxes.FLUXTf(iFaces) = theFluxes.FLUXTf(iFaces) +  corr;
```

Listing 11.2 Assembly of the QUICK scheme

The deferred correction value is computed for all interior faces as

$$\dot{m}_f\left(\phi_f^{HR}-\phi_f^{UPWIND}\right)=\dot{m}_f\left(\underbrace{\phi_C+\frac{1}{2}\left(\nabla\phi_C+\nabla\phi_f\right)\cdot\mathbf{d}_{Cf}}_{QUICK}-\underbrace{\phi_C}_{UPWIND}\right)$$
$$=\dot{m}_f\frac{1}{2}\left(\nabla\phi_C+\nabla\phi_f\right)\cdot\mathbf{d}_{Cf} \tag{11.162}$$

11.9.2 OpenFOAM®

In OpenFOAM® [18] the convection term can be evaluated either explicitly using **fvc::div(mDot, phi)** or implicitly via the **fvm::div(mDot,phi)** function. The **fvc::div(mDot,phi)** returns a field in which the divergence of phi is evaluated in each cell. The field is added to the right hand side of the system of equations. The **fvm::div(mDot,phi)** returns the **fvMatrix**, a matrix of coefficients that are evaluated based on the linearization of the face fluxes. The matrix of coefficients is then added to the left hand side of the system of equations.

The scripts of **fvm::div** and **fvc::div** functions can be found in the file **FOAM_SRC/finiteVolume/finiteVolume/convectionSchemes/gaussConvectionScheme/gaussConvectionSheme.C.** As already stated, the convection class is based on the type of interpolation scheme at the face and accordingly the declaration of the class is performed as displayed in Listing 11.3.

```
template<class Type>
class gaussConvectionScheme
:
    public fv::convectionScheme<Type>
{
    // Private data

        tmp<surfaceInterpolationScheme<Type> > tinterpScheme_;
```

Listing 11.3 Convection scheme class declaration

In the above declaration the private member **tinterpScheme_** describes the face interpolation type on which the divergence operator is based. Additionally to the above mentioned function, the **gaussConvectionScheme** class defines two auxiliary functions named **interpolate** and **flux**, as shown in Listing 11.4, which wrap the **surfaceInterpolation** class used to perform interpolation from volume fields to surface fields.

```
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
gaussConvectionScheme<Type>::interpolate
(
    const surfaceScalarField&,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    return tinterpScheme_().interpolate(vf);
}

template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
gaussConvectionScheme<Type>::flux
(
    const surfaceScalarField& faceFlux,

    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    return faceFlux*interpolate(faceFlux, vf);
}
```

Listing 11.4 The **gaussConvectionScheme** class that defines the **interpolate** and **flux** functions

For the explicit discretization of convection, the operator defines a field where the divergence of the cell face fluxes are stored. The code used by the **fvc::div** operator for the explicit evaluation of the divergence of a field over a specific volume is as follows (Listing 11.5):

```
template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
gaussConvectionScheme<Type>::fvcDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<GeometricField<Type, fvPatchField, volMesh> > tConvection
    (
        fvc::surfaceIntegrate(flux(faceFlux, vf))
    );

    tConvection().rename
    (
        "convection(" + faceFlux.name() + ',' + vf.name() + ')'
    );

    return tConvection;
}
```

Listing 11.5 Script used for the explicit evaluation of the divergence operator

The calculation of the divergence of a field involves the following three steps:

1. Evaluating values at the faces of the element.
2. Multiplying the value at the face with the mass flux at the face (i.e., **faceFlux**).
3. Summing the contributions of all cell faces and dividing the sum by the cell volume.

First the face value of the generic variable **vf** is evaluated. Then the estimate obtained is multiplied by the corresponding face mass flux value using auxiliary functions. Finally the sum over the faces of the cell is performed using the **surfaceIntegrate** function. The implementation of this function can be found under **FOAM_SRC/finiteVolume/finiteVolume/fvc/fvcSurfaceIntegrate.C** and its script is given in Listing 11.6.

```
template<class Type>
void surfaceIntegrate
(
    Field<Type>& ivf,
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf
)
{
    const fvMesh& mesh = ssf.mesh();

    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();

    const Field<Type>& issf = ssf;

    forAll(owner, facei)
    {
        ivf[owner[facei]] += issf[facei];
        ivf[neighbour[facei]] -= issf[facei];
    }

    forAll(mesh.boundary(), patchi)
    {
        const labelUList& pFaceCells =
            mesh.boundary()[patchi].faceCells();

        const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];

        forAll(mesh.boundary()[patchi], facei)
        {
            ivf[pFaceCells[facei]] += pssf[facei];
        }
    }

    ivf /= mesh.V();
}
```

Listing 11.6 Script of the **surfaceIntegrate** function

The implicit discretization of the convection term is performed using the **fvm::div** operator. It does so by implicitly expressing the dependent variable value at the face as function of the owner and neighbor values according to

$$\phi_f = \underbrace{\varpi}_{\substack{owner \\ coefficient}} \phi_O + \underbrace{(1-\varpi)}_{\substack{neighbour \\ coefficient}} \phi_N \tag{11.163}$$

where subscripts O and N refer to owner and neighbor, respectively, and ϖ represents the weight assigned to the contribution of the owner to the value at the face. The way the weight is calculated will be described in the next chapter. The coefficients of ϕ_O and ϕ_N are then used to assemble the matrix of coefficients. The script used to perform implicit discretization of the divergence operator reads (Listing 11.7)

```
template<class Type>
tmp<fvMatrix<Type> >
gaussConvectionScheme<Type>::fvmDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<surfaceScalarField> tweights = tinterpScheme_().weights(vf);
    const surfaceScalarField& weights = tweights();

    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            faceFlux.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm();
…
```

Listing 11.7 Implicit calculation of the divergence of a field

where in a first place an **fvMatrix** is defined and then, as shown in Listing 11.8, it is properly filled with the corresponding coefficients as

```
    fvm.lower() = -weights.internalField()*faceFlux.internalField();
    fvm.upper() = fvm.lower() + faceFlux.internalField();
    fvm.negSumDiag();
```

Listing 11.8 Properly assembling the weights contributions to coefficients

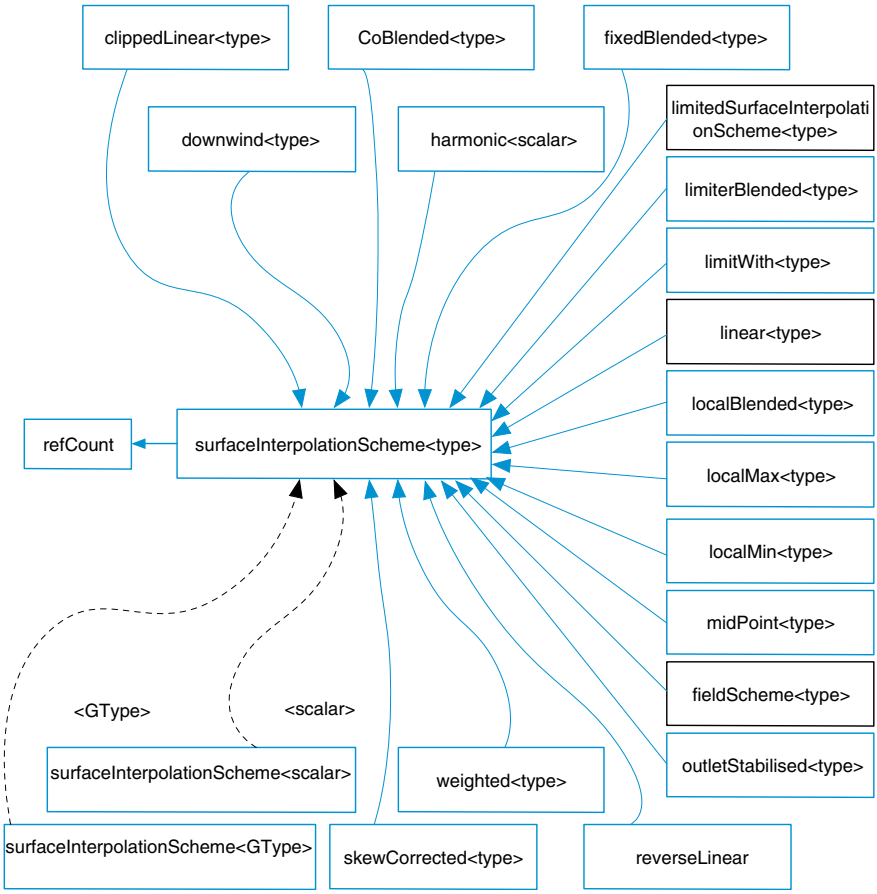


Fig. 11.21 **UML** Graph for the surfaceInterpolationScheme class, where a box with a black border denotes a documented struct or class for which not all inheritance/containment relations are shown

where the upper and lower vectors are now defined using the interpolation weights denoted **tweights**, as described above. Further, Eq. (11.163) indicates that the implementation of the implicit operator in OpenFOAM® is based on the so called "downwind" discretization (as explained later) in which all schemes are discretized in a fully implicit manner independently of the order of accuracy and without using a deferred correction approach. This technique is also similar to the downwind weighing factor method that will be described in the next chapter.

As described above for both explicit and implicit discretization methods, the divergence operator for the calculation of the convection term is based on the face interpolation and the calculation of the weights. OpenFOAM®performs these tasks in a base class denoted by **surfaceInterpolationScheme**. From this base class a large number of interpolation schemes are derived and implemented, as shown in the **UML** graph depicted in Fig. 11.21. For better understanding, additional details about this class are given next.

As mentioned above, the **fvc::div** and **fvm::div** operators use the **surfaceInterpolationScheme** class to perform the needed tasks for each discretization method. In this class, the functions used to calculate the values at the faces (vf) and the weights are displayed in Listings 11.9 and 11.10, respectively, and where **tinterpScheme_** (Listing 11.10) is a **surfaceInterpolationScheme** object.

```
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
gaussConvectionScheme<Type>::interpolate
(
    const surfaceScalarField&,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const

{
    return tinterpScheme_().interpolate(vf);
}
```

Listing 11.9 The interpolation function where the value at the face (vf) is computed

and

```
template<class Type>
tmp<fvMatrix<Type> >
gaussConvectionScheme<Type>::fvmDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<surfaceScalarField> tweights = tinterpScheme_().weights(vf);
    const surfaceScalarField& weights = tweights();
```

Listing 11.10 The function used to calculate the weights

The definition of the class can be found in **FOAM_SRC/finiteVolume/interpolation/surfaceInterpolation/surfaceInterpolationScheme/SurfaceInterpolationScheme.H** where the two main functions (Listings 11.11 and 11.12), necessary for the calculation of the divergence operator are defined.

```
//- Return the face-interpolate of the given cell field
//  with explicit correction
virtual tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
interpolate(const GeometricField<Type, fvPatchField, volMesh>&) const;
```

Listing 11.11 Definition of the main function for face interpolation

```
//- Return the interpolation weighting factors for the given field
virtual tmp<surfaceScalarField> weights
(
    const GeometricField<Type, fvPatchField, volMesh>&
) const = 0;
```

Listing 11.12 Definition of the main function for weight factors calculation

The **interpolate** function in Listing 11.9 (here OpenFOAM® adopts the same function name but for a different implementation) is used with the explicit operator **fvc::div** and is defined as a normal virtual function meaning that a derived class may or may not adopt it. For most of the derived classes this function is not selected and the definition in the **surfaceInterpolationScheme** class is based on a modified form of Eq. (11.163) written as

$$\phi_f = \underbrace{\varpi}_{\substack{owner\\coefficient}} \phi_O + \underbrace{(1-\varpi)}_{\substack{neighbour\\coefficient}} \phi_N = \phi_N + \varpi(\phi_O - \phi_N) \tag{11.164}$$

To implement Eq. (11.164), OpenFOAM® uses auxiliary functions with names similar to the ones used earlier in the **surfaceInterpolationScheme** class. First an **interpolate** class with a single argument is instantiated from the divergence operator. Then inside the **interpolate** function a new **interpolate** function with two arguments is introduced along with a new **weights** function, using the script shown in Listing 11.13.

```
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
surfaceInterpolationScheme<Type>::interpolate
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
…

    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tsf
        = interpolate(vf, weights(vf));
…
```

Listing 11.13 The new interpolate function with two arguments

The new **interpolate** function (the one with the two arguments) compute the face value according to Eq. (11.164). The script used to perform this task is given by (Listing 11.14)

```
//- Return the face-interpolate of the given cell field
//  with the given weighting factors
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
surfaceInterpolationScheme<Type>::interpolate
(
    const GeometricField<Type, fvPatchField, volMesh>& vf,
    const tmp<surfaceScalarField>& tlambdas
)
{
…
    const surfaceScalarField& lambdas = tlambdas();

    const Field<Type>& vfi = vf.internalField();
    const scalarField& lambda = lambdas.internalField();

    const fvMesh& mesh = vf.mesh();
    const labelUList& P = mesh.owner();
    const labelUList& N = mesh.neighbour();

   GeometricField<Type, fvsPatchField, surfaceMesh>& sf = tsf();

    Field<Type>& sfi = sf.internalField();


    for (register label fi=0; fi<P.size(); fi++)
    {
        sfi[fi] = lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]];
    }
…
```

Listing 11.14 Script used to compute face values according to Eq. (11.164)

where lambda represents the weight ϖ in Eq. (11.164).

As briefly introduced above, the **weights** function is now a pure virtual function and it defines the weights of the interpolation. Contrary to the **interpolate** function, the **weights** function being pure virtual has to be redefined for every derived class (C^{++} requirement). Looking again at Fig. 11.20, all derived classes have to define the **weights** function. The redefinition must be performed by constructing the proper weights corresponding to the scheme adopted.

For the case of the central difference scheme the **weights** function is defined in a class named **linear<Type>** with its script given by (Listing 11.15),

```
//Member Functions

//- Return the interpolation weighting factors
virtual tmp<surfaceScalarField> weights
(
   const GeometricField<Type, fvPatchField, volMesh>&
) const
{
   return this->mesh().surfaceInterpolation::weights();
}
```

Listing 11.15 Script for calculating the weights of the central difference scheme

where **surfaceInterpolation::weights()** returns the distance weights based on the mesh, thus defining the central difference discretization.

Another example is the *downwind* scheme defined by Eq. 11.44, and for which the **weights** function is given by (Listing 11.16)

```
//- Return the interpolation weighting factors
virtual tmp<surfaceScalarField> weights
(
    const GeometricField<Type, fvPatchField, volMesh>&
) const
{
    return neg(faceFlux_);
}
```

Listing 11.16 Script for calculating the weights of the downwind scheme

where the **neg** function returns 0 for positive fluxes and 1 for negative ones, which is the opposite of the **pos** function used with the upwind scheme.

11.10 Closure

The chapter dealt with the discretization of the convection term. The difficulties associated with the use of a linear symmetrical profile were discussed and remedies suggested. The upwind scheme, although stable and generates physically plausible results, is highly diffusive smearing sharp gradients and producing results that are first order accurate. Upwind-biased HO convection schemes were shown to improve accuracy but to produce a new type of error known as the dispersion error, which manifest itself in the solution through wiggles, oscillations, and over/under shoots across sharp gradients. No attempt was made to address this error in this chapter. Among other issues, this will form the subject of the next chapter that further expands on the discretization of the convection term.

11.11 Exercises

Exercise 1

(a) Find third order accurate expressions, i.e., $O(\Delta x^3)$, of ϕ_U, ϕ_C, and ϕ_D by using a Taylor series expansion about the face f for the one dimensional uniform grid shown in Fig. 11.22.
(b) Derive a high-order scheme using a linear combination $(a\phi_U + b\phi_C + c\phi_D)$ of these expressions in such a way as to eliminate the first and second order derivatives from the final expression for ϕ_f with the additional condition that $a + b + c = 1$.
(c) Prove that the scheme you just derived is third order accurate in its representation of the convection operator.

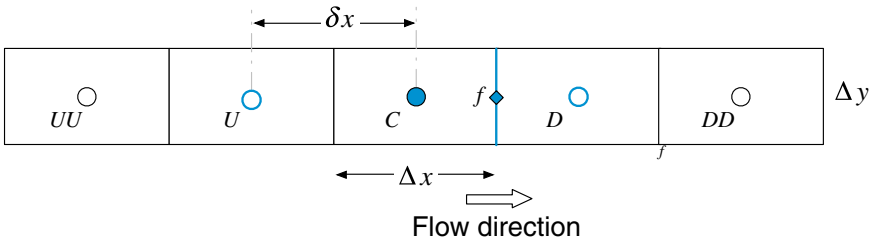


Fig. 11.22 A uniform one dimensional grid system

Exercise 2

The QUICK scheme fits a quadratic function to three nodal values to estimate the value of a scalar at a cell face, according to

$$\phi_f = -\frac{1}{8}\phi_U + \frac{3}{4}\phi_C + \frac{3}{8}\phi_D$$

(a) For a uniform cartesian two dimensional grid write down the expressions for ϕ_e, ϕ_w, ϕ_n and ϕ_s in terms of the values at neighboring nodes, assuming that the velocity components u and v are known, constant, and positive.
(b) Neglecting diffusion and assuming a uniform source Q^ϕ per unit volume, derive an algebraic discretization of the ϕ scalar conservation equation given by

$$\nabla.(\rho \mathbf{v}\phi) = Q^\phi$$

in the form

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C$$

assuming that a cell is of unit depth, with the area of its e, w, n, and s faces denoted by S_e, S_w, S_n, and S_s, respectively, and its volume by V_C.

(c) Splitting the QUICK expression for ϕ_f into the form "Upwind differencing" + "deferred correction" so that the coefficients become similar to those of the upwind scheme, then moving the deferred correction to the source term, write an expression for this source term.

Exercise 3

In a steady two dimensional situation, the variable ϕ is governed by

$$\nabla.(\rho\mathbf{v}\phi) = Q^{\phi}$$

where $\rho = 1$, $Q^{\phi} = 15 - 3\phi$.

The flow field is such that $\mathbf{v} = \mathbf{li} + 4\mathbf{j}$ everywhere and $\Delta x = \Delta y = 1$. The domain is discretized using the orthogonal grid shown in Fig. 11.23 with the values of ϕ given at the inlet boundaries as shown in the figure. Using the finite volume approach and the Second Order Upwind convection scheme to

(a) derive the algebraic equations for the four control volumes, and
(b) compute the values of ϕ_1, ϕ_2, ϕ_3 and ϕ_4 using 3 iterations of a simple Gauss-Siedel type solver.

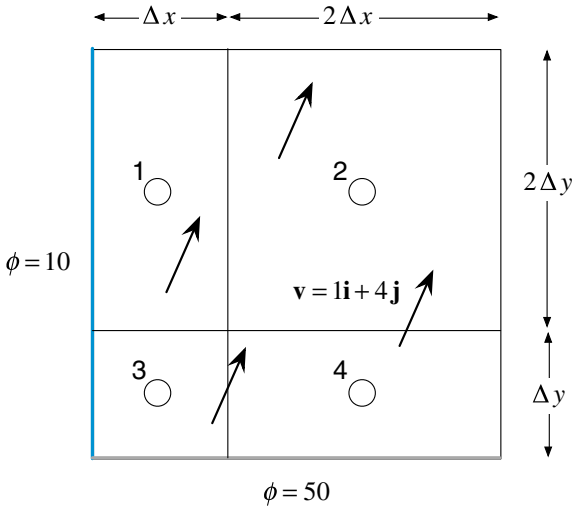


Fig. 11.23 A two dimensional configuration discretized using a non-uniform Cartesian grid for the advection of a scalar ϕ in the presence of a source term

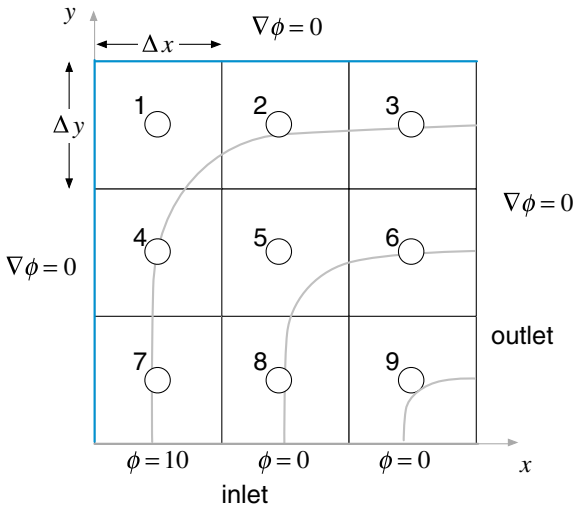


Fig. 11.24 Convection of a two dimensional scalar field

Exercise 4

Consider the steady transport of a scalar ϕ in the domain shown in Fig. 11.24. The governing conservation equation is given by

$$\nabla.(\rho \mathbf{v}\phi) = 0$$

where $\rho = 1$, $\mathbf{v} = 2yx^2\mathbf{i} - 2xy^2\mathbf{j}$, and $\Delta x = \Delta y = 1/3$.

(a) Using the UPWIND scheme, discretize the equation over the computational domain and find the value of ϕ at each element centroid.
(b) Using the QUICK scheme, applied via a deferred correction approach, discretize the equation over the computational domain and find the value of ϕ at each element centroid.

Exercise 5 (OpenFOAM®)

(a) Using Doxygen [19], list all derived classes of the *surfaceInterpolationScheme<Type>* class (these classes implement the virtual *interpolate* function).
(b) Describe the weights function of the class midPoint<Type>, downwind <Type> and linear<Type>.
(c) Write a class that inherits the *surfaceInterpolationScheme<Type>* class and implements the interpolate function using a geometric average.

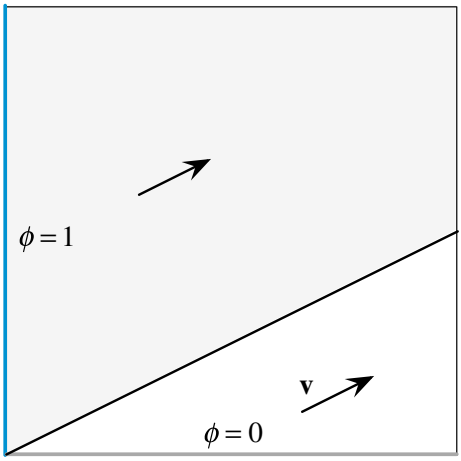


Fig. 11.25 Advection of a step profile in an oblique velocity field

Exercise 6 (OpenFOAM®, uFVM)
The advection of a step profile in an oblique velocity field, **v** = 2**i** + **j**, shown in Fig. 11.25 is governed by

$$\nabla . (\rho \mathbf{v} \phi) = 0$$

For different grid sizes, setup the problem and solve it in OpenFOAM® and uFVM using the following advection schemes assuming unit dimensions in x and y directions, and compare results with the exact solution $(\rho = 1)$:

(a) UPWIND
(b) QUICK
(c) SOU

Exercise 7 (OpenFOAM®, uFVM)
The Smith-Hutton test governed by

$$\nabla . (\rho \mathbf{v} \phi) = 0$$

and illustrated in Fig. 11.26, involves the pure advection of a step profile in a rotational velocity field described as

$$\mathbf{v} = 2y\left(1 - x^2\right)\mathbf{i} - 2x\left(1 - y^2\right)\mathbf{j}$$

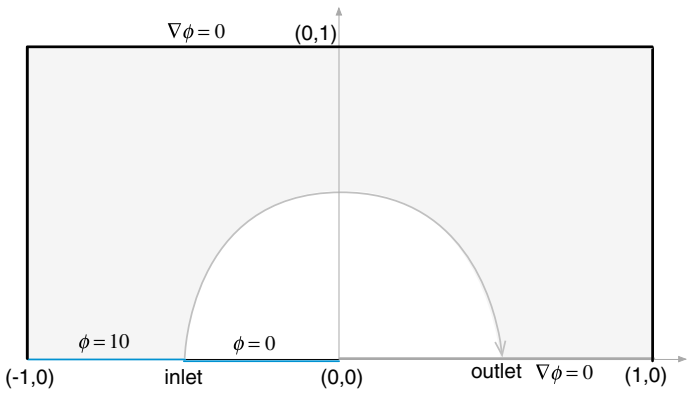


Fig. 11.26 Advection of a step profile in a two dimensional rotational velocity field

For different grid sizes, solve the test in openFOAM® and uFVM using the following advection schemes, and compare results with the exact solution ($\rho = 1$):

(a) UPWIND
(b) QUICK
(c) SOU

References

1. Spalding DB (1972) A novel finite difference formulation for differential expressions involving both first and second derivatives. Int J Numer Meth Eng 4:551–559
2. Leonard BP (1979) A stable and accurate convective modelling procedure based on quadratic upstream interpolation. Comput Methods Appl Mech Eng 19:59–98
3. Raithby GD (1976) A critical evaluation of upstream differencing applied to problems involving fluid flow. Comput Methods Appl Mech Eng 9:75–103
4. Patankar SV (1980) Numerical heat transfer and fluid flow. Hemisphere Publishing Corporation, McGraw-Hill
5. Patankar SV, Baliga BR (1978) A new finite-difference scheme for parabolic differential equations. Numer Heat Transf 1:27–37
6. Courant R, Isaacson E, Rees M (1952) On the solution of nonlinear hyperbolic differential equations by finite differences. Commun Pure Appl Math 5:243–255
7. Moukalled F, Darwish M (2012) Transient schemes for capturing interfaces of free-surface flows. Numer Heat Transf Part B Fundam 61(3):171–203
8. Darwish M, Moukalled F (2006) Convective schemes for capturing interfaces of free-surface flows on unstructured grids. Numer Heat Transf Part B Fundam 49(1):19–42
9. Leonard BP (1979) A survey of finite difference of opinion on numerical muddling of the incomprehensible defective confusion equation. In: Hughes TJR (ed) Finite element methods for convection dominated flows, AMD-34, ASME

10. Shyy W (1985) A study of finite difference approximations to steady state convection dominate flow problems. J Comput Phys 57:415–438
11. Leonard BP, Leschziner MA, McGuirk J (1978) Third order finite-difference method for steady two-dimensional convection. In: Taylor C, Morgan K, Brebbia CA (eds) Numerical methods in laminar and turbulent flows. Pentech Press, London, pp 807–819
12. Fromm JE (1968) A method for reducing dispersion in convective difference schemes. J Comput Phys 3:176–189
13. Stubley GD, Raithby GD, Strong AB (1980) Proposal for a new discrete method based on an assessment of discretization errors. Numerical Heat Transfer 3:411–428
14. de Vahl Davis G, Mallinson GD (1972) False diffusion in numerical fluid mechanics. University of New South Wales, School of Mechanical and Industrial Engineering (Report 1972/FMT/1)
15. Raithby GD (1976) Skew upstream differencing schemes for problems involving fluid flow. Comput Methods Appl Mech Eng 9:153–164
16. Darwish M, Moukalled F (1996) A new route for building bounded skew-upwind schemes. Comput Methods Appl Mech Eng 129:221–233
17. Khosla PK, Rubin SG (1974) A diagonally dominant second-order accurate implicit scheme. Comput Fluids 2:207–209
18. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
19. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 12
High Resolution Schemes

Abstract This chapter continues the development of convection schemes and discusses approaches by which boundedness is enforced on High Order (HO) convection schemes to produce High Resolution (HR) schemes. The recipe for a HR scheme is shown to involve a combination of a HO profile and a Convection Boundedness Criterion (CBC) ensuring that no oscillatory behavior is experienced in the solution. The Normalized Variable Formulation (NVF) and the Total Variation Diminishing (TVD) frameworks for developing HR schemes are introduced. Even though the two approaches look very different they are shown to be almost identical. The Normalized Variable Diagram (NVD) and Sweby's (or $r-\psi$) diagram for visualizing HR schemes in the NVF and TVD formulation, respectively, are presented. The functional relationships for several HR schemes are specified in the context of both the NVF and TVD formulations. In addition to the Deferred Correction (DC) procedure discussed in the previous chapter, two additional techniques for the implementation of HO and HR schemes in structured and unstructured grids are introduced, namely the Downwind Weighing Factor (DWF) method and the Normalized Weighing Factor (NWF) method.

12.1 The Normalized Variable Formulation (NVF)

The Normalized Variable Formulation (NVF) is a framework for the description and analysis of High Resolution (HR) schemes. It was introduced by Leonard [1–3] and gained popularity with the Gaskell and Lau simplified Convection Boundedness Criterion (CBC) [4]. The Normalized Variable Diagram (NVD) is a useful tool for the development and analysis of HO and HR schemes.

The NVF is a face formulation procedure based on locally normalizing the dependent variable for which the value ϕ_f at face f is to be constructed. The approach relies on the upwind (ϕ_C), downwind (ϕ_D), and far upwind (ϕ_U) node values, illustrated in Fig. 12.1, to express the normalized variable as

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_12

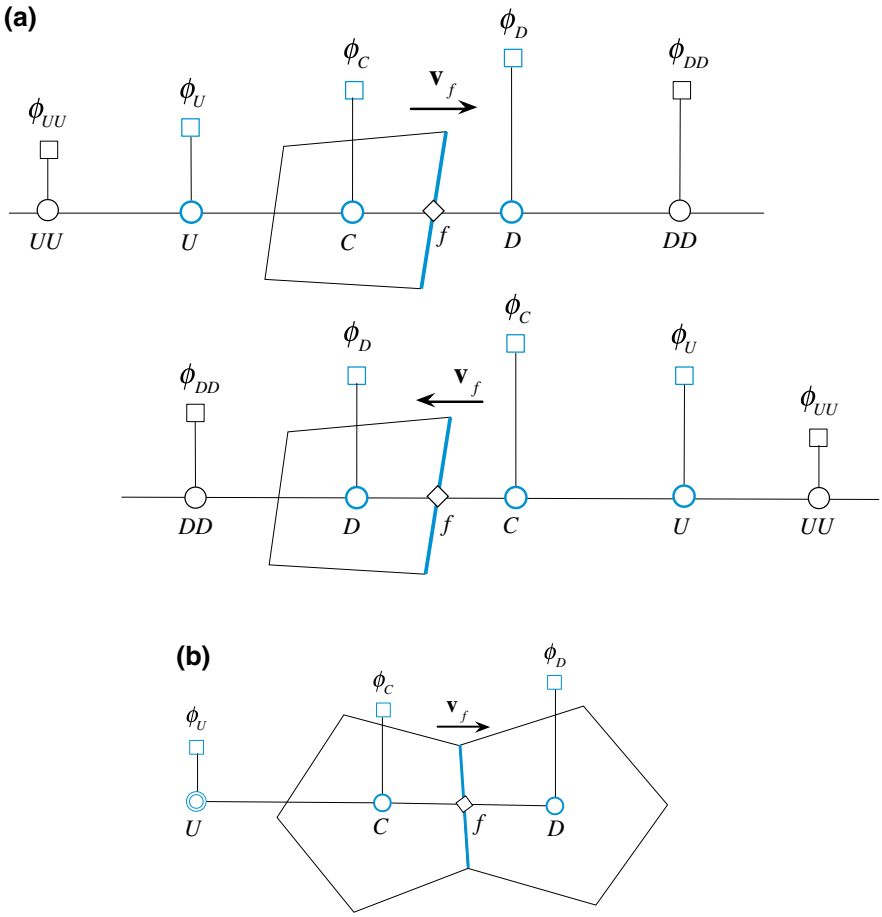


Fig. 12.1 A schematic showing **a** the U, C, and D node locations used in describing convection schemes on structured grids **b** the C, D and extrapolated U nodes for an unstructured grid

$$\tilde{\phi} = \frac{\phi - \phi_U}{\phi_D - \phi_U} \tag{12.1}$$

With this normalization the relation

$$\phi_f = f(\phi_U, \phi_C, \phi_D) \tag{12.2}$$

is transformed to

$$\tilde{\phi}_f = f\left(\tilde{\phi}_C\right) \tag{12.3}$$

since the normalized values of ϕ_D and ϕ_U become equal to

$$\tilde{\phi}_U = 0 \text{ and } \tilde{\phi}_D = 1 \tag{12.4}$$

while the normalized value of $\phi_C\left(\tilde{\phi}_C\right)$ becomes an indicator of smoothness for the ϕ field. Values of $\tilde{\phi}_C$ between 0 and 1 $\left(0<\tilde{\phi}_C<1\right)$, represent a monotonic profile while values of $\tilde{\phi}_C$ that are less than 0 $\left(\tilde{\phi}_C<0\right)$ or greater than 1 $\left(\tilde{\phi}_C>1\right)$ indicate an extremum at C. In addition, values of $\tilde{\phi}_C \approx 0$ or $\tilde{\phi}_C \approx 1$ indicate a gradient jump. These configurations are illustrated in Fig. 12.2.

Normalization is also useful for transforming the functional relationships of HO schemes into linear relations between $\tilde{\phi}_f$ and $\tilde{\phi}_C$. For example, the normalized functional relationships of the HO schemes presented in the previous chapter are as follows:

Upwind: $$\phi_f = \phi_C \quad \Rightarrow \tilde{\phi}_f = \tilde{\phi}_C \tag{12.5}$$

Central difference: $$\phi_f = \frac{1}{2}(\phi_C + \phi_D) \quad \Rightarrow \tilde{\phi}_f = \frac{1}{2}\left(1 + \tilde{\phi}_C\right) \tag{12.6}$$

Second order upwind: $$\phi_f = \frac{3}{2}\phi_C - \frac{1}{2}\phi_U \quad \Rightarrow \tilde{\phi}_f = \frac{3}{2}\tilde{\phi}_C \tag{12.7}$$

FROMM: $$\phi_f = \phi_C + \frac{\phi_D - \phi_U}{4} \quad \Rightarrow \tilde{\phi}_f = \tilde{\phi}_C + \frac{1}{4} \tag{12.8}$$

QUICK: $$\phi_f = \frac{3}{8}\phi_D + \frac{3}{4}\phi_C - \frac{1}{8}\phi_U \quad \Rightarrow \tilde{\phi}_f = \frac{3}{8} + \frac{3}{4}\tilde{\phi}_C \tag{12.9}$$

Downwind: $$\phi_f = \phi_D \quad \Rightarrow \tilde{\phi}_f = 1 \tag{12.10}$$

Thus, for all HO schemes that are based on three nodal values, $\tilde{\phi}_f$ can always be expressed as a linear function of $\tilde{\phi}_C$, i.e., $\tilde{\phi}_f = \ell\tilde{\phi}_C + k$, where the values of ℓ and k depend on the scheme. Therefore, if $\tilde{\phi}_f$ is plotted as a function of $\tilde{\phi}_C$ in the $\left(\tilde{\phi}_C, \tilde{\phi}_f\right)$ plane, then the functional relationships of these schemes will appear as straight lines on the plot. The resultant plot is denoted by the Normalized Variable Diagram (NVD). An NVD on which the functional relationships of the above schemes are plotted is displayed in Fig. 12.3.

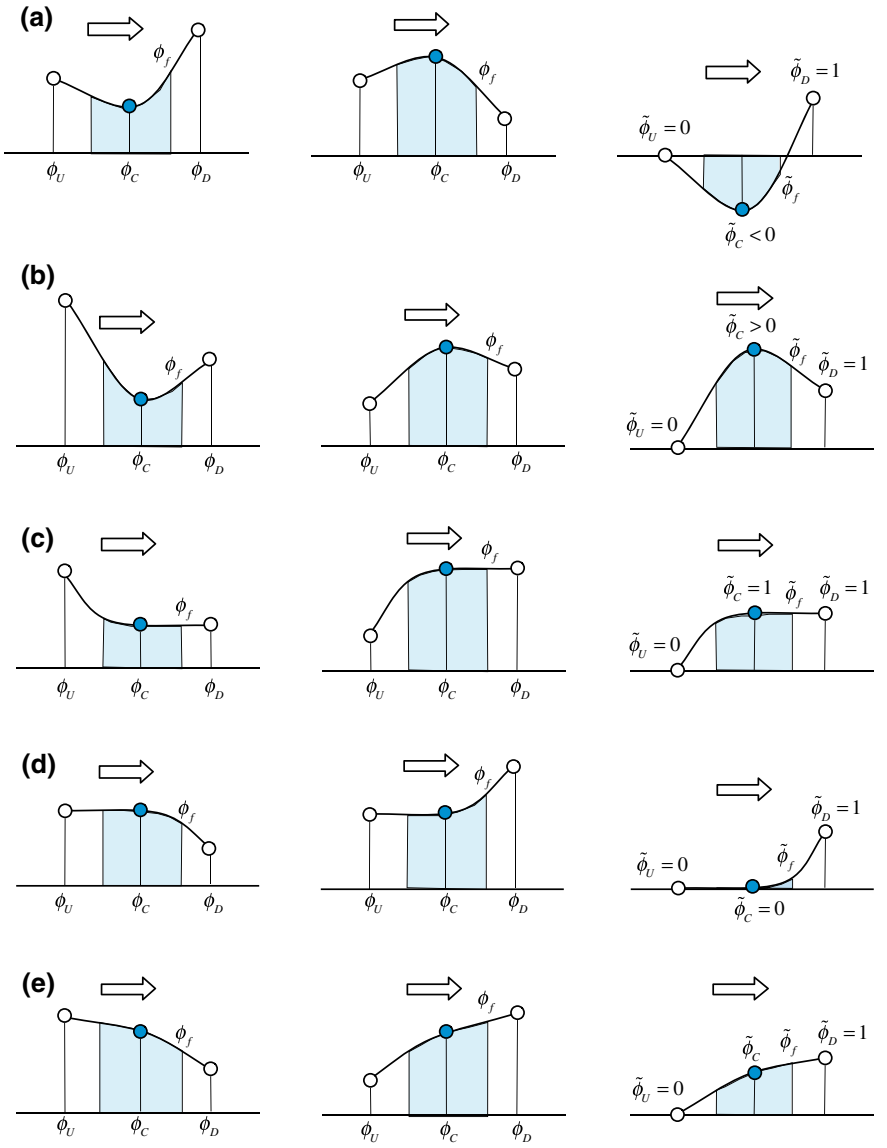


Fig. 12.2 Schematics of the situations when **a** $\tilde{\phi}_C < 0$, **b** $\tilde{\phi}_C > 1$, **c** $\tilde{\phi}_C = 1$, **d** $\tilde{\phi}_C = 0$, and **e** $0 < \tilde{\phi}_C < 1$

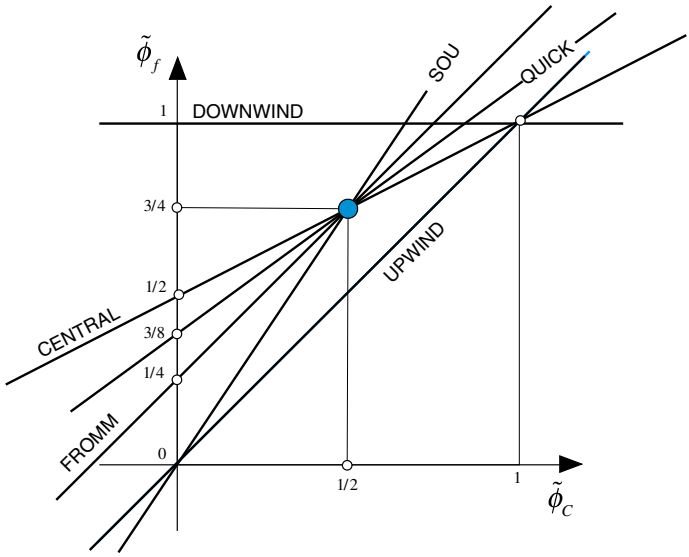


Fig. 12.3 Some HO schemes written in normalized form and plotted on a Normalized Variable Diagram (NVD)

Example 1
Derive the NVF form of the QUICK scheme.

Solution
Starting with

$$\phi_f = \frac{3}{8}\phi_D + \frac{3}{4}\phi_C - \frac{1}{8}\phi_U$$

Applying normalization to both sides yields

$$\frac{\phi_f - \phi_U}{\phi_D - \phi_U} = \frac{\left(\frac{3}{8}\phi_D + \frac{3}{4}\phi_C - \frac{1}{8}\phi_U\right) - \phi_U}{\phi_D - \phi_U}$$

noting that

$$\frac{3}{8} + \frac{3}{4} - \frac{1}{8} = \frac{3}{8} + \frac{6}{8} - \frac{1}{8} = \frac{8}{8} = 1$$

the following is obtained:

$$\frac{\phi_f - \phi_U}{(\phi_D - \phi_U)} = \frac{\frac{3}{8}(\phi_D - \phi_U) + \frac{3}{4}(\phi_C - \phi_U) - \frac{1}{8}(\phi_U - \phi_U)}{(\phi_D - \phi_U)}$$
$$= \frac{3}{8} + \frac{3}{4}\left(\frac{\phi_C - \phi_U}{\phi_D - \phi_U}\right)$$

thus

$$\tilde{\phi}_f = \frac{3}{8} + \frac{3}{4}\tilde{\phi}_C$$

The NVD reveals that except for the first order upwind and downwind schemes, all second order and third order schemes pass through the point $Q(0.5, 0.75)$ (for uniform grids). In fact, it can be shown that for a scheme to be second order accurate it has to pass through Q. If, in addition, its slope at Q is 0.75 then it will be third order accurate (e.g., QUICK). The upwind scheme was shown to be very diffusive, while the downwind scheme very compressive (anti-diffusive). Therefore, from the NVD it can graphically be deduced that any scheme whose functional relationship is close to the upwind scheme is diffusive while anyone close to the downwind scheme is compressive.

Example 2
Show that for schemes developed over uniform cartesian grids to be second order accurate their functional relationships should pass through the point $Q(1/2, 3/4)$ *in the NVD.*

Solution
Second order schemes involve three points. Expanding ϕ_C, ϕ_U, and ϕ_D in terms of ϕ_f, the following is obtained:

$$\phi_C = \phi_f - \frac{1}{2}\Delta x\phi'_f + O(\Delta x^2)$$
$$\phi_U = \phi_f - \frac{3}{2}\Delta x\phi'_f + O(\Delta x^2)$$
$$\phi_D = \phi_f + \frac{1}{2}\Delta x\phi'_f + O(\Delta x^2)$$

The value of ϕ_f is generally obtained as a combination of the values at the three locations as

$$a\phi_C + b\phi_U + c\phi_D = (a + b + c)\phi_f + \left(-\frac{1}{2}a - \frac{3}{2}b + \frac{1}{2}c\right)\Delta x\phi'_f + O(\Delta x^2)$$

The value of ϕ_f will be second order accurate if

$$\left(-\frac{1}{2}a-\frac{3}{2}b+\frac{1}{2}c\right)=0\Rightarrow b=\frac{c-a}{3}$$

A first order approximation of ϕ_f is obtained as

$$\begin{aligned}(a+b+c)\phi_f &= a\phi_C+b\phi_U+c\phi_D\Rightarrow \phi_f\\ &=\frac{a}{(a+b+c)}\phi_C+\frac{b}{(a+b+c)}\phi_U\\ &\quad+\frac{c}{(a+b+c)}\phi_D\\ &\Rightarrow (a+b+c)\tilde{\phi}_f=a\tilde{\phi}_C+c\end{aligned}$$

For the above approximation to be second order accurate the following should be true:

$$\left(a+\frac{c-a}{3}+c\right)\tilde{\phi}_f=a\tilde{\phi}_C+c\Rightarrow\left(\frac{2a+4c}{3}\right)\tilde{\phi}_f=a\tilde{\phi}_C+c$$

The above equality will be satisfied for any value of a and c, and consequently any second order scheme, when $\tilde{\phi}_f=3/4$ and $\tilde{\phi}_C=1/2$, in which case

$$\left(\frac{2a+4c}{3}\right)\tilde{\phi}_f=a\tilde{\phi}_C+c\Rightarrow\left(\frac{2a+4c}{3}\right)\frac{3}{4}=\frac{1}{2}a+c\Rightarrow\frac{1}{2}a+c=\frac{1}{2}a+c$$

Therefore all second order schemes pass through the point $Q(1/2,\,3/4)$.

The HO schemes presented in the previous chapter were shown to drastically decrease the truncation error suffered by the first order upwind scheme, while remaining stable. Still, one of the main shortcomings of these schemes is their unboundedness, i.e., their tendency to produce under/overshoots and even oscillations near sudden jumps or steep gradients in the convected variable (see Figs. 11.14b and 11.17). While in some applications small overshoots and/or oscillations may be tolerable, in others, they can lead to catastrophic results, such as in turbulent flow calculations where the convected variable can be the viscosity coefficient.

This oscillatory behavior near steep gradients characterizes all HO linear convective schemes. In fact these schemes are not monotonous in the sense that they produce local maxima and/or minima, i.e., they are not extrema preserving. For a scheme to be extrema preserving, maxima in the solution must be non-increasing and minima non-decreasing (the scheme should not produce over/under shoots). In fact it

was demonstrated by Godunov and Ryabenki [5] that any linear numerical scheme that is monotone can be at most first-order accurate. This implies that all higher order linear schemes cannot be monotonicity preserving, and that to construct monotonicity preserving schemes, non-linear limiter functions should be used. With this understanding, work on developing high order oscillation-free convection schemes resulted in several techniques [6–10] that can be grouped under two categories. In the first approach [11–13] a limited anti-diffusive flux is added to a first-order upwind scheme in such a way that the resulting scheme is capable of resolving sharp gradients without oscillations. In the second category, a smoothing diffusive flux is introduced into an unbounded HO scheme to damp unphysical oscillations [14–17].

Due to their multi-step nature and the difficulty in balancing the two fluxes, flux blending techniques tend to be very expensive numerically. This is why in this book two approaches for developing HR schemes falling under the flux limiter method will be presented. The first follows a composite procedure whereby high order schemes are combined with bounded low order ones, with the switch between them being controlled by a certain criterion [18]. The second method is based on adding to a diffusive first order upwind term an anti-diffusive flux multiplied by a flux limiter. In this case, the resulting HR schemes are also denoted by Total Variational Diminishing (TVD) schemes as explained in a later section.

The composite schemes approach will be presented first within the framework of the Normalized Variable Formulation (NVF) and will be visualized on a Normalized Variable Diagram (NVD). Therefore the NVF and NVD are first described. The use of the NVD will be instrumental for the definition of a criterion that ensures the boundedness of any high order interpolation scheme.

12.2 The Convection Boundedness Criterion (CBC)

A numerical scheme is expected to preserve the physical properties of the phenomenon it is trying to describe or approximate. Therefore the conditions that a bounded convection scheme should satisfy can best be understood by analyzing the physical properties of convection. Since convection transports fluid properties from upstream to downstream, then approximation to convection should possess this transportive attribute. Thus, numerical convection schemes should be upwind biased or else they will lack the convective stability. Therefore in addition to the values at the nodes straddling the interface ϕ_C and ϕ_D, the value at the far upwind node, i.e., ϕ_U, is also important in analyzing advective schemes. Values at nodes farther away are less important. In the NVF presented above, values are normalized such that the effect of ϕ_U is also considered. This is extremely useful as it helps identifying the conditions for which the numerical convection scheme is monotone. Whereas Spekreise's [19] and Barth and Jespersen [20] definition of a monotone scheme (or bounded scheme) involves all neighbors surrounding the face, Leonard [21] and Gaskel and Lau [4] based their definition of monotonicity only on the neighboring points along the local coordinate system such that

$$\min(\phi_C, \phi_D) \leq \phi_f \leq \max(\phi_C, \phi_D) \tag{12.11}$$

Normalizing, the above condition becomes

$$\min\left(\tilde{\phi}_C, 1\right) \leq \tilde{\phi}_f \leq \max\left(\tilde{\phi}_C, 1\right) \tag{12.12}$$

The Convection Boundedness Criterion (CBC) for implicit steady state flow calculation developed by Gaskell and Lau states that for a scheme to have the boundedness property its functional relationship should be continuous, should be bounded from below by $\tilde{\phi}_C$ and from above by unity, and should pass through the points (0, 0) and (1, 1), in the monotonic range $\left(0 < \tilde{\phi}_C < 1\right)$, and for $\tilde{\phi}_C > 1$ or $\tilde{\phi}_C < 0$, the functional relationship $f\left(\tilde{\phi}_C\right)$ should be equal to $\tilde{\phi}_C$. The above conditions illustrated on an NVD in Fig. 12.4, can be mathematically formulated as

$$\tilde{\phi}_f = \begin{cases} f\left(\tilde{\phi}_C\right) & \textit{continuous} \\ f\left(\tilde{\phi}_C\right) = 1 & \textit{if } \tilde{\phi}_C = 1 \\ f\left(\tilde{\phi}_C\right) \textit{ with } \tilde{\phi}_C < f\left(\tilde{\phi}_C\right) < 1 & \textit{if } 0 < \tilde{\phi}_C < 1 \\ f\left(\tilde{\phi}_C\right) = 0 & \textit{if } \tilde{\phi}_C = 0 \\ f\left(\tilde{\phi}_C\right) = \tilde{\phi}_C & \textit{if } \tilde{\phi}_C < 0 \textit{ or } \tilde{\phi}_C > 1 \end{cases} \tag{12.13}$$

The Convection Boundedness Criterion is quite intuitive and can be interpreted by referring to Figs. 12.4 and 12.5. When ϕ_C is in a monotonic profile the interpolation profile at the cell surface should not yield any new extremum. Thus it is constrained by the ϕ values at the nodes straddling the face. As the value of ϕ_C get closer to ϕ_D while still in the monotonic regime, the value of ϕ_f will also tend toward ϕ_D. When ϕ_C becomes equal to ϕ_D, then ϕ_f also becomes equal to ϕ_D and

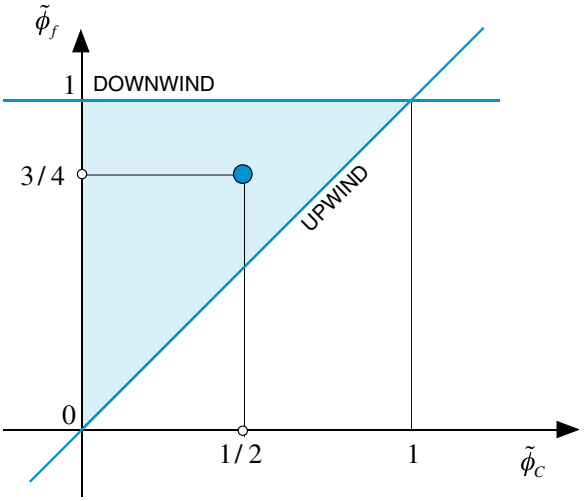


Fig. 12.4 The Convection Boundedness Criterion (CBC) on an NVD Diagram showing the region where $\tilde{\phi}_f$ is bounded

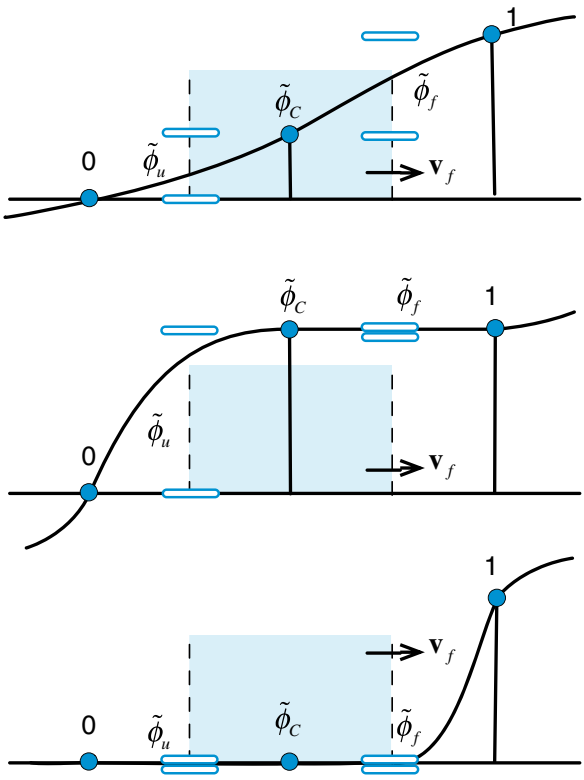


Fig. 12.5 Values of $\tilde{\phi}_f$ and the Convection Boundedness Criterion

thus the condition that $\left(\tilde{\phi}_C, \tilde{\phi}_f\right)$ passes through the point (1, 1). When the value of ϕ_C is such that $\tilde{\phi}_C > 1$, ϕ_f is assigned the upwind value, i.e., ϕ_C. This has the effect of yielding the largest outflow condition possible while fulfilling the condition that ϕ_f is bounded by the nodes straddling the cell face. This behavior means that any undue oscillation will be damped since ϕ_C will tend to a lower value because outflow is larger than inflow in these conditions.

Therefore if there is no external physical mechanism to yield the extrema (a source term for example) the extrema will die out. A similar mechanism takes place when $\tilde{\phi}_C < 0$. However as ϕ_C gets closer to ϕ_U coming from the non-monotonic region, ϕ_f will be equal to the upwind value ϕ_C until $\phi_C = \phi_U$ implying the condition that the profile of ϕ_f passes through the point (0, 0). When $\tilde{\phi}_C < 0$ or $\tilde{\phi}_C > 1$ the solution will be in a region where convection is dominant and the upwind approximation will be an excellent one.

12.3 High Resolution (HR) Schemes

Constructing a bounded HO scheme, i.e., a HR scheme, using the NVD is relatively a simple exercise. Any high order base scheme can be bounded using an ad-hoc set of curves.

To construct a HR scheme, the monotonic profile in the range $0 \le \tilde{\phi}_C \le 1$ should pass through the points (0, 0) and (1, 1), while remaining within the upper triangular shaded region on the NVD (Fig. 12.4). On the other hand, in the non-monotonic range, i.e., $\tilde{\phi}_C < 0$ and/or $\tilde{\phi}_C > 1$, the profile should follow the upwind profile. A number of well-known High-Resolution schemes built in this manner are illustrated in Fig. 12.6.

For improved convergence behavior, any composite HR scheme should avoid hard angles at its profile connection points as well as at its horizontal and vertical profiles. For example with the SMART scheme, which is constructed using the QUICK scheme, the convergence can be substantially improved by a minor modification to the vertical portion of its composite profile using $\tilde{\phi}_f = 3\tilde{\phi}_C$ in the region $0 \le \tilde{\phi}_C \le 1/6$. For the STOIC schemes the modification is applied in the $0 \le \tilde{\phi}_C \le 1/5$ region. Also the horizontal portion of the composite profile for both SMART and STOIC may be slightly modified to further improve convergence. For example, one such modification is to impose a linear profile for $9/10 \le \tilde{\phi}_f \le 1$, which corresponds to $7/10 \le \tilde{\phi}_C \le 1$ altering the last portion of the profile to be given by $\tilde{\phi}_f = \tilde{\phi}_C/3 + 2/3$. Other modifications are also possible (e.g., one may decide to modify the horizontal portion of the composite profile in the $0.95 \le \tilde{\phi}_f \le 1$ region). A similar modification can also be made for the bounded CD scheme to improve its convergence characteristics. The modified NVDs of SMART, STOIC, and SUPERBEE schemes are shown in Fig. 12.7.

Mathematically, the functional relationships of the composite HR schemes displayed in Figs. 12.6 and 12.7 are given by

$$\text{MINMOD} \quad \tilde{\phi}_f = \begin{cases} \frac{3}{2}\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \frac{1}{2} \\ \frac{1}{2}\tilde{\phi}_C + \frac{1}{2} & \frac{1}{2} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.14}$$

$$\text{Bounded CD} \quad \tilde{\phi}_f = \begin{cases} \frac{1}{2}\tilde{\phi}_C + \frac{1}{2} & 0 \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.15}$$

$$\text{OSHER} \quad \tilde{\phi}_f = \begin{cases} \frac{3}{2}\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \frac{2}{3} \\ 1 & \frac{2}{3} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.16}$$

$$\text{SMART} \quad \tilde{\phi}_f = \begin{cases} \frac{3}{4}\tilde{\phi}_C + \frac{3}{8} & 0 \le \tilde{\phi}_C \le \frac{5}{6} \\ 1 & \frac{5}{6} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.17}$$

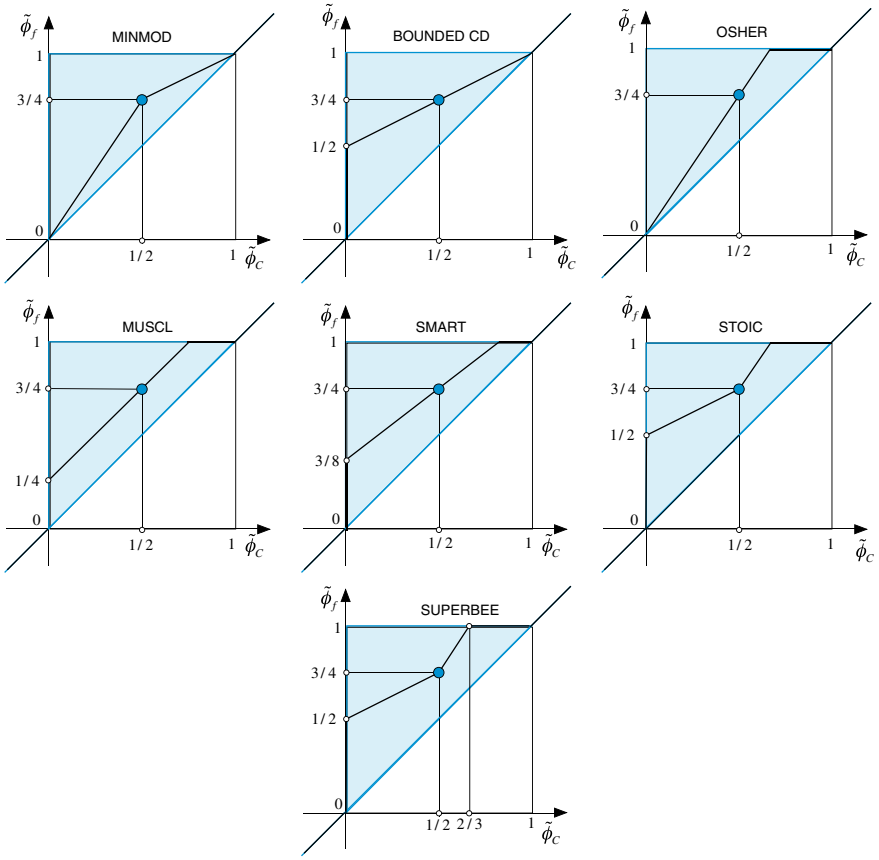


Fig. 12.6 NVD of several HR schemes

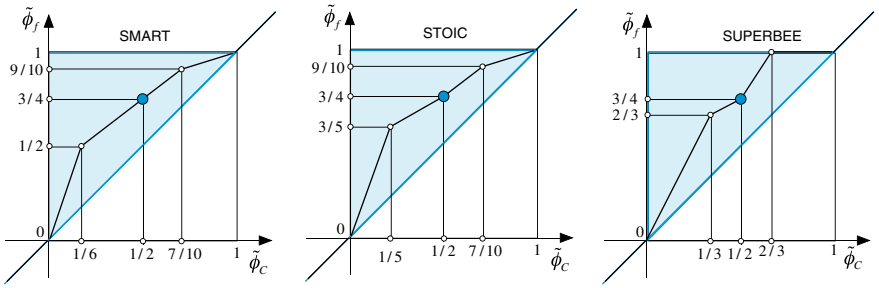


Fig. 12.7 NVD of the modified SMART, STOIC, and SUPERBEE schemes

$$\text{Modified SMART} \quad \tilde{\phi}_f = \begin{cases} 3\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \dfrac{1}{6} \\ \dfrac{3}{4}\tilde{\phi}_C + \dfrac{3}{8} & \dfrac{1}{6} \le \tilde{\phi}_C \le \dfrac{7}{10} \\ \dfrac{1}{3}\tilde{\phi}_C + \dfrac{2}{3} & \dfrac{7}{10} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.18}$$

$$\text{STOIC} \quad \tilde{\phi}_f = \begin{cases} \dfrac{1}{2}\tilde{\phi}_C + \dfrac{1}{2} & 0 \le \tilde{\phi}_C \le \dfrac{1}{2} \\ \dfrac{3}{4}\tilde{\phi}_C + \dfrac{3}{8} & \dfrac{1}{2} \le \tilde{\phi}_C \le \dfrac{5}{6} \\ 1 & \dfrac{5}{6} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.19}$$

$$\text{Modified STOIC} \quad \tilde{\phi}_f = \begin{cases} 3\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \dfrac{1}{5} \\ \dfrac{1}{2}\tilde{\phi}_C + \dfrac{1}{2} & \dfrac{1}{5} \le \tilde{\phi}_C \le \dfrac{1}{2} \\ \dfrac{3}{4}\tilde{\phi}_C + \dfrac{3}{8} & \dfrac{1}{2} \le \tilde{\phi}_C \le \dfrac{7}{10} \\ \dfrac{1}{3}\tilde{\phi}_C + \dfrac{2}{3} & \dfrac{7}{10} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.20}$$

$$\text{MUSCL} \quad \tilde{\phi}_f = \begin{cases} 2\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \dfrac{1}{4} \\ \tilde{\phi}_C + \dfrac{1}{4} & \dfrac{1}{4} \le \tilde{\phi}_C \le \dfrac{3}{4} \\ 1 & \dfrac{3}{4} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.21}$$

$$\text{SUPERBEE} \quad \tilde{\phi}_f = \begin{cases} \dfrac{1}{2} + \dfrac{1}{2}\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \dfrac{1}{2} \\ \dfrac{3}{2}\tilde{\phi}_C & \dfrac{1}{2} \le \tilde{\phi}_C \le \dfrac{2}{3} \\ 1 & \dfrac{2}{3} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.22}$$

$$\text{Modified SUPERBEE} \quad \tilde{\phi}_f = \begin{cases} 2\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \frac{1}{3} \\ \frac{1}{2}\tilde{\phi}_C + \frac{1}{2} & \frac{1}{3} \le \tilde{\phi}_C \le \frac{1}{2} \\ \frac{3}{2}\tilde{\phi}_C & \frac{1}{2} \le \tilde{\phi}_C \le \frac{2}{3} \\ 1 & \frac{2}{3} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases} \tag{12.23}$$

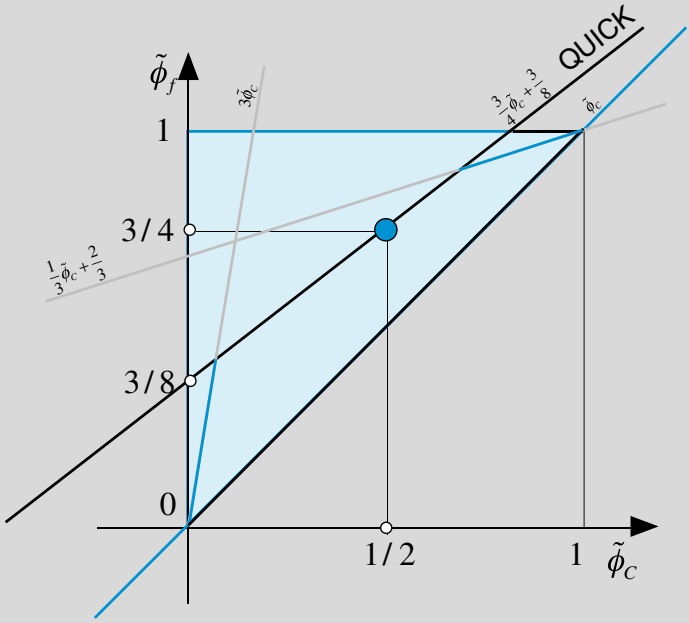
Many other schemes were developed following this methodology such as CLAM [22], UTOPIA [23], SHARP [8], and ULTRA-SHARP [24, 25], to cite a few.

Example 3
Derive the NVF form of the SMART scheme.

Solution
Starting with the QUICK scheme, we have

$$\tilde{\phi}_{f,QUICK} = \frac{3}{4}\tilde{\phi}_C + \frac{3}{8}$$

On the NVF diagram this looks like the black line. In order to enforce the CBC, the interpolation profile of the QUICK scheme is modified to follow the blue lines as shown in the figure below



Thus the bounded QUICK scheme, now called the SMART scheme, is written as

$$\tilde{\phi}_f = \begin{cases} 3\tilde{\phi}_C & 0 \le \tilde{\phi}_C \le \dfrac{1}{6} \\ \dfrac{3}{4}\tilde{\phi}_C + \dfrac{3}{8} & \dfrac{1}{6} \le \tilde{\phi}_C \le \dfrac{7}{10} \\ \dfrac{1}{3}\tilde{\phi}_C + \dfrac{2}{3} & \dfrac{7}{10} \le \tilde{\phi}_C \le 1 \\ \tilde{\phi}_C & \textit{elsewhere} \end{cases}$$

12.4 The TVD Framework

Another popular approach for developing HR convective schemes is the Total Variation Diminishing (TVD) framework. In solving numerically an advection partial differential equation for a variable ϕ of the form presented so far, Total Variation (*TV*) is defined as

$$TV = \sum_i \left|\phi_{i+1} - \phi_i\right| \tag{12.24}$$

where i represents the index of a node in the spatial solution domain. A numerical method is said to be Total Variation Diminishing (TVD) if the *TV* in the solution does not increase with time. Mathematically this is equivalent to

$$TV\left(\phi^{t+\Delta t}\right) \le TV(\phi^t) \tag{12.25}$$

In his seminal paper, Harten [26] proved that a monotone scheme is TVD, and a TVD scheme is monotonicity preserving. A monotonicity preserving scheme does not create any new local extrema within the solution domain, i.e., the value of a local minimum is non-decreasing, and the value of a local maximum is non-increasing.

It is not intended here to give full mathematical derivations of the TVD approach. Rather, the intention is simply to explain the methodology for constructing TVD schemes. The approach used is based on the work of Sweby [27].

Consider the unsteady one-dimensional convection equation (11.73), which was used in the previous chapter to study the stability of convection schemes. In the absence of diffusion and sources this equation reduces to

$$\frac{\partial(\rho\phi)}{\partial t} = \underbrace{-\frac{\partial(\rho u\phi)}{\partial x}}_{RHS} \tag{12.26}$$

The general discretized form of the *RHS* term based on a five point stencil can be written as

$$RHS = -a(\phi_C - \phi_U) + b(\phi_D - \phi_C) \tag{12.27}$$

where U, C, and D represent the far upstream, upstream, and downstream nodes shown in Fig. 12.1a. Sweby and Harten proved that a sufficient condition for a numerical scheme presented by Eq. (12.27) to be TVD or monotone is for the coefficients per unit mass flow rate to satisfy the inequalities

$$a \geq 0, \; b \geq 0, \quad \text{and} \quad 0 \leq a + b \leq 1 \tag{12.28}$$

where the expressions for the coefficients a and b depend on the adopted convection scheme. Referring back to the convection schemes presented above, it was found that the first order upwind scheme is very diffusive while the second order central difference scheme is highly dispersive. The need is for a scheme that lies somewhere between the upwind and the central difference schemes, i.e., a scheme that has the stability of the upwind scheme and the accuracy of the central difference scheme. Such a scheme can be constructed starting from the central difference scheme written as

$$\phi_f = \underbrace{\frac{1}{2}(\phi_D + \phi_C)}_{CD} = \underbrace{\phi_C}_{upwind} + \underbrace{\frac{1}{2}(\phi_D - \phi_C)}_{anti-diffusive\,flux} \tag{12.29}$$

where the notation used earlier is adopted with C denoting the upwind node, D the downwind node, and f the value at the cell face straddling the C and D nodes. As implied by Eq. (12.29), the central difference scheme can be written as the sum of the upwind scheme and a flux which is supposed to be anti-diffusive since the CD scheme is dispersive. This flux is desirable as it makes the scheme second order accurate. The side effect is the unphysical oscillation it creates due to the decrease in numerical diffusion. Therefore a better approach would be one in which a portion of this anti-diffusive flux is added to the upwind scheme in such a way that the second order accuracy is preserved without creating any unphysical oscillations. One way to do that is to multiply this flux by a limiter function (also called limiter or flux limiter) that will prevent its excessive use in regions where oscillations might occur (e.g., across large gradients) while maximizing its contribution in smooth areas. Denoting such a limiter by $\psi(r)$, where r is usually taken as the ratio of two consecutive gradients, ϕ_f is calculated as

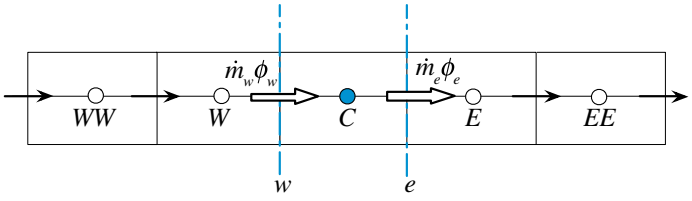


Fig. 12.8 Convective fluxes in a one dimensional domain

$$\phi_f = \phi_C + \frac{1}{2}\psi(r_f)(\phi_D - \phi_C) \text{ with } r_f = \frac{\phi_C - \phi_U}{\phi_D - \phi_C} \tag{12.30}$$

where U is the node upwind to C and D the node downwind to C. In order to preserve the sign of the anti-diffusive flux, $\psi(r_f)$ is taken to be nonnegative. Therefore developing a TVD scheme reduces to finding limiters that will make the numerical scheme TVD or monotone. The conditions that these limiters have to satisfy in order for the convection scheme to be monotonicity preserving are derived next by invoking the flux limiter in the discretization of the *RHS* of Eq. (12.27) via the interface values given by Eq. (12.30).

Considering the one dimensional domain shown in Fig. 12.8, the convective fluxes at the element faces are given by

$$\begin{aligned}
\dot{m}_e\phi_e &= \left[\phi_C + \frac{1}{2}\psi(r_e^+)(\phi_E - \phi_C)\right]\|\dot{m}_e, 0\| \\
&\quad - \left[\phi_E + \frac{1}{2}\psi(r_e^-)(\phi_C - \phi_E)\right]\|-\dot{m}_e, 0\| \\
\dot{m}_w\phi_w &= \left[\phi_C + \frac{1}{2}\psi(r_w^+)(\phi_W - \phi_C)\right]\|\dot{m}_w, 0\| \\
&\quad - \left[\phi_W + \frac{1}{2}\psi(r_w^-)(\phi_C - \phi_W)\right]\|-\dot{m}_w, 0\| \\
r_e^+ &= \frac{\phi_C - \phi_W}{\phi_E - \phi_C}, r_e^- = \frac{\phi_E - \phi_{EE}}{\phi_C - \phi_E}, \\
r_w^+ &= \frac{\phi_C - \phi_E}{\phi_W - \phi_C}, r_w^- = \frac{\phi_W - \phi_{WW}}{\phi_C - \phi_W}
\end{aligned} \tag{12.31}$$

To simplify the derivations to follow, a positive velocity is assumed. Under these conditions, the discretized form of the *RHS* of Eq. (12.24) is obtained as

$$RHS = -\dot{m}_e\left[\phi_C + \frac{1}{2}\psi(r_e^+)(\phi_E - \phi_C)\right] - \dot{m}_w\left[\phi_W + \frac{1}{2}\psi(r_w^-)(\phi_C - \phi_W)\right] \tag{12.32}$$

while the continuity equation is given by

$$\dot{m}_e + \dot{m}_w = 0 \Rightarrow \dot{m}_w = -\dot{m}_e \tag{12.33}$$

Invoking the continuity constraint, the *RHS* equation can be rearranged into

$$\begin{aligned}
RHS &= -\dot{m}_e\left[1 + \frac{1}{2}\psi(r_e^+)\underbrace{\frac{(\phi_E - \phi_C)}{(\phi_C - \phi_W)}}_{1/r_e^+} - \frac{1}{2}\psi(r_w^-)\right](\phi_C - \phi_W) \\
&= -\dot{m}_e\left[1 + \frac{1}{2}\frac{\psi(r_e^+)}{r_e^+} - \frac{1}{2}\psi(r_w^-)\right](\phi_C - \phi_W)
\end{aligned} \tag{12.34}$$

Comparing Eq. (12.34) with Eq. (12.27), the values of a and b are found to be

$$a = 1 + \frac{1}{2}\frac{\psi(r_e^+)}{r_e^+} - \frac{1}{2}\psi(r_w^-) \quad b = 0 \tag{12.35}$$

For the scheme to be TVD, the following should hold [Eq. (12.28)]:

$$0 \le 1 + \frac{1}{2}\frac{\psi(r_e^+)}{r_e^+} - \frac{1}{2}\psi(r_w^-) \le 1 \tag{12.36}$$

which can be expanded to

$$\begin{aligned} 1 + \frac{1}{2}\frac{\psi(r)}{r} - \frac{1}{2}\psi(r) \ge 0 &\Rightarrow \frac{1}{2}\frac{\psi(r)}{r} - \frac{1}{2}\psi(r) \ge -1 \Rightarrow \psi(r) - \frac{\psi(r)}{r} \le 2 \\ 1 + \frac{1}{2}\frac{\psi(r)}{r} - \frac{1}{2}\psi(r) \le 1 &\Rightarrow \frac{1}{2}\frac{\psi(r)}{r} - \frac{1}{2}\psi(r) \le 0 \Rightarrow \psi(r) - \frac{\psi(r)}{r} \ge 0 \end{aligned} \tag{12.37}$$

or simply

$$0 \le \psi(r) - \frac{\psi(r)}{r} \le 2 \tag{12.38}$$

If in addition to having $\psi(r) \ge 0$, a condition is imposed whereby $\psi(r) = 0$ for negative values of r, then the above conditions will be satisfied if

$$\psi(r) \le 2 \quad \text{and} \quad \psi(r) \le 2r \tag{12.39}$$

Combining all conditions that the limiter has to satisfy to produce a TVD scheme, a criterion similar to the CBC can be developed and is given by

$$\psi(r) = \begin{cases} \min(2r, 2) & r > 0 \\ 0 & r \le 0 \end{cases} \tag{12.40}$$

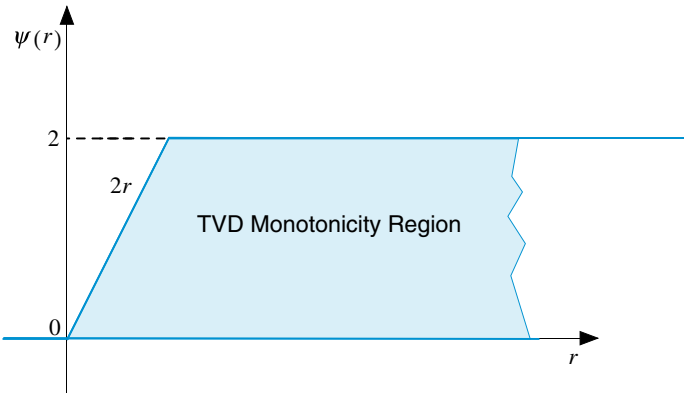


Fig. 12.9 TVD monotonicity region on a $r - \psi$ diagram

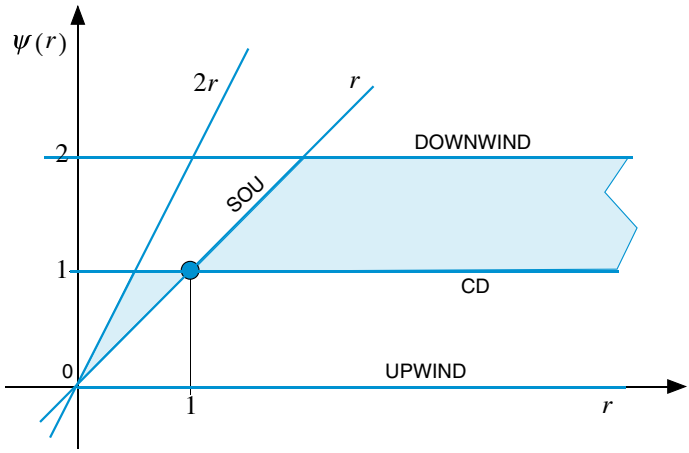


Fig. 12.10 Limiters of SOU and CD schemes on a $r - \psi$ diagram

As depicted in Fig. 12.9, these conditions can be drawn on a $r - \psi$ diagram, which is also denoted by Sweby's diagram, to show the TVD monotonicity region (blue region in the plot). Using this diagram, it is simple to grasp the formulation of TVD schemes. Any flux limiter $\psi(r)$ formulated to lie within the TVD monotonicity region yields a TVD scheme. Sweby's diagram is very similar to the NVD presented above.

The limiters for all schemes presented so far can be derived and their functional relationships drawn on Sweby's diagram. In specific the limiter of the CD is easily obtained from Eq. (12.29) as $\psi^{CD}(r_f) = 1$ while that of the SOU scheme can be computed as follows:

$$\begin{aligned} \phi_f &= \phi_C + \frac{1}{2}\psi^{SOU}(r_f)(\phi_D - \phi_C) = \frac{3}{2}\phi_C - \frac{1}{2}\phi_U \\ \Rightarrow \psi^{SOU}(r_f) &= \frac{\phi_C - \phi_U}{\phi_D - \phi_C} = r_f \end{aligned} \tag{12.41}$$

The limiters for both schemes are displayed in Fig. 12.10.

Sweby [27] also noted that because $\psi(r_f) = 0$ for $r_f < 0$, second order accuracy is lost at extrema of the solution. The SOU and CD schemes are second order schemes and by inspecting Fig. 12.10 it is clearly seen that both of them pass through the point (1, 1). In addition, as demonstrated in the work of Van Leer [28], any second order scheme can be written as a weighted average of the CD and SOU schemes. Thus for a scheme to be second order its limiter has to pass through the point (1, 1) and, as shown in Fig. 12.10, its limiter should lie in the region bordered by the CD and SOU limiters (blue region in the plot). The corresponding region on an NVD is shown in Fig. 12.11.

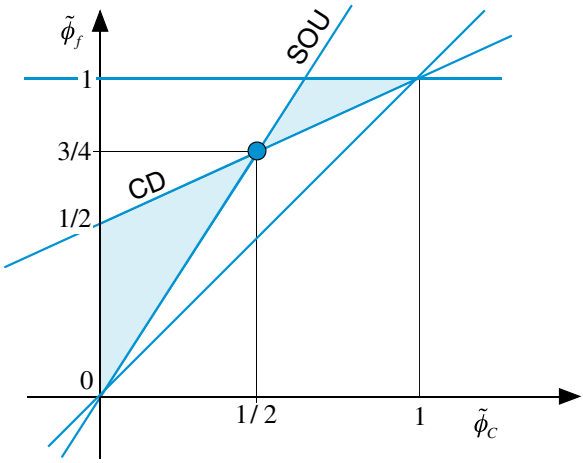


Fig. 12.11 Region on an NVD equivalent to the TVD monotonicity region on a Sweby's diagram for second order schemes

Adopting this approach and following the procedure used with the SOU scheme, the functional relationships of the limiters for many of the HO schemes presented above can be easily computed and are given by

$$\begin{cases} \textit{Upwind} & \psi(r_f) = 0 \\ \textit{Downwind} & \psi(r_f) = 2 \\ \textit{FROMM} & \psi(r_f) = \dfrac{1+r_f}{2} \\ \textit{SOU} & \psi(r_f) = r_f \\ \textit{CD} & \psi(r_f) = 1 \\ \textit{QUICK} & \psi(r_f) = \dfrac{3+r_f}{4} \end{cases} \tag{12.42}$$

The FROMM scheme is the average of the CD and SOU scheme. Its functional relationship is mathematically written as

$$\left.\begin{aligned} \phi_f &= \frac{1}{2}\left(\frac{\phi_C+\phi_D}{2}+\frac{3}{2}\phi_C-\frac{1}{2}\phi_U\right) = \phi_C+\frac{\phi_D-\phi_U}{4} \\ \phi_f &= \phi_C+\frac{1}{2}\psi(r_f)(\phi_D-\phi_C) \end{aligned}\right\} \Rightarrow \psi(r_f) = \frac{1+r_f}{2} \tag{12.43}$$

The functional relationships of these limiters are displayed in Fig. 12.12. With the exception of the upwind scheme limiter all others are seen not to be totally lying within the monotonicity region. As such these schemes are unbounded.

By limiting the $\psi(r_f)$ functions of the various schemes given above to lie within the monotonicity region displayed in Fig. 12.9, these HO schemes are transformed into HR TVD schemes. Many TVD schemes have been developed in that manner and the limiters for a number of them are shown in Fig. 12.13a–d with the functional relationships of their limiters given by

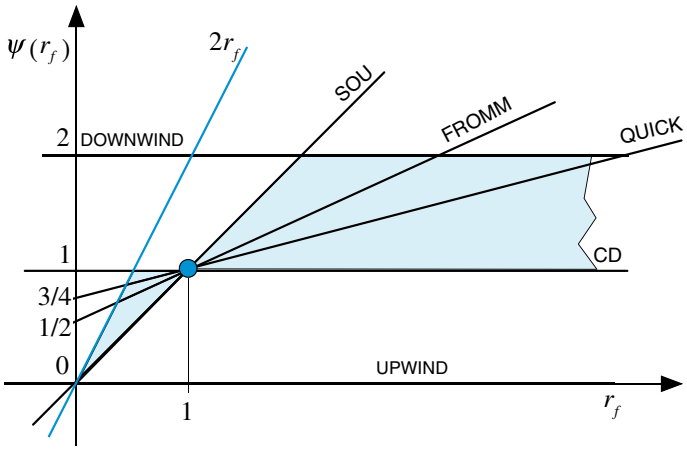


Fig. 12.12 High Order schemes and TVD monotonicity region on Sweby's diagram

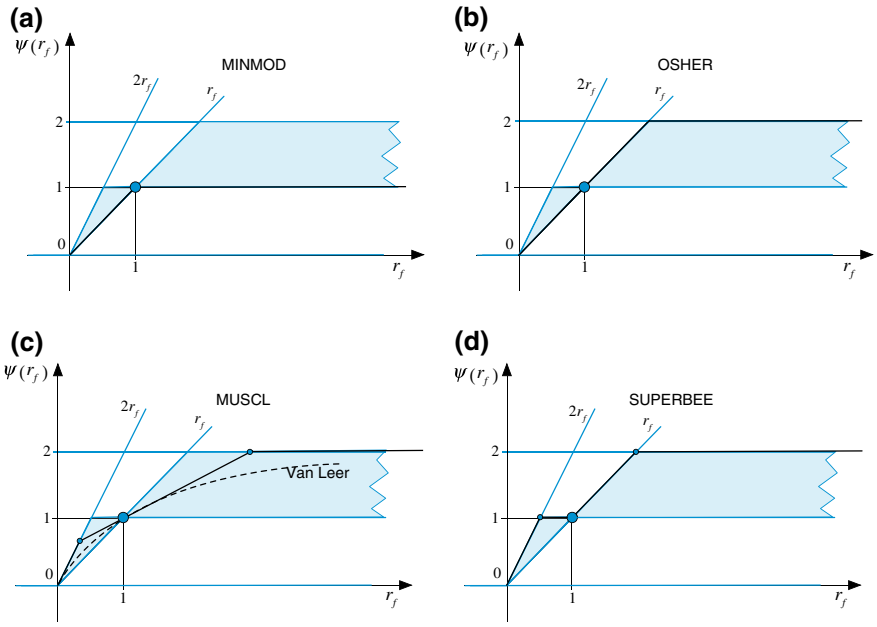


Fig. 12.13 Limiters of the **a** MINMOD, **b** OSHER, **c** MUSCL, and **d** SUPERBEE TVD schemes on a Sweby diagram

$$\begin{cases} SUPERBEE & \psi(r_f) = \max\left(0, \min\left(1, 2r_f\right), \min\left(2, r_f\right)\right) \\ MINMOD & \psi(r_f) = \max\left(0, \min\left(1, r_f\right)\right) \\ OSHER & \psi(r_f) = \max\left(0, \min\left(2, r_f\right)\right) \\ Van\ Leer & \psi(r_f) = \dfrac{r_f + |r_f|}{1 + |r_f|} \\ MUSCL & \psi(r_f) = \max\left(0, \min\left(2r_f, \left(r_f + 1\right)/2, 2\right)\right) \end{cases} \tag{12.44}$$

12.5 The NVF-TVD Relation

Both NVF and TVD formulations enforce Boundedness following different approaches, which can be demonstrated to be somewhat related. This is done by first deriving a relation between r_f and $\tilde{\phi}_C$, then comparing the NVF-CBC (Eq. 12.13) with the TVD-CBC (Eq. 12.40), and finally presenting the general transformation that allows the functional relationship of any TVD scheme to be written in the NVF framework and vice versa.

The relation between r_f and $\tilde{\phi}_C$ can be easily derived starting with the definition of r_f and is obtained as

$$\begin{aligned} r_f &= \frac{\phi_C - \phi_U}{\phi_D - \phi_C} = \frac{(\phi_C - \phi_U)/(\phi_D - \phi_U)}{(\phi_D - \phi_U + \phi_U - \phi_C)/(\phi_D - \phi_U)} \\ &= \frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C} \Rightarrow \tilde{\phi}_C = \frac{r_f}{1 + r_f} \end{aligned} \tag{12.45}$$

Using Eq. (12.45) a number of linear schemes can be compared in the two frameworks. The limiter $\psi(r_f) = 0$, which represents the Upwind scheme in the TVD formulation is also equivalent to the upwind scheme in the NVF formulation (i.e., $\tilde{\phi}_f = \tilde{\phi}_C$). This follows from the fact that $\psi(r_f) = 0 \Rightarrow \phi_f = \phi_U \Rightarrow \tilde{\phi}_f = \tilde{\phi}_C$. The upwind scheme is imposed as a limit for the TVD-CBC when $r_f \leq 0$, the equivalent condition in the NVF-CBC is obtained as

$$r_f \leq 0 \Rightarrow \frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C} \leq 0 \Rightarrow \begin{cases} \tilde{\phi}_C \leq 0 \\ \tilde{\phi}_C > 1 \end{cases} \tag{12.46}$$

These also represent the conditions for imposing the Upwind scheme in the NVF-CBC.

Moreover, on the NVF-CBC, the functional relationship has to increase monotonically in the region $0 \leq \tilde{\phi}_C \leq 1$. On Sweby's diagram the region extends over the interval $0 \leq r_f < +\infty$. Both regions represent the same interval as demonstrated by the following relation:

$$\tilde{\phi}_C \to 1 \Rightarrow r_f = \frac{\tilde{\phi}_C}{1-\tilde{\phi}_C} \to +\infty \tag{12.47}$$

Further, for the TVD-CBC condition

$$\psi(r_f) \leq 2 \tag{12.48}$$

the equivalent condition in the NVF-CBC can be obtained as follows:

$$\left.\begin{aligned} &\psi(r_f) = 2 \\ &\phi_f = \phi_C + \frac{1}{2}\psi(r_f)(\phi_D - \phi_C) \end{aligned}\right\} \Rightarrow \phi_f = \phi_C + (\phi_D - \phi_C) = \phi_D \Rightarrow \tilde{\phi}_f = 1 \tag{12.49}$$

Thus,

$$\psi(r_f) \leq 2 \Rightarrow \tilde{\phi}_f \leq 1 \tag{12.50}$$

which is the condition that should be satisfied by the NVF-CBC. The last condition imposed by the TVD-CBC on $\psi(r_f)$ is given by

$$\psi(r_f) \leq 2r_f \tag{12.51}$$

The equivalent condition using the NVF-CBC is obtained as

$$\left.\begin{aligned} &\psi(r_f) = 2r_f \\ &\phi_f = \phi_C + \frac{1}{2}\psi(r_f)(\phi_D - \phi_C) \end{aligned}\right\} \Rightarrow \phi_f = \phi_C + \frac{\phi_C - \phi_U}{\phi_D - \phi_C}(\phi_D - \phi_C) = 2\phi_C - \phi_U \tag{12.52}$$

which can be normalized to yield

$$\phi_f = 2\phi_C - \phi_U \Rightarrow \phi_f - \phi_U = 2\phi_C - 2\phi_U \Rightarrow \tilde{\phi}_f = 2\tilde{\phi}_C \tag{12.53}$$

This is more restrictive than the NVF-CBC and is the only difference between the two formulations. Based on this condition, the TVD-CBC and the modified NVF-CBC would look as shown in Fig. 12.14a with the monotonicity region reduced to the upwind line and the blue area. While the modified TVD-CBC and the NVF-CBC (i.e., the condition $\tilde{\phi}_C = 0$ on the NVF-CBC corresponds to $r_f = 0$ on the TVD-CBC) would look as shown in Fig. 12.14b. Regarding second order accuracy, it was stated that for a TVD scheme to be second order accurate it has to pass through the point (1, 1), i.e., $\psi(1) = 1$. The equivalent values using the NVF are found as

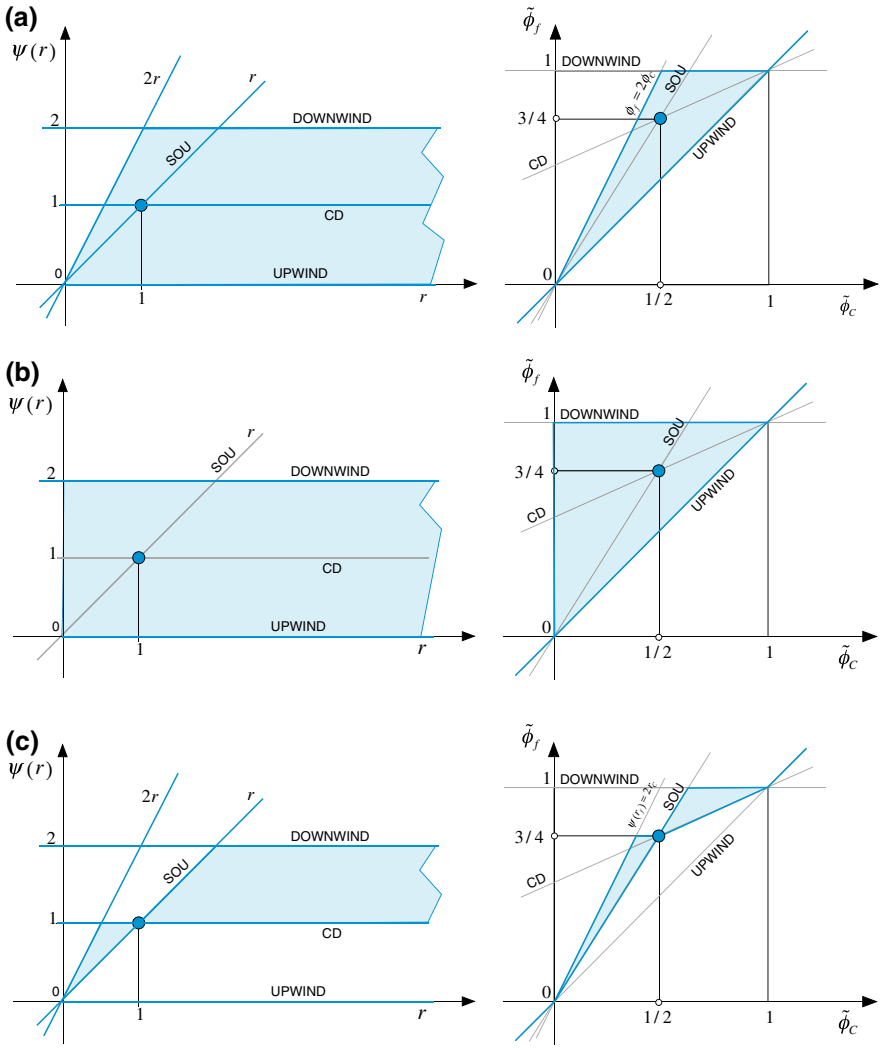


Fig. 12.14 **a** TVD-CBC on Sweby and Normalized Variable Diagrams. **b** NVF-CBC on Sweby and Normalized Variable Diagrams. **c** TVD-CBC on Sweby and Normalized Variable Diagrams for second order schemes

$$
\begin{aligned}
r_f = 1 \Rightarrow \frac{\tilde{\phi}_C}{1-\tilde{\phi}_C} = 1 \Rightarrow \tilde{\phi}_C = 1 - \tilde{\phi}_C \Rightarrow \tilde{\phi}_C = 0.5 \\
\phi_f = \phi_C + \frac{1}{2}\psi(1)(\phi_D - \phi_C) = \phi_C + \frac{1}{2}(\phi_D - \phi_C) = \frac{1}{2}(\phi_D + \phi_C) \\
\therefore \phi_f - \phi_U = \frac{1}{2}(\phi_D - \phi_U + \phi_C - \phi_U) \Rightarrow \tilde{\phi}_f = \frac{1}{2}\left(1 + \tilde{\phi}_C\right) = 0.75
\end{aligned}
\tag{12.54}
$$

which is exactly the point $Q(0.5, 0.75)$ found in the NFV. As stated earlier, Van Leer demonstrated that any second order scheme can be written as a weighted average of the CD and SOU schemes. Therefore its functional relationship should lie between the functional relationships of the CD and SOU schemes with their TVD-CBC monotonicity regions reduced to the upwind line and the blue area shown on a Sweby diagram and an NVD in Fig. 12.14c.

The above procedure can be generalized to transform any TVD scheme into an equivalent NVF scheme and vice versa. Starting with a scheme in the NVF framework, the value at the face ϕ_f is expressed as

$$\phi_f = f\left(\tilde{\phi}_C\right)(\phi_D - \phi_U) + \phi_U \text{ with } \tilde{\phi}_C = \frac{\phi_C - \phi_U}{\phi_D - \phi_U} \tag{12.55}$$

whereas for a TVD scheme ϕ_f is given by

$$\phi_f = \phi_C + \frac{1}{2}\psi(r_f)(\phi_D - \phi_C) \text{ with } r_f = \frac{\phi_C - \phi_U}{\phi_D - \phi_C} \tag{12.56}$$

Equating the above two ϕ_f equations, yields

$$\phi_f = \phi_C + \frac{1}{2}\psi(r_f)(\phi_D - \phi_C) = f\left(\tilde{\phi}_C\right)(\phi_D - \phi_U) + \phi_U \tag{12.57}$$

Thus

$$\psi(r_f)\frac{(\phi_D - \phi_C)}{(\phi_D - \phi_U)} = 2\frac{f\left(\tilde{\phi}_C\right)(\phi_D - \phi_U)}{(\phi_D - \phi_U)} - 2\frac{(\phi_C - \phi_U)}{(\phi_D - \phi_U)} = 2\left(f\left(\tilde{\phi}_C\right) - \tilde{\phi}_C\right) \tag{12.58}$$

The term on the left hand side of the above equation can be modified to

$$\psi(r_f)\frac{(\phi_D - \phi_C)}{(\phi_D - \phi_U)} = \psi(r_f)\frac{(\phi_D - \phi_U - \phi_C + \phi_U)}{(\phi_D - \phi_U)} = \psi(r_f)\left(1 - \tilde{\phi}_C\right) \tag{12.59}$$

leading to

$$\psi(r_f)\left(1 - \tilde{\phi}_C\right) = 2\frac{f\left(\tilde{\phi}_C\right)(\phi_D - \phi_U) - (\phi_C - \phi_U)}{(\phi_D - \phi_U)} = \psi(r_f) = 2\frac{f\left(\tilde{\phi}_C\right) - \tilde{\phi}_C}{1 - \tilde{\phi}_C} \tag{12.60}$$

Equation (12.60) may also be written as

$$f\left(\tilde{\phi}_C\right) = \frac{\psi(r_f) + 2r_f}{2(1 + r_f)} \tag{12.61}$$

As an example, the functional relationship of the UPWIND Scheme in the NVF framework is $\tilde{\phi}_f = \tilde{\phi}_C$, its TVD limiter is found as

$$\tilde{\phi}_f = \tilde{\phi}_C \Rightarrow \psi(r_f) = 2\frac{f\left(\tilde{\phi}_C\right) - \tilde{\phi}_C}{1 - \tilde{\phi}_C} = 2\frac{\tilde{\phi}_C - \tilde{\phi}_C}{1 - \tilde{\phi}_C} = 0 \tag{12.62}$$

The TVD limiter for the DOWNWIND Scheme is $\psi(r_f) = 2$, its NVF functional relationship can be obtained as

$$\tilde{\phi}_f = f\left(\tilde{\phi}_C\right) = \frac{\psi(r_f) + 2r_f}{2(1 + r_f)} = \frac{2 + 2r_f}{2(1 + r_f)} = 1 \tag{12.63}$$

Knowing the NVF form of the SOU scheme, its TVD limiter is computed as

$$\tilde{\phi}_f = \frac{3}{2}\tilde{\phi}_C \Rightarrow \psi(r_f) = 2\frac{\left(\frac{3}{2}\tilde{\phi}_C - \tilde{\phi}_C\right)}{\left(1 - \tilde{\phi}_C\right)} = 2\frac{\left(0.5\tilde{\phi}_C\right)}{\left(1 - \tilde{\phi}_C\right)} = \frac{\tilde{\phi}_C}{\left(1 - \tilde{\phi}_C\right)} = r_f \tag{12.64}$$

The same is applicable to other schemes.

Example 4
Starting with the TVD-Van Leer formulation, derive the NVF-Van Leer scheme.

Solution
The TVD-Van Leer limiter is given by

$$\psi(r_f) = \frac{r_f + |r_f|}{1 + |r_f|}.$$

Noting that

$$r_f = \frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C}$$

its TVD functional relationship is transformed to

$$\psi(r_f) = \frac{\frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C} + \left|\frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C}\right|}{1 + \left|\frac{\tilde{\phi}_C}{1 - \tilde{\phi}_C}\right|} = \frac{\frac{|1 - \tilde{\phi}_C|}{1 - \tilde{\phi}_C}\tilde{\phi}_C + |\tilde{\phi}_C|}{|1 - \tilde{\phi}_C| + |\tilde{\phi}_C|}$$

Combining the above equation with the TVD relationship in normalized form, which is given by,

$$\tilde{\phi}_f = \tilde{\phi}_C + \frac{1}{2}\psi(r_f)\left(1 - \tilde{\phi}_C\right)$$

yields

$$\begin{aligned}\tilde{\phi}_f &= \tilde{\phi}_C + \frac{1}{2}\psi(r_f)\left(1 - \tilde{\phi}_C\right) \\ &= \tilde{\phi}_C + \frac{1}{2}\frac{\frac{\left|1-\tilde{\phi}_C\right|}{1-\tilde{\phi}_C}\tilde{\phi}_C + \left|\tilde{\phi}_C\right|}{\left|1-\tilde{\phi}_C\right| + \left|\tilde{\phi}_C\right|}\left(1 - \tilde{\phi}_C\right)\end{aligned}$$

The following three cases are identified:

a. Case 1: $0 < \tilde{\phi}_C < 1$
In this case $\left|\tilde{\phi}_C\right| = \tilde{\phi}_C$ and $\left|1 - \tilde{\phi}_C\right| = 1 - \tilde{\phi}_C$, thus

$$\left.\begin{aligned}&\psi(r_f) = 2\tilde{\phi}_C \\ &\tilde{\phi}_f = \tilde{\phi}_C + \frac{1}{2}\psi(r_f)\left(1 - \tilde{\phi}_C\right)\end{aligned}\right\} \tilde{\phi}_f = \tilde{\phi}_C + \tilde{\phi}_C\left(1 - \tilde{\phi}_C\right) = 2\tilde{\phi}_C - \left(\tilde{\phi}_C\right)^2$$

b. Case 2: $\tilde{\phi}_C > 1$
In this case $\left|\tilde{\phi}_C\right| = \tilde{\phi}_C$ and $\left|1 - \tilde{\phi}_C\right| = \tilde{\phi}_C - 1$, thus

$$\left.\begin{aligned}&\psi(r_f) = 0 \\ &\tilde{\phi}_f = \tilde{\phi}_C + \frac{1}{2}\psi(r_f)\left(1 - \tilde{\phi}_C\right)\end{aligned}\right\} \tilde{\phi}_f = \tilde{\phi}_C$$

c. Case 3: $\tilde{\phi}_C < 0$
In this case $\left|\tilde{\phi}_C\right| = -\tilde{\phi}_C$ and $\left|1 - \tilde{\phi}_C\right| = 1 - \tilde{\phi}_C$, thus

$$\left.\begin{aligned}&\psi(r_f) = 0 \\ &\tilde{\phi}_f = \tilde{\phi}_C + \frac{1}{2}\psi(r_f)\left(1 - \tilde{\phi}_C\right)\end{aligned}\right\} \tilde{\phi}_f = \tilde{\phi}_C$$

Combining the results of the three cases into one NVF formulation, the functional relationship of the Van Leer Scheme becomes

$$\tilde{\phi}_{f,\text{VanLeer}} = \begin{cases} 2\tilde{\phi}_C - \left(\tilde{\phi}_C\right)^2 & 0 < \tilde{\phi}_C < 1 \\ \tilde{\phi}_C & \textit{otherwise} \end{cases}$$

12.6 HR Schemes in Unstructured Grid Systems

As mentioned in Chap. 11, another alternative that can be followed to overcome the hurdle of not having a clear upwind location U in unstructured grids, which is needed in the calculation of $\tilde{\phi}_C$ or r_f, is to create a virtual one. As depicted in Fig. 12.15, the easiest way is to assume U to lie on the line joining the nodes C and D such that C is the midpoint of the segment joining the points U and D. With this assumption and based on the analysis done earlier, the following can be written:

$$\phi_D - \phi_U = \nabla\phi_C \cdot \mathbf{d}_{UD} = 2\nabla\phi_C \cdot \mathbf{d}_{CD} \tag{12.65}$$

from which the value of ϕ_U is computed as

$$\phi_U = \phi_D - 2\nabla\phi_C \cdot \mathbf{d}_{CD} \tag{12.66}$$

where $\mathbf{d}_{CD}$ is the vector between the nodes C and D, and $\mathbf{d}_{UD}$ is the vector between nodes D and the *virtual* node U. As mentioned above U is constructed such that C is taken to be the centre of the UD segment. With the value of ϕ_U computed, the use of either the NVF or the TVD approach proceeds as described above.

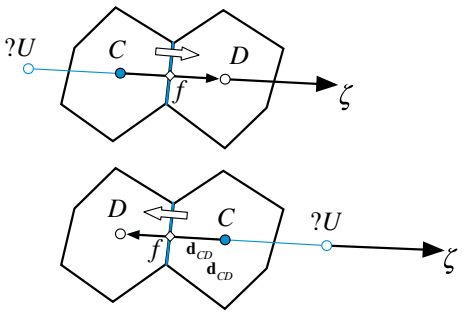


Fig. 12.15 Virtual upwind node in unstructured grids

12.7 Deferred Correction for HR Schemes

The numerical implementation of HR schemes is best understood through an example. For that purpose the multi dimensional advection equation with a source is considered. Again the velocity field is assumed to be known and the conservation equation in vector form is given by

$$\nabla \cdot (\rho \mathbf{v}\phi) = Q^{\phi} \tag{12.67}$$

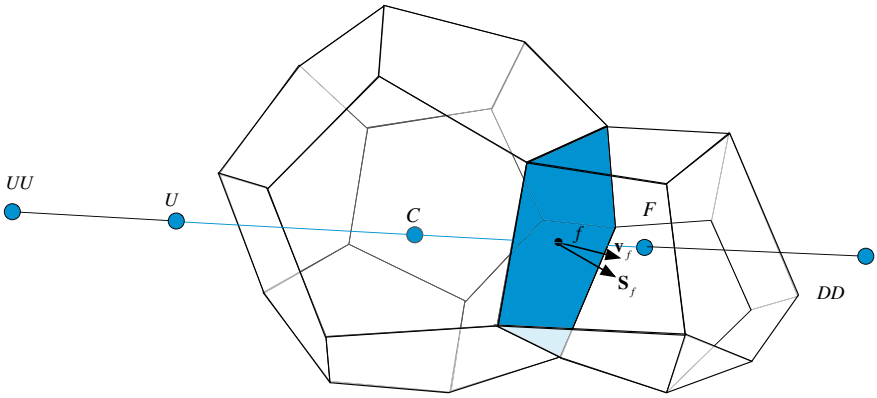


Fig. 12.16 Two three-dimensional elements, which represent part of an unstructured grid system

Integrating over the element of volume V_C shown in Fig. 12.16, applying the divergence theorem, and replacing the surface integral by a summation over the element faces, Eq. (12.67) becomes

$$\sum_{f\sim nb(C)} (\rho \mathbf{v}\phi)_f \cdot \mathbf{S}_f = Q_C^{\phi} V_C \tag{12.68}$$

Noticing that the mass flow rate at a cell face is given by

$$\dot{m}_f = (\rho \mathbf{v})_f \cdot \mathbf{S}_f \tag{12.69}$$

then Eq. (12.68) can be rewritten as

$$\sum_{f\sim nb(C)} \dot{m}_f \phi_f = Q_C^{\phi} \mathrm{V}_C \tag{12.70}$$

The value of ϕ_f is obtained using any of the advection schemes presented earlier keeping in mind that, in order to be able to solve for the unknown values at the main nodes, the algebraic form of the discretized equation should look like

$$a_C \phi_C + \sum_{F\sim NB(C)} a_F \phi_F = b_C \tag{12.71}$$

The difficulty lies in the instability that arises when expressing ϕ_f in terms of nodal values as described next.

12.7.1 The Difficulty with the Direct Use of Nodal Values

The difficulty that arises when explicitly expressing ϕ_f in terms of neighboring values will be explained by discretizing the convection flux using a TVD scheme. Referring to Fig. 12.16, the convective flux at a face f is written as

$$\begin{aligned}\dot{m}_f\phi_f = &\left[\phi_C + \frac{1}{2}\psi\overbrace{\left(\frac{\phi_C-\phi_U}{\phi_F-\phi_C}\right)}^{r_f^+}(\phi_F-\phi_C)\right]\|\dot{m}_f,0\| \\ &-\left[\phi_F + \frac{1}{2}\psi\overbrace{\left(\frac{\phi_F-\phi_{DD}}{\phi_C-\phi_F}\right)}^{r_f^-}(\phi_C-\phi_F)\right]\|-\dot{m}_f,0\| \\ &= \left[\phi_C + \frac{1}{2}\psi\left(r_f^+\right)(\phi_F-\phi_C)\right]\|\dot{m}_f,0\| \\ &\quad -\left[\phi_F + \frac{1}{2}\psi\left(r_f^-\right)(\phi_C-\phi_F)\right]\|-\dot{m}_f,0\|\end{aligned} \tag{12.72}$$

Substituting Eq. (12.72) into Eq. (12.70) yields

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{12.73}$$

where

$$\begin{aligned}a_F &= FluxF_f = -\|-\dot{m}_f,0\| + \frac{1}{2}\psi\left(r_f^+\right)\|\dot{m}_f,0\| + \frac{1}{2}\psi\left(r_f^-\right)\|-\dot{m}_f,0\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = -\sum_{F\sim NB(C)} a_F + \sum_{F\sim NB(C)} \dot{m}_f \\ b_C &= Q_C^\phi V_C\end{aligned} \tag{12.74}$$

To see the weakness in this formulation, Eq. (12.67) is simplified to the one dimensional problem depicted in Fig. 12.8. Assuming the flow to be in the positive direction and using the terminology displayed in the figure, Eq. (12.73) becomes

$$a_C\phi_C + a_E\phi_E + a_W\phi_W = b_C \tag{12.75}$$

where

$$\begin{aligned} a_E &= FluxF_e = \frac{1}{2}\psi\left(r_e^+\right)\dot{m}_e \\ a_W &= FluxF_w = \left[-1+\frac{1}{2}\psi\left(r_w^-\right)\right]\dot{m}_e \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = -(a_E + a_W) \\ b_C &= Q_C^{\phi} V_C \end{aligned} \tag{12.76}$$

Since $0 \le \psi(r) \le 2$, the a_E and a_W coefficients will be of opposite signs (except for the UPWIND scheme, i.e., when $\psi\left(r_f\right) = 0$), thereby violating one of the basic rules for stability and causing convergence difficulties of the iterative procedure. Following a similar approach with the NVF leads to the same shortcomings.

A remedy, which was presented in the previous chapter, is the deferred correction (DC) procedure, in which the coefficients are based on the upwind scheme, while the difference between the HR and upwind schemes is added as a source term in the algebraic equation. The DC procedure is simple to implement and can be used in structured and unstructured grid systems, however as the difference between the cell face values calculated with the upwind scheme and that calculated with the HR scheme becomes larger, the convergence rate diminishes. This effect can be easily estimated on an NVD; the difference between the UPWIND line and that of the chosen HR scheme is the normalized difference between the cell face values. The larger this difference is, the lower the convergence rate will be. This has enticed researchers to look for other techniques for implementing HR schemes that are more implicit not affecting the convergence rate. Two of these techniques are described in the next section.

12.8 The DWF and NWF Methods

Several techniques have been developed to overcome the reduction in the convergence rate associated with the use of the Deferred Correction (DC) procedure for the implementation of HR schemes. Two of these methods are presented below, namely the Downwind Weighing Factor (DWF) method of Leonard and Mokhtari [25] (implemented in OpenFOAM®) and the Normalized Weighing Factor (NWF) method of Darwish and Moukalled [29].

The implementation details for both methods are presented in the context of solving the convection equation (Eq. 12.67) over the three-dimensional unstructured grid system shown in Fig. 12.16. The discretized equation for the element of volume V_C shown in Fig. 12.16 can be written as

$$\sum_{f\sim nb(C)} \dot{m}_f \phi_f = Q_C^{\phi} V_C \tag{12.77}$$

The ϕ values at cell faces are computed using a HR scheme and the objective of the various methods is to incorporate these values in the discretized equation in the most effective manner.

12.8.1 The Downwind Weighing Factor (DWF) Method

The Downwind Weighing Factor (*DWF*) [25] defined as

$$DWF_f = \frac{\phi_f - \phi_C}{\phi_D - \phi_C} = \frac{\tilde{\phi}_f - \tilde{\phi}_C}{1 - \tilde{\phi}_C} \tag{12.78}$$

is used to rewrite the face value such that

$$\phi_f = DWF_f \phi_D + \left(1 - DWF_f\right)\phi_C = \phi_C + DWF_f(\phi_D - \phi_C) \tag{12.79}$$

thereby redistributing the HR scheme estimate ϕ_f or the normalized value $\tilde{\phi}_f$ between the Upwind and Downwind nodes. The effect is a reduced stencil for the discretized coefficients. Since the value of ϕ_f computed using a HR scheme lies between ϕ_C and ϕ_D, the value of DWF_f is always between 0 and 1, i.e., $0 \leq DWF_f \leq 1$.

Now rather than computing the DWF_f explicitly from the computed ϕ_f value, the DWF_f can be expressed directly from the functional relationships of the HR scheme. Table 12.1 presents such relationships for several HO and HR schemes on uniform grid.

Comparing the TVD formulation given by Eq. (12.30) with Eq. (12.79), it is clear that

$$DWF_f = \frac{1}{2}\psi(r_f) \tag{12.80}$$

As shown in the previous section on TVD schemes, the coefficients obtained from such implementation will not be diagonally dominant and the formulation is thus not stable for many flow configurations. As the method is used in OpenFOAM®, the reasons behind the numerical difficulties that are generally experienced by the code when solving convection dominated flow problems are now clear.

For completeness, the analysis of the implementation via the DWF is presented. Starting with Eq. (12.79), the convection flux in the general case is written in the form

$$\begin{aligned} \dot{m}_f \phi_f = {} & \left\|\dot{m}_f, 0\right\| \left[DWF_f^+ \phi_F + \left(1 - DWF_f^+\right)\phi_C\right] \\ & - \left\|-\dot{m}_f, 0\right\| \left[DWF_f^- \phi_C + \left(1 - DWF_f^-\right)\phi_F\right] \end{aligned} \tag{12.81}$$

with

$$\begin{aligned} DWF_f^+ &= \frac{\phi_f - \phi_C}{\phi_F - \phi_C} \\ DWF_f^- &= \frac{\phi_f - \phi_F}{\phi_C - \phi_F} \end{aligned} \tag{12.82}$$

Table 12.1 Functional relationships of the DWF_f for some HO and HR schemes on uniform grid

Scheme	Downwind weighing factor-NVF
Upwind	$DWF_f = 0$
SOU	$DWF_f = \dfrac{\tilde{\phi}_C}{2\left(1-\tilde{\phi}_C\right)}$
CD	$DWF_f = \dfrac{1}{2}$
FROMM	$DWF_f = \dfrac{1}{4\left(1-\tilde{\phi}_C\right)}$
QUICK	$DWF_f = \dfrac{1}{4} + \dfrac{1}{8\left(1-\tilde{\phi}_C\right)}$
Downwind	$DWF_f = 1$
MINMOD	$DWF_f = \begin{cases} \dfrac{1}{2}\dfrac{\tilde{\phi}_C}{\left(1-\tilde{\phi}_C\right)} & 0 \le \tilde{\phi}_C \le \dfrac{1}{2} \\ \dfrac{1}{2} & \dfrac{1}{2} \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$
Bounded CD	$DWF_f = \begin{cases} \dfrac{1}{2} & 0 \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$
OSHER [30]	$DWF_f = \begin{cases} \dfrac{1}{2}\dfrac{\tilde{\phi}_C}{\left(1-\tilde{\phi}_C\right)} & 0 \le \tilde{\phi}_C \le \dfrac{2}{3} \\ 1 & \dfrac{2}{3} \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$
SMART	$DWF_f = \begin{cases} \dfrac{2\tilde{\phi}_C}{1-\tilde{\phi}_C} & 0 \le \tilde{\phi}_C \le \dfrac{1}{6} \\ \dfrac{1}{4} + \dfrac{1}{8\left(1-\tilde{\phi}_C\right)} & \dfrac{1}{6} \le \tilde{\phi}_C \le \dfrac{5}{6} \\ 1 & \dfrac{5}{6} \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$
STOIC	$DWF_f = \begin{cases} \dfrac{2\tilde{\phi}_C}{1-\tilde{\phi}_C} & 0 \le \tilde{\phi}_C \le \dfrac{1}{5} \\ \dfrac{1}{2} & \dfrac{1}{5} \le \tilde{\phi}_C \le \dfrac{1}{2} \\ \dfrac{1}{4} + \dfrac{1}{8\left(1-\tilde{\phi}_C\right)} & \dfrac{1}{2} \le \tilde{\phi}_C \le \dfrac{5}{6} \\ 1 & \dfrac{5}{6} \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$
MUSCL	$DWF_f = \begin{cases} \dfrac{\tilde{\phi}_C}{1-\tilde{\phi}_C} & 0 \le \tilde{\phi}_C \le \dfrac{1}{4} \\ \dfrac{1}{4\left(1-\tilde{\phi}_C\right)} & \dfrac{1}{4} \le \tilde{\phi}_C \le \dfrac{3}{4} \\ 1 & \dfrac{3}{4} \le \tilde{\phi}_C \le 1 \\ 0 & elsewhere \end{cases}$

The fluxes in Eq. (12.77) can now be expressed as

$$\begin{aligned} FluxF_f &= \left\|\dot{m}_f, 0\right\| DWF_f^+ - \left\|-\dot{m}_f, 0\right\| \left(1 - DWF_f^-\right) \\ FluxC_f &= \left\|\dot{m}_f, 0\right\| \left(1 - DWF_f^+\right) - \left\|-\dot{m}_f, 0\right\| DWF_f^- \end{aligned} \tag{12.83}$$

and the discretized equation becomes

$$a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C \tag{12.84}$$

with

$$\begin{aligned} a_F &= FluxF_f \\ a_C &= \sum_{f \sim nb(C)} FluxC_f = -\sum_{F \sim NB(C)} a_F + \sum_{f \sim nb(C)} \dot{m}_f \\ b_C &= Q_C^{\phi} V_C \end{aligned} \tag{12.85}$$

The coefficients in Eq. (12.85) result in a highly unstable system of equations, thus requiring substantial relaxation. This can be demonstrated on a simple one-dimensional mesh (Fig. 12.8). Without loss of generality, a positive flow field is assumed, which reduces Eq. (12.84) to

$$a_C \phi_C + a_E \phi_E + a_W \phi_W = b_C \tag{12.86}$$

where

$$\begin{aligned} a_E &= FluxF_e = \dot{m}_e DWF_e^+ \\ a_W &= FluxF_w = \dot{m}_w \left(1 - DWF_w^-\right) \\ a_C &= \sum_{f \sim nb(C)} FluxC_f = -\left[\dot{m}_e DWF_e^+ + \dot{m}_w \left(1 - DWF_w^-\right)\right] + \dot{m}_e + \dot{m}_w \end{aligned} \tag{12.87}$$

The continuity constraint implies that

$$\dot{m}_e + \dot{m}_w = 0 \Rightarrow \dot{m}_w = -\dot{m}_e \tag{12.88}$$

thus the coefficients become

$$\begin{aligned} a_E &= FluxF_e = \dot{m}_e DWF_e^+ \\ a_W &= FluxF_w = -\dot{m}_e \left(1 - DWF_w^-\right) \\ a_C &= \sum_{f \sim nb(C)} FluxC_f = -\dot{m}_e \left[DWF_e^+ + DWF_w^- - 1\right] \end{aligned} \tag{12.89}$$

In the above equation the a_E and a_W coefficients have opposite signs, a violation to one of the basic coefficient rules. Furthermore for values of the *DWF* factors that are larger than 0.5 (i.e., $DWF > 0.5$), the diagonal coefficient a_C becomes negative resulting in a system not solvable by iterative means. This would occur whenever $\phi_f > 0.5(\phi_C + \phi_D)$ a situation common to all HR schemes for $\phi_C > 0.5$. In fact the DWF moves much of the HR flux influence onto the downwind value causing the above mentioned issues, a situation that resembles in effect the central difference scheme.

12.8.2 The Normalized Weighing Factor (NWF) Method

The Normalized Weighing Factor (NWF) method was developed [29] to address the shortcomings of the DWF method. It operates by linearizing the normalized interpolation profile such that

$$\tilde{\phi}_f = \ell \tilde{\phi}_C + k \tag{12.90}$$

where ℓ and k are constants that represent the slope and intercept of the linear function within any interval of ϕ_f, with the number of intervals depending on the HR scheme used. This is an exact representation for nearly all HR schemes.

For example, by equating Eq. (12.90) to the NVF form of the MINMOD scheme (Eq. 12.14), the values of ℓ and k are deduced to be

$$[\ell, k] = \begin{cases} \left[\frac{3}{2}, 0\right] & 0 < \tilde{\phi}_C < \frac{1}{2} \\ \left[\frac{1}{2}, \frac{1}{2}\right] & \frac{1}{2} \le \tilde{\phi}_C < 1 \\ [1, 0] & \textit{elsewhere} \end{cases} \tag{12.91}$$

In a second step Eq. (12.90) is rewritten as

$$\frac{\phi_f - \phi_U}{\phi_D - \phi_U} = \ell \frac{\phi_C - \phi_U}{\phi_D - \phi_U} + k \tag{12.92}$$

and transformed to

$$\phi_f = \ell(\phi_C - \phi_U) + k(\phi_D - \phi_U) + \phi_U = \ell\phi_C + k\phi_D + (1 - \ell - k)\phi_U \tag{12.93}$$

where ϕ_U, ϕ_D, and ϕ_C are the values at the U, D, and C nodes whose locations depend on the flow direction. The values of ℓ and k for a number of HO and HR schemes are listed in Table 12.2 (for unstructured and/or structured uniform grid).

Since for an unstructured grid the U location is virtual, the term involving ϕ_U is treated in a deferred correction fashion. However the value of the resulting deferred

Table 12.2 NVF values of NWF $[\ell, k]$ factors for some HO and HR schemes

Scheme	Uniform grid (NVF)
Upwind	$[\ell, k] = [1, 0]$
SOU	$[\ell, k] = \left[\frac{3}{2}, 0\right]$
CD	$[\ell, k] = \left[\frac{1}{2}, \frac{1}{2}\right]$
FROMM	$[\ell, k] = \left[1, \frac{1}{4}\right]$
QUICK	$[\ell, k] = \left[\frac{3}{4}, \frac{3}{8}\right]$
MINMOD	$[\ell, k] = \begin{cases} \left[\frac{3}{2}, 0\right] & 0 < \tilde{\phi}_C < \frac{1}{2} \\ \left[\frac{1}{2}, \frac{1}{2}\right] & \frac{1}{2} \le \tilde{\phi}_C < 1 \\ [1, 0] & elsewhere \end{cases}$
OSHER	$[\ell, k] = \begin{cases} \left[\frac{3}{2}, 0\right] & 0 < \tilde{\phi}_C < \frac{2}{3} \\ [0, 1] & \frac{2}{3} \le \tilde{\phi}_C < 1 \\ [1, 0] & elsewhere \end{cases}$
MUSCL	$[\ell, k] = \begin{cases} [2, 0] & 0 < \tilde{\phi}_C < \frac{1}{4} \\ \left[1, \frac{1}{4}\right] & \frac{1}{4} \le \tilde{\phi}_C < \frac{3}{4} \\ [0, 1] & \frac{3}{4} \le \tilde{\phi}_C < 1 \\ [1, 0] & elsewhere \end{cases}$
SMART	$[\ell, k] = \begin{cases} [4, 0] & 0 < \tilde{\phi}_C < \frac{1}{6} \\ \left[\frac{3}{4}, \frac{3}{8}\right] & \frac{1}{6} \le \tilde{\phi}_C < \frac{5}{6} \\ [0, 1] & \frac{5}{6} \le \tilde{\phi}_C < 1 \\ [1, 0] & elsewhere \end{cases}$

correction source term (i.e., $(1 - \ell - k)\phi_U \, l \ge 0, k \ge 0$) is smaller than the one that would be obtained with the standard deferred correction treatment (i.e., ϕ_U). Thus the NWF requires less under-relaxation than the standard DC method and thus allows for faster convergence.

Starting with Eq. (12.93), the convection flux in the general case (i.e., multi dimensional unstructured grid) can be written in the form

$$\begin{aligned} \dot{m}_f \phi_f = & \| \dot{m}_f, 0 \| \left[\ell_f^+ \phi_C + k_f^+ \phi_F + \left(1 - \ell_f^+ - k_f^+\right) \phi_U^+ \right] \\ & - \| -\dot{m}_f, 0 \| \left[\ell_f^- \phi_F + k_f^- \phi_C + \left(1 - \ell_f^- - k_f^-\right) \phi_U^- \right] \end{aligned} \tag{12.94}$$

and linearized to yield

$$\begin{aligned} FluxF_f &= \|\dot{m}_f, 0\|\, k_f^+ - \|-\dot{m}_f, 0\|\, \ell_f^- \\ FluxC_f &= \|\dot{m}_f, 0\|\, \ell_f^+ - \|-\dot{m}_f, 0\|\, k_f^- \\ FluxV_f &= \|\dot{m}_f, 0\| \left(1 - \ell_f^+ - k_f^+\right)\phi_U^+ - \|-\dot{m}_f, 0\| \left(1 - \ell_f^- - k_f^-\right)\phi_U^- \end{aligned} \tag{12.95}$$

Substituting Eq. (12.95) into Eq. (12.77) the algebraic equation is obtained as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{12.96}$$

where now

$$\begin{aligned} a_F &= FluxF_f = k_f^+\|\dot{m}_f, 0\| - \ell_f^-\|-\dot{m}_f, 0\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \sum_{f\sim nb(C)} \left(\ell_f^+\|\dot{m}_f, 0\| - k_f^-\|-\dot{m}_f, 0\|\right) \\ b_C &= Q_C^\phi V_C - \underbrace{\sum_{f\sim nb(C)} FluxV_f}_{b_C^{DC}} \\ &= Q_C^\phi V_C - \underbrace{\sum_{f-nb(C)} \left[\left(1 - \ell_f^+ - k_f^+\right)\phi_U^+\|\dot{m}_f, 0\| - \left(1 - \ell_f^- - k_f^-\right)\phi_U^-\|-\dot{m}_f, 0\|\right]}_{b_C^{DC}} \end{aligned} \tag{12.97}$$

The NWF was originally developed for use on structured grids with its formulation in that context allowing for a full implicit treatment of HR schemes. The full implicitness of the method on structured grid is the result of ϕ_U being an actual node in the computational domain that can be resolved in the algebraic equation, which now has a larger stencil that includes the far nodes *EE* and *WW*. For the one dimensional structured grid displayed in Fig. 12.8, the NWF form of the algebraic equation becomes

$$a_C\phi_C + \sum_{F\sim E,W,EE,WW} a_F\phi_F = b_C \tag{12.98}$$

where

$$\begin{aligned} a_E &= FluxF_e = \|\dot{m}_e, 0\|k_e^+ - \|-\dot{m}_e, 0\|\ell_e^- + \|\dot{m}_w, 0\|\left(1 - \ell_w^+ - k_w^+\right) \\ a_W &= FluxF_w = \|\dot{m}_w, 0\|k_w^+ - \|-\dot{m}_w, 0\|\ell_w^- + \|\dot{m}_e, 0\|\left(1 - \ell_e^+ - k_e^+\right) \\ a_{EE} &= FluxF_{ee} = -\|-\dot{m}_e, 0\|\left(1 - \ell_e^- - k_e^-\right) \\ a_{WW} &= FluxF_{ww} = -\|-\dot{m}_w, 0\|\left(1 - \ell_w^- - k_w^-\right) \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \|\dot{m}_e, 0\|\ell_e^+ + \|\dot{m}_w, 0\|\ell_w^+ - \|-\dot{m}_e, 0\|k_e^- - \|-\dot{m}_w, 0\|k_w^- \\ &= -(a_E + a_W + a_{EE} + a_{WW}) + (\dot{m}_e + \dot{m}_w) \end{aligned} \tag{12.99}$$

In the NWF reformulation of the HR schemes since the value of ℓ is greater than that of k (see Fig. 12.6), except in a narrow region of the NVD close to the Downwind line as explained next, the value of a_C is always positive and instability does not arise. Along the Downwind line of the NVD, where $(\ell, k) = (0, 1)$, a value of zero for the a_C coefficient is obtained. In this case (ℓ, k) is set to $\left(L, 1 - L\phi_f\right)$ where L is usually set to the value of ℓ in the previous interval of the composite scheme.

This basically allows the NWF to be much more robust than the DWF as it guarantees positive a_C coefficients.

12.8.2.1 The NWF Method in the Context of the TVD

With the exception of the MUSCL Van Leer limiter, the limiters of the TVD formulation of all HR schemes presented earlier appear as straight lines on Sweby's diagram and as such can be written as a set of linear equations of the form

$$\psi\left(r_f\right) = mr_f + n \tag{12.100}$$

where m and n are constants (slope and intercept of the linear function and depend on geometric quantities only) within any interval of $\psi\left(r_f\right)$, with the number of intervals depending on the HR TVD scheme used. For example, by equating Eq. (12.100) to the TVD form of the MINMOD scheme (Eq. 12.44), the values of m and n are deduced to be

$$\text{MINMOD} \quad [m, n] = \begin{cases} [1, 0] & 0 < r_f < 1 \\ [0, 1] & r_f \geq 1 \\ [0, 0] & r_f \leq 0 \end{cases} \tag{12.101}$$

Substituting Eq. (12.100) in Eq. (12.30), the interface value is found to be

$$\begin{aligned} \phi_f &= \phi_C + \frac{1}{2}\left(mr_f + n\right)\left(\phi_D - \phi_C\right) \\ &= \phi_C + \frac{1}{2}\left(m\frac{\phi_C - \phi_U}{\phi_D - \phi_C} + n\right)\left(\phi_D - \phi_C\right) \\ &= \left(1 + \frac{1}{2}m - \frac{1}{2}n\right)\phi_C + \frac{1}{2}n\phi_D - \frac{1}{2}m\phi_U \end{aligned} \tag{12.102}$$

where ϕ_U, ϕ_D, and ϕ_C are again the values at the U, D, and C nodes whose locations depend on the flow direction. The values of m and n for a number of HO and HR schemes are listed in Table 12.3 for uniform grids. Moreover Eq. (12.102) has the same form as Eq. (12.93) with

$$\ell = 1 + \frac{1}{2}m - \frac{1}{2}n \quad \text{and} \quad k = \frac{1}{2}n \tag{12.103}$$

Thus an approach similar to that of the NVF-NWF can be used for the implementation of the TVD-NWF.

Table 12.3 TVD values of NWF $[m,n]$ factors for some HO and HR schemes

Scheme	Uniform grid (NVF)
Upwind	$[m,n]=[0,0]$
SOU	$[m,n]=[1,0]$
CD	$[m,n]=[0,1]$
FROMM	$[m,n]=\left[\frac{1}{2},\frac{1}{2}\right]$
QUICK	$[m,n]=\left[\frac{1}{4},\frac{3}{4}\right]$
DOWNWIND	$[m,n]=[0,2]$
OSHER	$[m,n]=\begin{cases}[1,0] & 0<r_f<2\\ [0,2] & r_f\geq 2\\ [0,0] & r\leq 0\end{cases}$
MUSCL	$[m,n]=\begin{cases}[2,0] & 0<r_f<\frac{1}{3}\\ \left[\frac{1}{2},\frac{1}{2}\right] & \frac{1}{3}\leq r_f<3\\ [0,2] & r_f\geq 3\\ [0,0] & r_f\leq 0\end{cases}$
SUPERBEE	$[m,n]=\begin{cases}[2,0] & 0<r_f<\frac{1}{2}\\ [0,1] & \frac{1}{2}\leq r_f<1\\ [1,0] & 1\leq r_f<2\\ [0,2] & r_f\geq 2\\ [0,0] & r_f\leq 0\end{cases}$

12.9 Boundary Conditions

The Boundary conditions for the convection term are generally much simpler than for the diffusion term. The particulars of the implementation of the following boundary conditions: "Inlet", "Outlet", "Wall", and "Symmetry" are now detailed.

Typical boundary elements are shown in Fig. 12.17. A boundary cell, as mentioned earlier in this book, is one that has one or more faces on the boundary. Discrete values of ϕ are stored both at centroids of boundary cells and of boundary faces.

Let C denotes the centroid of the boundary element with one boundary face of centroid b and of surface vector $\mathbf{S}_b$ pointing outward (Fig. 12.17). As before, the discretization process over cell C of a pure convection problem in a multidimensional domain yields

$$\sum_{f\sim nb(C)}\left(\mathbf{J}^{\phi,C}\cdot\mathbf{S}\right)_f=0 \tag{12.104}$$

The fluxes on the interior faces are discretized as before. Independent of the boundary condition type, the boundary flux $\mathbf{J}_b^{\phi,C}$ may be written using the boundary face centroid value as

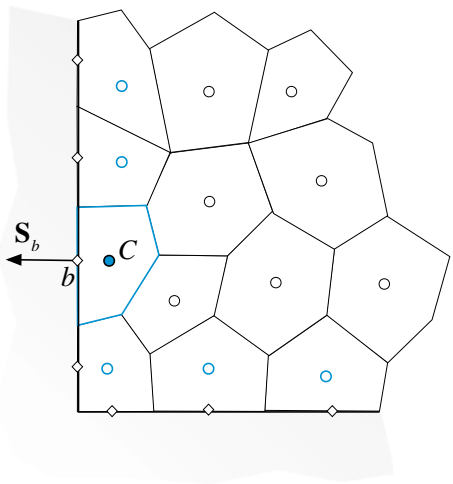


Fig. 12.17 Boundary elements with one or two boundary faces

$$\mathbf{J}_b^{\phi,C} = (\rho \mathbf{v}\phi)_b \tag{12.105}$$

such that

$$\mathbf{J}_b^{\phi,C} \cdot \mathbf{S}_b = (\rho \mathbf{v}\phi)_b \cdot \mathbf{S}_b = \dot{m}_b \phi_b \tag{12.106}$$

Thus the discretized equation of the boundary cell is expressed as

$$\sum_{f \sim nb(C)} (\rho \mathbf{v}\phi \cdot \mathbf{S})_f + (\rho \mathbf{v}\phi \cdot \mathbf{S})_b = 0 \tag{12.107}$$

where subscript f refers to interior faces and subscript b to the boundary face. The specification of boundary conditions involves either specifying the unknown boundary value ϕ_b, or alternatively, the boundary flux $\mathbf{J}_b^{\phi,C}$. Using Eq. (12.107), the discretized equations at a boundary element for the different boundary condition types of convection problems are derived next.

12.9.1 Inlet Boundary Condition

At inlet to a domain (Fig. 12.18), the value of ϕ is usually specified. Since the velocity field is assumed to be known, then the convective flux at inlet is also known. Therefore the boundary flux is moved to the right hand side and treated as a source term. With this modification Eq. (12.107) becomes

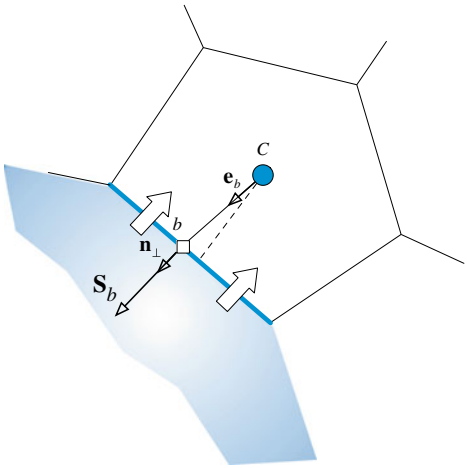


Fig. 12.18 Inlet boundary condition for the convection flux

$$\sum_{f\sim nb(C)} (\rho \mathbf{v}\cdot\mathbf{S})_f \phi_f = -(\rho \mathbf{v}\cdot\mathbf{S})_b \phi_b = -\dot{m}_b \phi_b \tag{12.108}$$

If a HR scheme is used to discretize the convection flux at interior faces and is implemented via a deferred correction approach, then the modified algebraic equation for the boundary element can be written as

$$a_C \phi_C + \sum_{F\sim NB(C)} a_F \phi_F = b_C \tag{12.109}$$

where

$$\begin{aligned} a_F &= FluxF_f = -\|-\dot{m}_f, 0\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \sum_{f\sim nb(C)} \|\dot{m}_f, 0\| \\ &= -\sum_{F\sim NB(C)} a_F + \sum_{f\sim nb(C)} \dot{m}_f \\ b_C &= -\sum_{f\sim nb(C)} FluxV_f = -\dot{m}_b \phi_b - \underbrace{\sum_{f\sim nb(C)} \dot{m}_f \left(\phi_f^{HR} - \phi_f^U\right)}_{b_c^{DC}} \end{aligned} \tag{12.110}$$

where F refers to interior neighboring nodes of the C grid point and f refers to interior faces of the boundary element.

12.9.2 Outlet Boundary Condition

At outlet from the domain (Fig. 12.19) no information downstream of the boundary grid point is available. However, being a directional phenomenon, the value of ϕ at the boundary is highly dependent on upstream locations. In fact, the upwind and SOU schemes, for example, do not require any information at outlet since its value can be expressed as a function of values at upstream nodes. The treatment that has proven to be very effective at an outlet boundary condition is to assume the ϕ profile to be fully developed, which is equivalent to assuming that the normal gradient to the face is zero [i.e., $(\nabla\phi \cdot \mathbf{n})_b = (\partial\phi/\partial n)_b = 0$]. The usual practice at an outlet (Fig. 12.19) is to apply the upwind scheme ($\phi_b = \phi_C$), which automatically results in a zero normal gradient.

Discretizing the convection flux at interior faces using a HR scheme implemented through a deferred correction approach, the modified algebraic equation for the boundary element can be written as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{12.111}$$

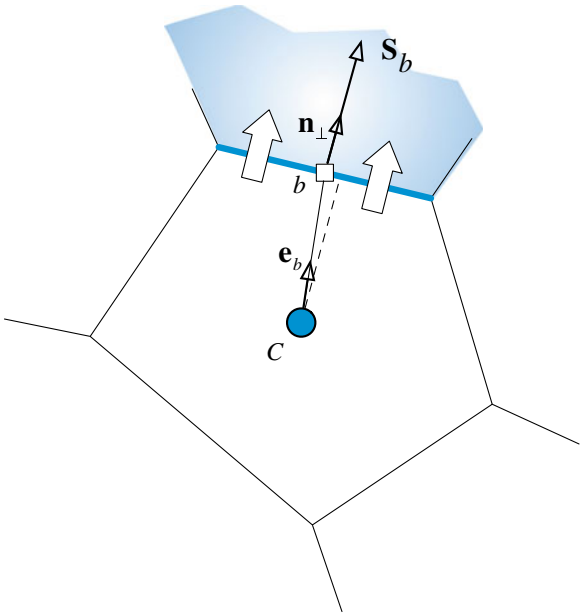


Fig. 12.19 Outlet boundary condition for the convection flux

where

$$\begin{aligned} a_F &= FluxF_f = -\,\|-\dot{m}_f, 0\,\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \sum_{f\sim nb(C)} \|\dot{m}_f, 0\,\| \\ &= -\sum_{F\sim NB(C)} a_F + \sum_{f\sim nb(C)} \left(\dot{m}_f + \dot{m}_b\right) \\ b_C &= -\sum_{f\sim nb(C)} FluxV_f = \underbrace{-\sum_{f\sim nb(C)} \dot{m}_f \left(\phi_f^{HR} - \phi_f^{U}\right)}_{b_c^{DC}} \end{aligned} \tag{12.112}$$

where f refers to the interior faces of the boundary element, and C and F refer to owner and neighbor, respectively.

12.9.3 Wall Boundary Condition

As shown in Fig. 12.20, the normal velocity at a wall is zero. As such the convection flux is zero and does not appear in the algebraic equation.

Again adopting a HR scheme with a deferred correction approach, the modified algebraic equation for the boundary element can be written as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{12.113}$$

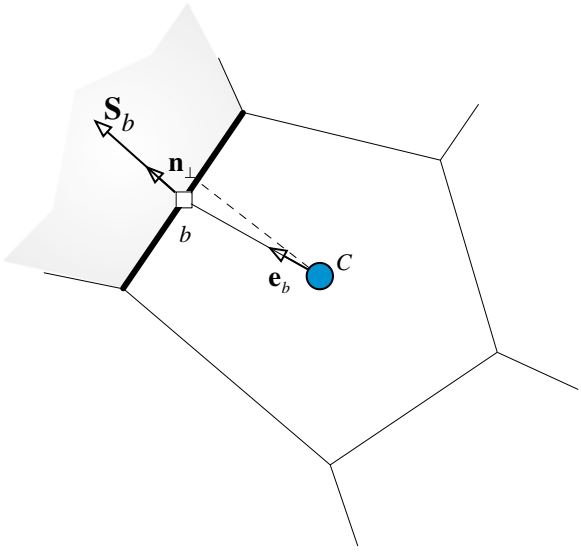


Fig. 12.20 Wall boundary condition for the convection flux

where

$$\begin{aligned} a_F &= FluxF_f = -\|-\dot{m}_f, 0\| \\ a_C &= \sum_{f\sim nb(C)} FluxC_f = \sum_{f\sim nb(C)} \|\dot{m}_f, 0\| \\ &= -\sum_{F\sim NB(C)} a_F + \sum_{f\sim nb(C)} \dot{m}_f \\ b_C &= -\sum_{f\sim nb(C)} FluxV_f = \underbrace{-\sum_{f\sim nb(C)} \dot{m}_f \left(\phi_f^{HR} - \phi_f^U\right)}_{b_c^{DC}} \end{aligned} \tag{12.114}$$

Again f refers to the interior faces of the boundary element and C and F refer to the owner and neighbor nodes, respectively.

12.9.4 Symmetry Boundary Condition

No flow crosses a symmetry boundary. Therefore it is treated in a similar fashion to a wall boundary condition with the convection flux normal to a symmetry boundary set to zero.

12.10 Computational Pointers

12.10.1 uFVM

Similar to HO schemes discussed in Chap. 11, HR schemes are implemented in uFVM using the deferred correction method. The main difference in the implementation stems from the use of the NVF or TVD relations rather than the calculation of the face value using directly the gradient. The implementation of the STOIC HR [31] scheme using the NVF formulation is shown in Listing 12.1 (cfdAssembleConvectionTermDCSTOIC).

It starts with the retrieval of the needed fields, followed by setting the upwind and downing indices for all interior faces. Then for each face, ϕ_C and ϕ_D are identified, ϕ_U is constructed, and $\tilde{\phi}_C$ is computed and used to calculate the face value from the NVF relationship of the adopted HR scheme. Finally ϕ_f is reconstructed from $\tilde{\phi}_f$ and used in the deferred correction method.

```
theFluidTag = cfdGetFluidTag(theEquationName);
theMdotName = ['Mdot' theFluidTag];
theMdotField = cfdGetMeshField(theMdotName,'Faces');
mdot_f = theMdotField.phi(iFaces);

iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';
pos = zeros(size(mdot_f));
pos(mdot_f>0)=1;

% find indices of U and D cells
iUpwind   =  pos   .*iOwners + (1-pos).*iNeighbours;
iDownwind = (1-pos).*iOwners +  pos   .*iNeighbours;

% find phi_C, phi_D and calculate phi_U
phi_C = phi(iUpwind,iComponent);
phi_D = phi(iDownwind,iComponent);
rCD                    =       [theMesh.elements(iDownwind).centroid]'-
[theMesh.elements(iUpwind).centroid]';

phi_U = phi_D - 2*dot(phiGrad(iUpwind,:,iComponent)',rCD')';

SMALL= 1e-6;
% calculate phi_tildaC
nominator = phi_C-phi_U;
denominator = phi_D-phi_U;
divideLoc = find(~((denominator<SMALL) & (denominator>-SMALL)));
phi_tildaC = ones(size(phi_C));
phi_tildaC(divideLoc) = nominator(divideLoc)./denominator(divideLoc);
```

Listing 12.1 Implementation of the STOIC HR scheme

```
% get phi_tildaf from STOIC function
phi_tildaf = zeros(size(phi_tildaC));
% lower UPWIND section
phi_tildaf       =       phi_tildaf       +       (phi_tildaC       <=
0)                     .*(phi_tildaC);
% intermediate section
phi_tildaf  =  phi_tildaf  +  (phi_tildaC  >  0)       .*(phi_tildaC  <
0.2).*(3*phi_tildaC);
% CDS section
phi_tildaf   =   phi_tildaf   +   (phi_tildaC   >=   0.2).*(phi_tildaC   <
0.5).*(0.5*phi_tildaC + 0.5);
% SMART section
phi_tildaf   =   phi_tildaf   +   (phi_tildaC   >=   0.5).*(phi_tildaC   <
5/6).*(0.75*phi_tildaC + 3/8);
% DOWNWIND section
phi_tildaf   =   phi_tildaf   +   (phi_tildaC   >=   5/6).*(phi_tildaC   <
1)  .*(ones(size(phi_tildaC)));
% upper UPWIND section
phi_tildaf       =       phi_tildaf       +       (phi_tildaC       >=
1)                     .*(phi_tildaC);


% calculate phi_f
phi_f = phi_tildaf.*(phi_D-phi_U) + phi_U;


% calculate correction
corr = mdot_f .* (phi_f - phi_C);


% apply deferred correction
theFluxes.FLUXTf(iFaces) = theFluxes.FLUXTf(iFaces) + corr;
```

Listing 12.1 (continued)

12.10.2 OpenFOAM®

As discussed in Chap. 11 the discretization of the convection term in OpenFOAM® [32] is accomplished through the base class **surfaceInterpolationScheme**. This class is inherited by any face interpolation algorithm. High resolution schemes, as discussed in this chapter, are special face interpolation algorithms. Thus, as displayed in Fig. 12.21, OpenFOAM® groups all TVD schemes in a base class, derived from the **surfaceInterpolationScheme** class, denoted by the **limitedSurfaceInterpolationScheme** class.

As shown in Listing 12.2, this class consists of the following three main functions: the virtual **weights** function representing the main virtual function of the **surfaceInterpolationScheme** class, a local **weights** function defined with three arguments, and a new virtual base function called **limiter**.

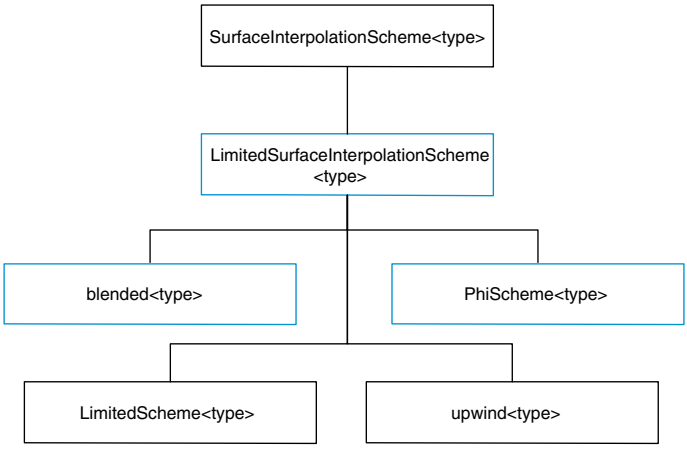


Fig. 12.21 UML showing the base class that groups all TVD schemes

```
template<class Type>
class limitedSurfaceInterpolationScheme
:
    public surfaceInterpolationScheme<Type>
{
…
// Member Functions
 //- Returns the interpolation weighting factors
 virtual tmp<surfaceScalarField> limiter
 (
     const GeometricField<Type, fvPatchField, volMesh>&
 ) const = 0;
 //- Returns the interpolation weighting factors for the given
 //  field, by limiting the given weights with the given limiter
 tmp<surfaceScalarField> weights
 (
     const GeometricField<Type, fvPatchField, volMesh>&,
     const surfaceScalarField& CDweights,
     tmp<surfaceScalarField> tLimiter
 ) const;
 //- Return the interpolation weighting factors for the given field
 virtual tmp<surfaceScalarField> weights
 (
     const GeometricField<Type, fvPatchField, volMesh>&
 ) const;
```

Listing 12.2 The **limitedSurfaceInterpolationScheme** class showing its three main functions

The script of the virtual **weights** function, shown in Listing 12.3, is given by

```
template<class Type>
tmp<surfaceScalarField>
limitedSurfaceInterpolationScheme<Type>::weights
(
    const GeometricField<Type, fvPatchField, volMesh>& phi
```

Listing 12.3 Script showing the implementation of the virtual **weights** function

```
        ) const
        {
            return this->weights
            (
                phi,
                this->mesh().surfaceInterpolation::weights(),
                this->limiter(phi)
            );
        }
```

Listing 12.3 (continued)

It is clear that executing the script instantiates the additional local **weights** function. The three main arguments of the local **weights** function are the flux **phi**, the linear interpolation **weights** (representing the central differencing weights), and the returning object of the virtual base class **limiter**. These are clearly shown in Listing 12.4 where the local **weights** function is defined.

```
template<class Type>
tmp<surfaceScalarField>
limitedSurfaceInterpolationScheme<Type>::weights
(
    const GeometricField<Type, fvPatchField, volMesh>& phi,
    const surfaceScalarField& CDweights,
    tmp<surfaceScalarField> tLimiter
) const
{
    surfaceScalarField& Weights = tLimiter();
    scalarField& pWeights = Weights.internalField();
    forAll(pWeights, face)
    {
        pWeights[face] =
            pWeights[face]*CDweights[face]
          + (1.0 - pWeights[face])*pos(faceFlux_[face]);
    }
```

Listing 12.4 Script showing the implementation of the local **weights** function

The calculation of the interpolation weights according to the TVD formulation implemented in Listing 12.4 may seem, from a first look, a bit unclear. Confusion may be eliminated by considering the case of a positive flux, for example, for which the weight ϖ is calculated as

$$\varpi = \varpi_{CD}\psi + (1 - \psi) \tag{12.115}$$

For a uniform grid $\varpi_{CD} = 1/2$ yielding

$$\varpi = \frac{\psi}{2} + (1 - \psi) = 1 - \frac{\psi}{2} \tag{12.116}$$

Recalling Eq. (11.164) and substituting the weight yields

$$\phi_f = \phi_N + \varpi(\phi_O - \phi_N) = \phi_N + \left(1 - \frac{\psi}{2}\right)(\phi_O - \phi_N) = \phi_O + \frac{\psi}{2}(\phi_N - \phi_O) \tag{12.117}$$

which is Eq. (12.30) in the TVD formulation.

So far the implementation of HR schemes in OpenFOAM® following the TVD formulation has been discussed along with its integration in the standard convection discretization procedure by properly defining the interpolation weights. Due to the many TVD schemes that can be used, OpenFOAM® has introduced a base virtual function denoted by **limiter** for the implementation of these schemes. The last step of the computational pointer is to describe how this class is defined and which classes, this **limiter** base class, engage.

All TVD limiters are organized through a class named **LimitedScheme** that inherits and defines the **limiter** function of the **limitedSurfaceInterpolation Scheme** class. The definition of this class is based on nested template classes definition in which the derived class is described with a template argument, as shown in Listing 12.5.

```
template<class Type, class Limiter, template<class> class LimitFunc>
class LimitedScheme
:
    public limitedSurfaceInterpolationScheme<Type>,
    public Limiter
{
…
```

Listing 12.5 The **LimitedScheme** class with the **Limiter** template

Here **Limiter** is not a function but just a template definition. Additionally the virtual **limiter** class is now specialized (Listing 12.6).

```
//- Return the interpolation weighting factors
 virtual tmp<surfaceScalarField> limiter
 (
     const GeometricField<Type, fvPatchField, volMesh>&
 ) const;
```

Listing 12.6 The virtual **limiter** class

As depicted in Listing 12.7 the definition of the **limiter** function is practically linked to an auxiliary function named **calcLimiter**.

```
 template<class Type, class Limiter, template<class> class LimitFunc>
Foam::tmp<Foam::surfaceScalarField>
Foam::LimitedScheme<Type, Limiter, LimitFunc>::limiter
(
    const GeometricField<Type, fvPatchField, volMesh>& phi
) const
{
…
       tmp<surfaceScalarField> tlimiterField
        (
            new surfaceScalarField
            (
                IOobject
                (
                    limiterFieldName,
                    mesh.time().timeName(),
                    mesh
                ),
                mesh,
                dimless
            )
        );
```

Listing 12.7 The script used to call the **calcLimiter** function

```
        calcLimiter(phi, tlimiterField());

        return tlimiterField;
…
```

Listing 12.7 (continued)

The core of the **calcLimiter** function is to evaluate the TVD limiter and to store it in the **tlimiterField**, as shown in Listings 12.7 and 12.8.

```
template<class Type, class Limiter, template<class> class LimitFunc>
void Foam::LimitedScheme<Type, Limiter, LimitFunc>::calcLimiter
(
    const GeometricField<Type, fvPatchField, volMesh>& phi,
    surfaceScalarField& limiterField
) const
{
    const fvMesh& mesh = this->mesh();


         tmp<GeometricField<typename  Limiter::phiType,  fvPatchField,
volMesh> >
        tlPhi = LimitFunc<Type>()(phi);


       const  GeometricField<typename  Limiter::phiType,  fvPatchField,
volMesh>&
        lPhi = tlPhi();


      tmp<GeometricField<typename  Limiter::gradPhiType,  fvPatchField,
volMesh> >
        tgradc(fvc::grad(lPhi));
     const GeometricField<typename Limiter::gradPhiType, fvPatchField,
volMesh>&
        gradc = tgradc();


                    const      surfaceScalarField&      CDweights      =
mesh.surfaceInterpolation::weights();
```

Listing 12.8 Script used with the **calcLimiter** function

```
    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();

    const vectorField& C = mesh.C();

    scalarField& pLim = limiterField.internalField();

    forAll(pLim, face)
    {
       label own = owner[face];
       label nei = neighbour[face];
       pLim[face] = Limiter::limiter
       (
           CDweights[face],
           this->faceFlux_[face],
           lPhi[own],
           lPhi[nei],
           gradc[own],
           gradc[nei],
           C[nei] - C[own]
       );
    }
```

Listing 12.8 (continued)

In the **calcLimiter** function the following steps are performed:

- Storing a copy of the field to be interpolated in *lPhi*
- Evaluating the gradient of the lPhi field using fvc::grad(lPhi)
- Collecting the central differencing weights
- Collecting the cell centers
- Evaluating the limiter by calling the nested template class: **pLim [face] = Limiter::limiter**.

As an example of a Limiter::limiter function, the TVD formulation of the SUPERBEE given in Eq. (12.44) is considered. The OpenFOAM® definition can be found in "$FOAM_SRC/finiteVolume/interpolation/surfaceInterpolation/limited Schemes/Super Bee/SuperBee.H" file. In this case the script of the **limiter** function is given in Listing (12.9) as

```
scalar limiter
(
    const scalar cdWeight,
    const scalar faceFlux,
    const typename LimiterFunc::phiType& phiP,
    const typename LimiterFunc::phiType& phiN,
    const typename LimiterFunc::gradPhiType& gradcP,
    const typename LimiterFunc::gradPhiType& gradcN,
    const vector& d
) const
{
    scalar r = LimiterFunc::r
    (
        faceFlux, phiP, phiN, gradcP, gradcN, d
    );

    return max(max(min(2*r, 1), min(r, 2)), 0);
}
```

Listing 12.9 The limiter function of the SuperBee scheme

where the arguments are as stated before including the gradients, the central differencing weights, etc., while the returned value follows exactly Eq. (12.44).

The r definition follows the same nested template class and the function itself is defined in the file "$FOAM_SRC/src/finiteVolume/interpolation/surface Interpolation/ limitedSchemes/LimitedScheme/NVDTVD.H" according to Eq. (12.66). The implementation details are given in Listing 12.10.

```
scalar r
(
   const scalar faceFlux,
   const scalar phiP,
   const scalar phiN,
   const vector& gradcP,
   const vector& gradcN,
   const vector& d
) const
{
```

Listing 12.10 Script used to calculate r

```
    scalar gradf = phiN - phiP;

    scalar gradcf;

    if (faceFlux > 0)
    {
        gradcf = d & gradcP;
    }
    else
    {
        gradcf = d & gradcN;
    }
...
    {
        return 2*(gradcf/gradf) - 1;
    }
}
```

Listing 12.10 (continued)

12.11 Closure

The chapter dealt with the bounding of HO convection schemes. This was accomplished by enforcing a convection boundedness criterion (CBC). The resulting HO bounded schemes were denoted by HR schemes. The Normalized Variable Formulation (NVF) and Total Variation Diminishing (TVD) approaches were introduced as frameworks for the development of HR schemes. Two techniques for the implementation of HO and HR schemes in structured and unstructured grids were introduced, namely the Downwind Weighing Factor (DWF) method and the Normalized Weighing Factor (NWF) method. The next chapter is devoted to the discretization of the unsteady term.

12.12 Exercises

Exercise 1

a. Starting with the NVF form of the SMART scheme derive its equivalent TVD form.
b. Starting with the TVD form of the OSHER scheme derive its equivalent NVF form.

Exercise 2

For non-uniform grids the equations for the various schemes become geometry dependent. This is also true for the point Q through which schemes have to pass to be second order accurate.

Find the coordinates of Q in the general case of a non-uniform grid.

Hint: define a normalized space variable as [33]

$$\tilde{x} = \frac{x - x_U}{x_D - x_U}$$

Exercise 3

Derive the DWF_f and NWF_f relationships of the OSHER and SMART schemes.

Exercise 4

For the one dimensional uniform mesh shown in Fig. 12.22, use the NVF-SMART, NVF-OSHER, QUICK, and SOU schemes to compute ϕ_f for the following situations:

a. $\phi_U = 30,\ \phi_C = 20,\ \phi_D = 10$
b. $\phi_U = 10,\ \phi_C = 5,\ \ \phi_D = 15$
c. $\phi_U = 30,\ \phi_C = 10,\ \phi_D = 5$
d. $\phi_U = 30,\ \phi_C = 25,\ \phi_D = 5$

Exercise 5

For the one dimensional mesh shown in Fig. 12.22, use the TVD-VanLeer and TVD-MINMOD schemes to compute ϕ_f for the following situations:

a. $\phi_U = 30,\ \phi_C = 20,\ \phi_D = 10$
b. $\phi_U = 10,\ \phi_C = 5,\ \ \phi_D = 15$
c. $\phi_U = 30,\ \phi_C = 10,\ \phi_D = 5$
d. $\phi_U = 30,\ \phi_C = 25,\ \phi_D = 5$

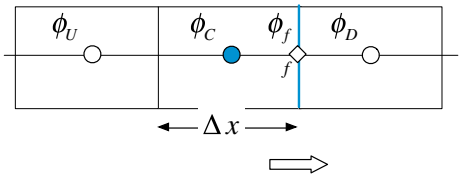


Fig. 12.22 A one dimensional uniform grid

Exercise 6

Consider the steady transport of a scalar ϕ in the domain shown in Fig. 12.23. The governing conservation equation is given by

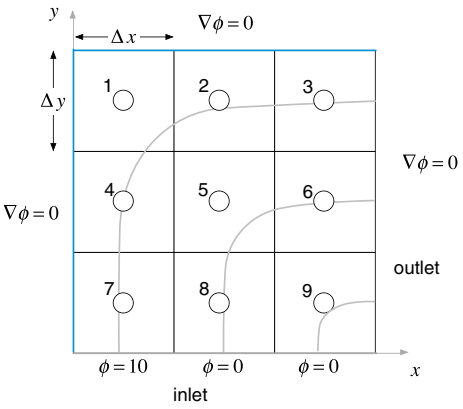


Fig. 12.23 Convection of a two dimensional scalar field

$$\nabla \cdot (\rho \mathbf{v} \phi) = 0$$

where $\rho = 1$, $\mathbf{v} = 2yx^2\mathbf{i} - 2xy^2\mathbf{j}$, and $\Delta x = \Delta y = 1/3$.

a. Using the NVF-SMART scheme, applied via a deferred correction approach, discretize the equation over the computational domain and find the value of ϕ at each element centroid.
b. Using the TVD-SMART scheme, applied via a deferred correction approach, discretize the equation over the computational domain and find the value of ϕ at each element centroid.
c. Using the NVF-SUPERBEE applied via the NVF-NWF method setup the system of equations over the domain.
d. Using the TVD-MUSCL applied via the TVD-DWF method setup the system of equations over the domain.

Exercise 7

The advection of a step profile in an oblique velocity field, $\mathbf{v} = 2\mathbf{i} + \mathbf{j}$, shown in Fig. 12.24 is governed by

$$\nabla \cdot (\rho \mathbf{v} \phi) = 0$$

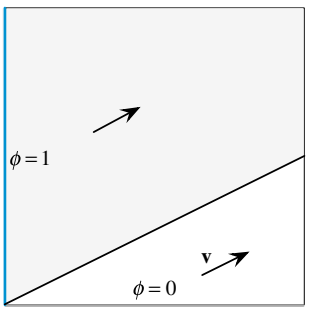


Fig. 12.24 Advection of a step profile in an oblique velocity field

For different grid sizes, setup the problem and solve it in OpenFOAM® and uFVM using the following HR advection schemes assuming unit dimensions in x and y directions, and compare results with the exact solution $(\rho = 1)$:

a. MINMOD
b. OSHER
c. SMART

Exercise 8
The Smith-Hutton test governed by

$$\nabla \cdot (\rho \mathbf{v} \phi) = 0$$

and illustrated in Fig. 12.25, involves the pure advection of a step profile in a rotational velocity field described as

$$\mathbf{v} = 2y\left(1 - x^2\right)\mathbf{i} - 2x\left(1 - y^2\right)\mathbf{j}$$

For different grid sizes, solve the test in OpenFOAM® and uFVM using the following advection schemes, and compare results with the exact solution $(\rho = 1)$:

a. Bounded CD
b. MUSCL
c. SUPERB

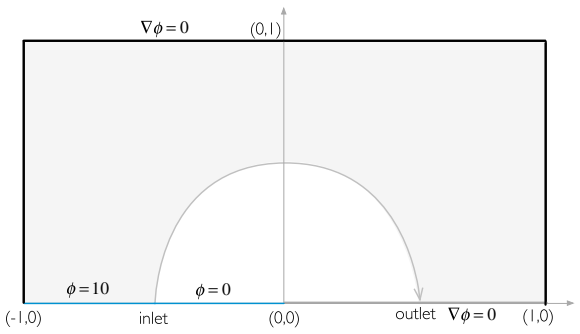


Fig. 12.25 Advection of a step profile in a two dimensional rotational velocity field

Exercise 9

a. Using Doxygen [34] list all the derived classes of the class limitedSurfaceInterpolationScheme<Type>.
b. Verify the correct implementation of the derived upwind<Type> class: check the weights function.
c. Find all OpenFOAM® limiter classes listed in Eq. (12.44) (vanLeerLimiter). Compare the formula with the OpenFOAM® implementation.

References

1. Leonard BP (1988) Universal limiter for transient interpolation modeling of the advective transport equations: the ULTIMATE conservative difference. NASA TR-100916, ICOMP-88-11
2. Leonard BP (1987) SHARP simulation of discontinuities in highly convective steady flow. NASA TM-100240
3. Leonard BP, Lock AP, MacVean MK (1995) Extended numerical integration for genuinely multidimensional advective transport insuring conservation. In: Proceedings of the ninth international conference numerical methods in laminar and turbulent flows, vol 9(1), pp 1–12
4. Gaskell PH, Lau AKC (1988) Curvature compensated convective transport: SMART, a new boundedness preserving transport algorithm. Int J Numer Meth Fluids 8(6):617–641
5. Godunov S, Ryabenki V (1963) Spectral stability criteria for boundary value problems for non self-adjoint difference equations. Uspekhi Mat Nauk 18:1–12
6. Fromm EA (1968) A method for reducing dispersion in convective difference schemes. J Comput Phys 3:176–189
7. Sheu TWH, Wang SK, Tsai SF (1998) Development of a high-resolution scheme for a multi-dimensional advection-diffusion equation. J Comput Phys 144(1):1–16
8. Leonard BP, Niknaffs HS (1991) Sharp monotonic resolution of discontinuities without clipping of narrow extrema. Comput Fluids 19:141–154
9. Van Leer B (1977) Towards the ultimate conservation difference scheme V. A second order sequel to Godunov's method. J Comput Phys 23:101–136
10. Zhu J, Rodi W (1991) A low-dispersion and bounded convection scheme. Comput Methods Appl Mech Eng 92:87–96
11. Yee HC, Warming RF, Harten A (1983) Implicit total variation diminishing (TVD) schemes for steady-state calculations. NASA Technical Memorandum 84832
12. Sweby PK (1984) High resolution schemes using flux limiters for hyperbolic conservation laws. SIAM J Numer Anal 21(5):995–1011
13. Chakravarth SR (1987) Development of upwind schemes for the Euler equations. NASA Contractor Report 4043
14. VanderHeyden WB, Kashiwa BA (1998) Compatible fluxes for Van Leer advection. J Comput Phys 146:1–28
15. Boris JP, Book DL (1973) Flux-corrected transport: I. SHASTA, a fluid transport algorithm that works. J Comput Phys 11:38–69
16. Kuzmin D, Turek S (2002) Flux correction tools for finite elements. J Comput Phys 175:525–558
17. Zalesak S (1979) Fully multidimensional flux corrected algorithm for fluid. J Comput Phys 31:335–362
18. Leonard BP (1988) Simple high-accuracy resolution program for convective modelling of discontinuities. Int J Numer Meth Eng 8:1291–1318
19. Spekreise S (1987) Multigrid solution of monotone second order discretizations of hyperbolic conservation laws. Math Comput 49(179):135–155
20. Barth T, Jespersen DC (1989) The design and application of upwind schemes on unstructured meshes. AIAA paper 89-0366
21. Leonard BP (1991) The ULTIMATE conservative difference scheme applied to unsteady one-dimensional advection. Comput Methods Appl Mech Eng 88(1):17–74
22. Dritschel DG, Fontane J (2010) The combined Lagrangian advection method. J Comput Phys 229:5408–5417
23. Leonard BP, MacVean MK, Lock AP (1993) Positivity-preserving numerical schemes for multidimensional advection. Technical Memorandum TM-106055 ICOMP-93-05, NASA
24. Leonard BP (1988) Simple high-accuracy resolution program for convective modelling of discontinuities. Int J Numer Meth Fluids 8:1291–1318

25. Leonard BP, Mokhtari S (1990) Beyond first-order upwinding: the ULTRA-SHARP alternative for non-oscillatory steady state simulation of convection. Int J Numer Methods Eng 30:729–766
26. Harten A (1983) High resolution schemes for hyperbolic conservation laws. J Comput Phys 49:357–393
27. Sweby PK (1984) High resolution schemes using flux limiters for hyperbolic conservation laws. SIAM J Numer Anal 21(5):995–1011
28. Van Leer B (1974) Towards the ultimate conservative difference scheme, 11. Monotonicity and conservation combined in a second order scheme. J Comput Phys 14:361–370
29. Darwish M, Moukalled F (1996) The normalized weighting factor method: a novel technique for accelerating the convergence of high-resolution convective schemes. Numer Heat Transf, Part B: Fundam 30:217–237
30. Osher S (1984) Shock modeling in transonic and supersonic flow. In: Habashi WG (ed) Recent advances in numerical methods in fluids, 4. Advances in computational transonics. Pineridge Press, Swansea
31. Darwish MS (1993) A new high-resolution scheme based on the normalized variable formulation. Numer Heat Transf, Part B: Fundam 24(3):353–371
32. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org
33. Darwish M, Moukalled F (1994) Normalized variable and space formulation methodology for high-resolution schemes. Numer Heat Transf, Part B 26(1):79–96
34. OpenFOAM Doxygen, 2015 Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 13
Temporal Discretization: The Transient Term

Abstract The discussions in previous chapters assumed steady state conditions, which did not require the discretization of the transient term. Accounting for transient phenomena adds a new dimension to the problem. However since transient variations are parabolic by nature, there is no need to define a field in the time dimension, as is the case for the spatial domain. In general only one or two additional variable fields, or time levels, are stored (depending on the numerical order of the selected scheme). Another difference with steady state configurations is that transient systems are modeled using a time stepping procedure. Starting with an initial condition at time $t = t_0$, the solution algorithm marches forward and finds a solution at time $t_1 = t_0 + \Delta t_1$. The solution found is the initial condition for the next time step and is used to obtain the solution at time $t_2 = t_1 + \Delta t_2$. The process is repeated until the required time is reached. The focus of this chapter is on techniques used for the discretization of the transient term. Two approaches for developing transient schemes are presented. In the first one Taylor expansions are used to express the transient term with the aid of nodal values. This is in effect a finite difference discretization. In the second approach the finite volume method is used on a pseudo time element in a similar fashion to what was done to the convection term. Several transient schemes are presented and their characteristics discussed.

13.1 Introduction

For transient simulations, the governing equations are discretized in both space and time. While the spatial discretization is performed in the spatial domain as was done for the steady-state case, the temporal discretization involves setting up a time coordinate along which the derivative (for the finite difference method) or the integral (for the finite volume method) of the transient term is evaluated (Fig. 13.1).

In general, the expression for the transient behavior, or time evolution, of a variable ϕ is governed by an equation of the form

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_13

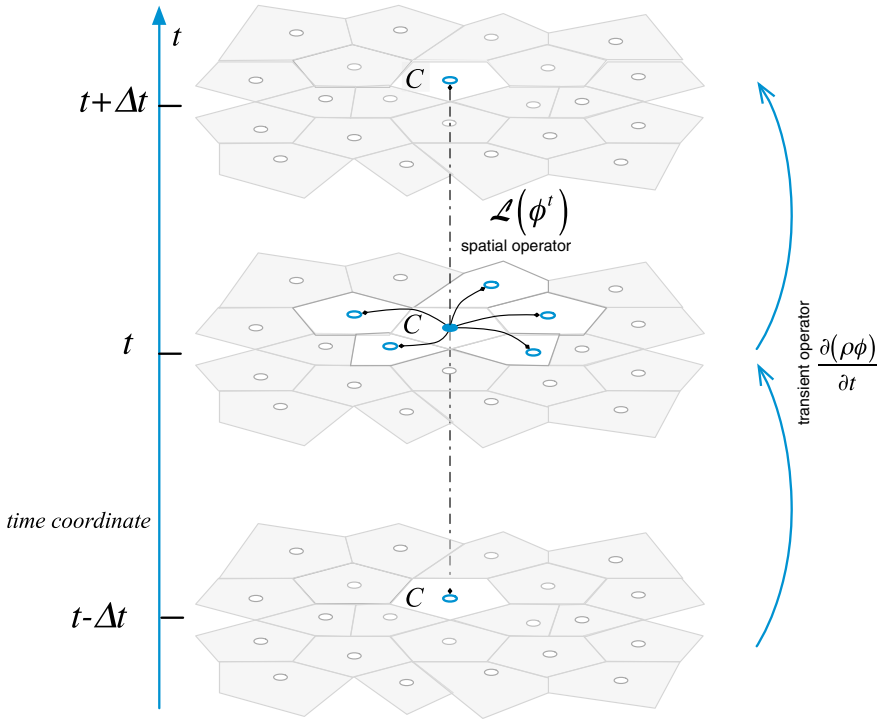


Fig. 13.1 Time coordinate, transient, and spatial operators

$$\frac{\partial(\rho\phi)}{\partial t} + \mathcal{L}(\phi) = 0 \tag{13.1}$$

where the function $\mathcal{L}(\phi)$ is a spatial operator that includes all non-transient terms (convection, advection, sources, etc.) and $\partial(\rho\phi)/\partial t$ is the transient operator, both displayed in Fig. 13.1.

Integrating Eq. (13.1) over an element C (Fig. 13.2) yields

$$\int_{V_C} \frac{\partial(\rho\phi)}{\partial t} dV + \int_{V_C} \mathcal{L}(\phi) dV = 0 \tag{13.2}$$

which, after a spatial discretization about the volume centroid, becomes

$$\frac{\partial(\rho_C \phi_C)}{\partial t} V_C + L\left(\phi_C^t\right) = 0 \tag{13.3}$$

where V_C is the volume of the discretization element and $L\left(\phi_C^t\right)$ is the spatial discretization operator expressed at some reference time t, which can be written in algebraic form as

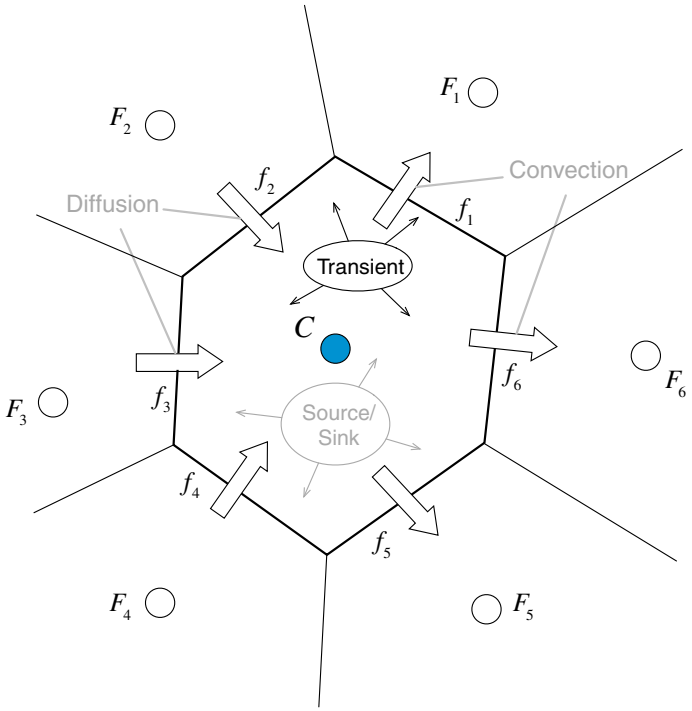


Fig. 13.2 Spatial element

$$L\left(\phi_C^t\right) = a_C\phi_C^t + \sum_{F\sim NB(C)} a_F\phi_F^t - b_C \tag{13.4}$$

In Eq. (13.3) the steady state discrete equation is recovered when $t \to \infty$. This is also true when steady state is reached through time marching, i.e., when $\phi_C^{t+\Delta t} = \phi_C^t$. This guarantees that the solution obtained when steady state is reached is the same as the one that would have been obtained with the problem solved directly as a steady state one.

For the discretization of the transient term, the practice traditionally has been to follow a finite difference approach [1–3], whereby a Taylor series expansion of $\partial(\rho\phi)/\partial t$ is used to express the derivative in terms of the discrete nodal values. In this chapter, another procedure that is more in line with the finite volume approach will also be presented. In this context, $\partial(\rho\phi)/\partial t$ is integrated over a temporal element [4] and transformed into face fluxes in a similar fashion to what was done with convection schemes, except that the discretization is now performed along the transient axes.

13.2 The Finite Difference Approach

Since in the transient space the grid is structured (Fig. 13.3), it has been quite common to treat the transient term using the finite difference method. In this approach, the spatial operator, $L(\phi)$, is discretized at time t, while the transient derivative is evaluated using a combination of Taylor expansions about time t resulting in a variety of transient schemes, some of which are described next.

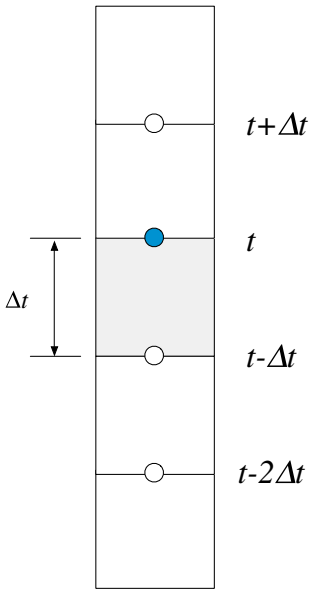


Fig. 13.3 Structured transient finite difference grid

13.2.1 Forward Euler Scheme

To evaluate the transient term, a Taylor expansion of the derived quantity about a time direction is needed. In this first case, the expansion is performed in a forward manner about time t. That is for some function T, its value at time $t + \Delta t$ is expressed using a Taylor series in terms of the values of T and its derivates at time t as

$$T(t + \Delta t) = T(t) + \frac{\partial T(t)}{\partial t}\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t^2}{2!} + \cdots. \tag{13.5}$$

Truncating the series starting with terms of order Δt^2, the first derivative can be formulated as

$$\frac{\partial T(t)}{\partial t} = \frac{T(t+\Delta t) - T(t)}{\Delta t} + O(\Delta t) \tag{13.6}$$

This is now a first order discretization since the equation was divided by Δt to yield the gradient approximation. Replacing T by $(\rho\phi)$ in Eq. (13.6) and substituting the resulting expression for the derivative in Eq. (13.3), the discretized equation becomes

$$\frac{(\rho_C\phi_C)^{t+\Delta t} - (\rho_C\phi_C)^t}{\Delta t} V_C + L\left(\phi_C^t\right) = 0. \tag{13.7}$$

The transient stencil for Eq. (13.7) shown in Fig. 13.4, indicates that the computation of $(\rho_C\phi_C)$ at time $t+\Delta t$ does not require solving a system of equations. Rather, values of ϕ_C at time $t+\Delta t$ can be computed explicitly based on values from the previous time step since all spatial terms are evaluated at the old time t. The resulting scheme belongs to the class denoted by explicit transient schemes [5–12]. The main characteristic of all explicit transient schemes is their capability of generating solutions by marching in time without the need to solve a system of equations at each time level. This provides a high computational efficiency and simplifies the parallelization of the computational mesh. Yet only few commercial codes have adopted this approach and for an important reason related to a limitation on the size of Δt, which will be discussed in the next section.

Substituting the discretized algebraic relation of the spatial operator into Eq. (13.7), the complete algebraic equation is obtained as

$$a_C^{t+\Delta t}\phi_C^{t+\Delta t} + a_C^t\phi_C^t = b_C - \left(a_C\phi_C^t + \sum_{F\sim NB(C)} a_F\phi_F^t\right) \tag{13.8}$$

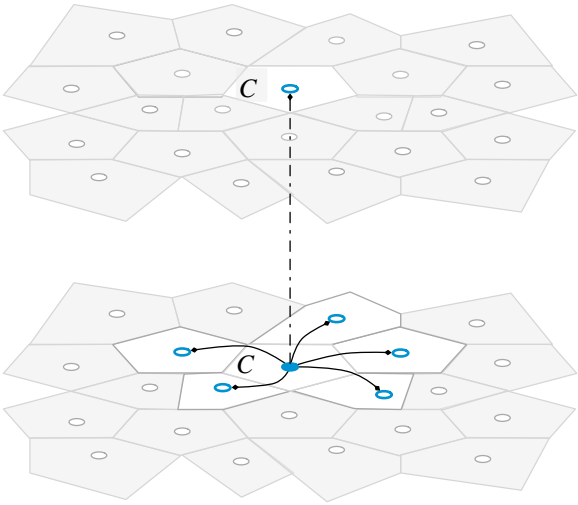


Fig. 13.4 The explicit Euler stencil

where

$$a_C^{t+\Delta t} = \frac{\rho_C^{t+\Delta t} V_C}{\Delta t}$$
$$a_C^{t} = -\frac{\rho_C^{t} V_C}{\Delta t} \tag{13.9}$$

In the above equations $a_C^{t+\Delta t}$ and a_C^t are the diagonal coefficients resulting from the discretization of the transient term, $\phi_C^{t+\Delta t}$ and ϕ_C^t are the values at time levels $t + \Delta t$ and t, respectively, and a_C, a_F, and b_C are the coefficients obtained from the spatial discretization.

To simplify notation, throughout this chapter variables referring to values obtained at a previous time step will be denoted with a superscript $^\circ$ and variables referring to values obtained two time steps earlier will be denoted with a superscript $^{\circ\circ}$. On the other hand no superscript will be used to denote variables at the current time step except for the coefficient of the unsteady term multiplying ϕ_C, which will be denoted with the superscript $^\bullet$. Adopting the new notation, Eqs. (13.8) and (13.9) become

$$a_C^\bullet \phi_C + a_C^\circ \phi_C^\circ = b_C - \left(a_C \phi_C^\circ + \sum_{F \sim NB(C)} a_F \phi_F^\circ \right) \tag{13.10}$$

where

$$a_C^\bullet = \frac{\rho_C V_C}{\Delta t}$$
$$a_C^\circ = -\frac{\rho_C^\circ V_C}{\Delta t} \tag{13.11}$$

Equation (13.10) can be re-arranged into

$$\phi_C = \frac{b_C - \left(\left(a_C + a_C^\circ \right) \phi_C^\circ + \sum_{F \sim NB(C)} a_F \phi_F^\circ \right)}{a_C^\bullet} \tag{13.12}$$

clearly showing that values of ϕ at the current time step are computed via an **explicit** relation without solving a system of equations.

13.2.2 Stability of the Forward Euler Scheme

The convergence and stability of numerical schemes was initially addressed by Courant, Friedrichs, and Lewy [13]. They showed that in order for the solution of a difference equation to converge to the solution of the partial differential equation the numerical scheme must use all the information contained in the initial data that influence the solution. This requirement has become later known as the CFL condition.

In reality the CFL condition can be interpreted simply as one of the basic rules that should be satisfied by the coefficients, namely the opposite signs rule extended to include the transient coefficients. Thus just as ϕ_F is considered a 'spatial' neighbor of ϕ_C, ϕ_C° is a 'temporal' neighbor of ϕ_C, and the opposite signs rule should equally apply to both. Noting that the diagonal coefficient is now $a_C^\bullet$ and the coefficient of its 'temporal' neighbor is $\left(a_C + a_C^\circ\right)$, the opposite signs requirement becomes

$$a_C + a_C^\circ \leq 0. \tag{13.13}$$

13.2.2.1 Stability of a Transient-Advection Case

For the one dimensional pure advection problem with a flow moving right wise shown in Fig. 13.5, the a_C and a_C° coefficients in the discretized equation of element C, using the upwind scheme for the interpolation of all variables at an element face, are given by

$$a_C = \dot{m}_e^\circ = \rho_C^\circ u_C^\circ \Delta y_C \qquad a_C^\circ = -\frac{\rho_C^\circ V_C}{\Delta t} = -\frac{\rho_C^\circ \Delta x_C \Delta y_C}{\Delta t} \tag{13.14}$$

Therefore, the CFL condition requires

$$a_C + a_C^\circ \leq 0 \quad \Rightarrow \quad \rho_C^\circ u_C^\circ \Delta y_C - \frac{\rho_C^\circ \Delta x_C \Delta y_C}{\Delta t} \leq 0 \tag{13.15}$$

or

$$\Delta t \leq \frac{\Delta x_C}{u_C^\circ}. \tag{13.16}$$

For convection dominated flows, defining a *CFL* number as

$$CFL^{conv} = \frac{\left|\mathbf{v}_C^\circ\right| \Delta t}{\Delta x_C} \tag{13.17}$$

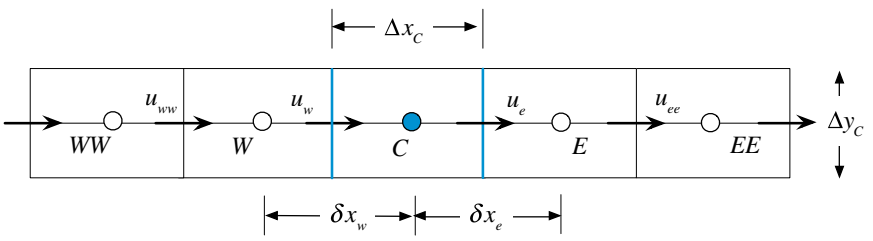


Fig. 13.5 A portion of the discretized domain for a one dimensional convection problem

implies that for numerical stability the *CFL* number should satisfy

$$CFL^{conv} \leq 1. \tag{13.18}$$

13.2.2.2 Stability of a Transient-Diffusion Case

For pure diffusion problems, the expression for the *CFL* number is different. For that purpose, the one dimensional pure diffusion problem schematically depicted in Fig. (13.6) is considered.

The a_C and a_C° coefficients in the discretized equation of element C using linear interpolation profiles are given by

$$a_C = \frac{\Gamma_e^\phi \Delta y_C}{\delta x_e} + \frac{\Gamma_w^\phi \Delta y_C}{\delta x_w} \qquad a_C^\circ = -\frac{\rho_C^\circ V_C}{\Delta t} = -\frac{\rho_C^\circ \Delta x_C \Delta y_C}{\Delta t} \tag{13.19}$$

Therefore, the CFL condition requires

$$a_C + a_C^\circ \leq 0 \quad \Rightarrow \quad \frac{\Gamma_e^\phi \Delta y_C}{\delta x_e} + \frac{\Gamma_w^\phi \Delta y_C}{\delta x_w} - \frac{\rho_C^\circ \Delta x_C \Delta y_C}{\Delta t} \leq 0 \tag{13.20}$$

or

$$\Delta t \leq \frac{\rho_C^\circ \Delta x_C}{\frac{\Gamma_e^\phi}{\delta x_e} + \frac{\Gamma_w^\phi}{\delta x_w}}. \tag{13.21}$$

For the case when the grid is uniform and the diffusion coefficient is constant, Eq. (13.21) becomes

$$\Delta t \leq \frac{\rho_C^\circ (\Delta x_C)^2}{2\Gamma_C^\phi}. \tag{13.22}$$

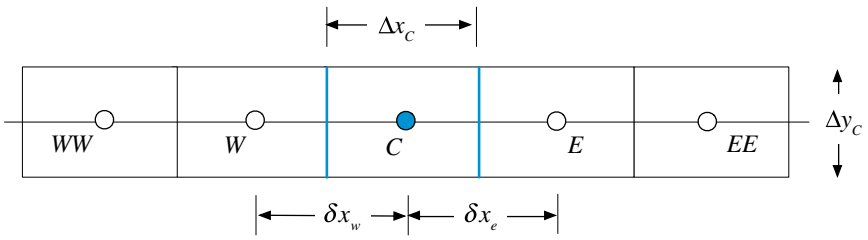


Fig. 13.6 A portion of the discretized domain for a one dimensional diffusion problem

For diffusion dominated problems, a *CFL* number is defined as

$$CFL^{diff} = \frac{\Gamma_C^{\phi} \Delta t}{\rho_C^{\circ} (\Delta x_C)^2} \tag{13.23}$$

implying that for stability the following condition should be satisfied:

$$CFL^{diff} \leq \frac{1}{2}. \tag{13.24}$$

13.2.2.3 Stability of a Transient-Convection-Diffusion Case

For the case of a multi dimensional unsteady convection and diffusion problem (Fig. 13.7) and based on the derivations presented in Chap. 12, the coefficients in Eq. (13.14) are given by

$$\begin{aligned} a_C^{\circ} &= -\frac{\rho_C^{\circ} V_C}{\Delta t} \\ a_C &= \sum_{f \sim nb(C)} \left(\Gamma_f^{\phi} \frac{E_f}{d_{CF}} + \left\| \dot{m}_f^{\circ}, 0 \right\| \right) \end{aligned} \tag{13.25}$$

Substituting the expressions for the coefficients from Eq. (13.25) in Eq. (13.13), the CFL condition becomes

$$\sum_{f \sim nb(C)} \left(\Gamma_f^{\phi} \frac{E_f}{d_{CF}} + \left\| \dot{m}_f^{\circ}, 0 \right\| \right) - \frac{\rho_C^{\circ} V_C}{\Delta t} \leq 0 \tag{13.26}$$

leading to the following constraint on the time step:

$$\Delta t \leq \frac{\rho_C^{\circ} V_C}{\sum_{f \sim nb(C)} \left(\Gamma_f^{\phi} \frac{E_f}{d_{CF}} + \left\| \dot{m}_f^{\circ}, 0 \right\| \right)}. \tag{13.27}$$

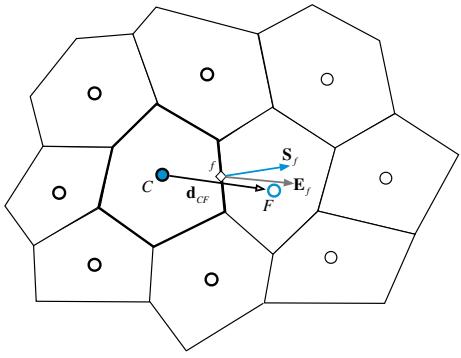


Fig. 13.7 A portion of the discretized domain for a multi dimensional convection problem

Equation (13.27) is the general requirement for stability of explicit transient schemes. In fact the conditions obtained earlier for pure convection and pure diffusion in one dimensional domains can be derived as special cases of Eq. (13.27). For the case of a one dimensional diffusion problem with a uniform grid of cell size Δx, constant density ρ, and a uniform diffusion coefficient Γ^{ϕ}, Eq. (13.27) reduces to

$$\Delta t \leq \frac{\rho_C^{\circ} \overbrace{V_C}^{\Delta x_C \Delta y_C}}{\sum_{f \sim nb(C)} \left(\underbrace{\Gamma_f^{\phi}}_{=\Gamma_e^{\phi}+\Gamma_w^{\phi}} \underbrace{\frac{E_f}{d_{CF}}}_{\Delta x_C} + \underbrace{\left\| \dot{m}_f^{\circ}, 0 \right\|}_{=0} \right)} \Rightarrow \Delta t \leq \frac{\rho_C^{\circ} (\Delta x_C)^2}{2\Gamma_C^{\phi}}. \tag{13.28}$$

While for the case of a one dimensional advection problem discretized using the upwind scheme and with the flow moving from left to right, Eq. (13.27) reduces to

$$\Delta t \leq \frac{\rho_C^{\circ} \overbrace{V_C}^{=\Delta x_C \Delta y_C}}{\sum_{f \sim nb(C)} \left(\underbrace{\Gamma_f^{\phi}}_{=0} \frac{E_f}{d_{CF}} + \underbrace{\left\| \dot{m}_f^{\circ}, 0 \right\|}_{\dot{m}_e^{\circ} = \rho_C^{\circ} u_C^{\circ} \Delta y_C} \right)} \Rightarrow \Delta t \leq \frac{\Delta x_C}{u_C^{\circ}}. \tag{13.29}$$

This stability constraint is stringent and very restrictive as it forces the use of extremely small time steps when solving transient problems. That is, whereas the computational cost at each time step is small in comparison to what would be required to solve a system of equations at that level, the imposed limitation by the CFL condition necessitates a larger number of steps to move the solution in time. Therefore the benefit of reducing the calculations per time step is lost by the much larger number of time steps required. Moreover, Eq. (13.27) indicates also that improving the spatial accuracy by decreasing the grid size, decreases further the maximum time step size that can be used without causing instabilities.

As shown next, such a constraint does not apply to implicit schemes for which the transient term has always the proper sign.

13.2.3 Backward Euler Scheme

To derive the backward Euler scheme, the value of the function T at time $t - \Delta t$ is expressed using a Taylor series using the values of T and its derivates at time t as

$$T(t-\Delta t) = T(t) - \frac{\partial T(t)}{\partial t}\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t^2}{2!} + \cdots \tag{13.30}$$

Manipulating Eq. (13.30), an equation for the first derivative is obtained as

$$\frac{\partial T(t)}{\partial t} = \frac{T(t) - T(t-\Delta t)}{\Delta t} + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t}{2!} + \cdots \tag{13.31}$$

Replacing T by $(\rho\phi)$ in Eq. (13.31) and substituting the resulting expression for the derivative in Eq. (13.3), the discretized equation becomes

$$\frac{(\rho_C\phi_C)^t - (\rho_C\phi_C)^{t-\Delta t}}{\Delta t} V_C + L\left(\phi_C^t\right) = 0. \tag{13.32}$$

Then invoking the algebraic relation of the spatial operator and the suggested notation, the complete algebraic form of the transient scalar equation is obtained as

$$\left(a_C^{\bullet} + a_C\right)\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C + a_C^{\circ}\phi_C^{\circ} \tag{13.33}$$

with the coefficients given by

$$\begin{aligned} a_C^{\bullet} &= \frac{\rho_C V_C}{\Delta t} \\ a_C^{\circ} &= -\frac{\rho_C^{\circ} V_C}{\Delta t} \end{aligned} \tag{13.34}$$

The stencil for Eq. (13.33) is shown in Fig. 13.8. It is clear that with the spatial operator evaluated at the same time level as the new temporal coefficient, resolving the ϕ field at

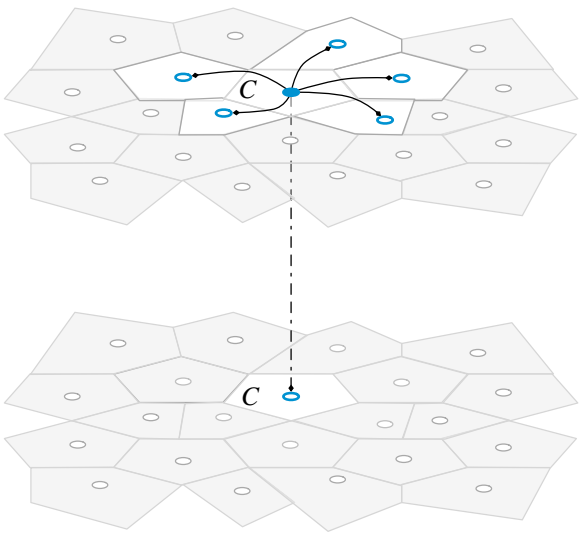


Fig. 13.8 Stencil for the backward Euler stencil

a new time level requires solving a system of equations. This type of schemes requiring the solution of a system of equations is denoted by implicit schemes [5–12].

As can be inferred from Eq. (13.34) a_C and a_C° are of opposite signs guaranteeing that ϕ_C is bounded by the values of its spatial neighbors at the current time step t and by the value of its temporal neighbor at the previous time step $t - \Delta t$. This implies that the scheme is always stable independent of the time step used, allowing for the solution to proceed rapidly by using large time steps. Nonetheless this is not the ideal scheme as it is of low order and solutions obtained with this scheme are of low accuracy unless small time steps are used, which puts its use in a quandary. Adopting large time steps for computational efficiency results in a solution of low accuracy and using small time steps for higher accuracy is associated with low computational efficiency.

13.2.4 Crank-Nicolson Scheme

In the Crank-Nicolson scheme [2, 14] a more accurate representation of the transient term is derived by expressing the values of the function T at times $t - \Delta t$ and $t + \Delta t$ in terms of the values of T and its derivates at time t as

$$\begin{aligned} T(t+\Delta t) &= T(t) + \frac{\partial T(t)}{\partial t}\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t^2}{2!} + \frac{\partial^3 T(t)}{\partial t^3}\frac{\Delta t^3}{3!} + \cdots \\ T(t-\Delta t) &= T(t) - \frac{\partial T(t)}{\partial t}\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t^2}{2!} - \frac{\partial^3 T(t)}{\partial t^3}\frac{\Delta t^3}{3!} + \cdots \end{aligned} \tag{13.35}$$

Then, subtracting $T(t + \Delta t)$ from $T(t - \Delta t)$ given in Eq. (13.35), an equation for the first derivative is obtained as

$$\frac{\partial T(t)}{\partial t} = \frac{T(t+\Delta t) - T(t-\Delta t)}{2\Delta t} + O\left(\Delta t^2\right) \tag{13.36}$$

Note that the order of accuracy of the derivative is now $O(\Delta t^2)$ since the second order derivative is completely eliminated.

Substituting the time derivative given by Eq. (13.36) into Eq. (13.3) yields

$$\frac{\left(\rho_C \phi_C\right)^{t+\Delta t} - \left(\rho_C \phi_C\right)^{t-\Delta t}}{2\Delta t} V_C + L\left(\phi_C^t\right) = 0 \tag{13.37}$$

Then invoking the algebraic relation of the spatial operator, and using the suggested notation, the complete algebraic form of the transient scalar equation is obtained as

$$a_C^\bullet \phi_C = b_C - \left(a_C \phi_C^\circ + \sum_{F \sim NB(C)} a_F \phi_F^\circ \right) - a_C^{\circ\circ} \phi_C^{\circ\circ} \tag{13.38}$$

with the coefficients given by

$$\begin{aligned} a_C^{\bullet} &= \frac{\rho_C V}{2\Delta t} \\ a_C^{\circ\circ} &= -\frac{\rho_C^{\circ\circ} V}{2\Delta t} \end{aligned} \tag{13.39}$$

The stencil for Eq. (13.38) is shown in Fig. 13.9. It is clear that the scheme is an explicit type scheme, since the evaluation of $(\rho\phi)^{t+\Delta t}$ can be performed using only old values. However two old levels are now needed, with the spatial operator being evaluated at one of these levels.

An analysis of the stability of the CN scheme can be performed after slightly modifying the original equation. Using the following approximation:

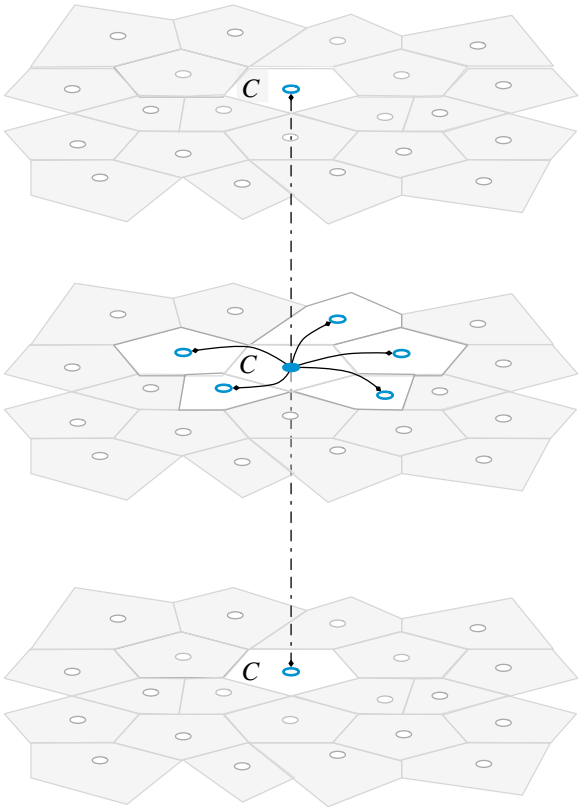


Fig. 13.9 Stencil of the Crank Nicholson Scheme

$$\phi^{\circ} \approx \frac{\phi + \phi^{\circ\circ}}{2} \tag{13.40}$$

the algebraic equation [Eq. (13.38)] becomes

$$a_C^{\bullet}\phi_C + 0.5\left(a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F\right) = b_C - 0.5\left(\left(a_C + 2a_C^{\circ\circ}\right)\phi_C^{\circ\circ} + \sum_{F\sim NB(C)} a_F\phi_F^{\circ\circ}\right) \tag{13.41}$$

Thus, the stability condition becomes

$$a_C + 2a_C^{\circ\circ} \le 0. \tag{13.42}$$

For the one dimensional transient advection problem displayed in Fig. 13.5, Eq. (13.42) results in

$$\Delta t \le \frac{2\rho_C^{\circ\circ}V_C}{\dot{m}_e^{\circ}} = \frac{2\rho_C^{\circ\circ}\Delta x_C\Delta y_C}{\rho_C^{\circ}u_C^{\circ}\Delta y_C} \approx \frac{2\Delta x_C}{\left|\mathbf{v}_e^{\circ}\right|}, \tag{13.43}$$

where it has been assumed that the advection term is discretized using the upwind scheme. Using the *CFL* number for convection defined above, Eq. (13.43) is expressed as

$$CFL^{conv} \le 2 \tag{13.44}$$

The larger CFL limitation is pleasing, but the improved accuracy is just more important as it allows for accurate solutions to be achieved without the need to resort to very small time steps, especially that the second order derivative is now eliminated from the error. More details on accuracy analysis will be presented in later sections.

13.2.5 Implementation Details

The CN scheme can also be derived by summing the Forward and Backward transient Euler schemes [4], as shown next.

$$\text{Forward Euler} \rightarrow \frac{(\rho_C\phi_C)^t - (\rho_C\phi_C)^{t-\Delta t}}{\Delta t}V_C = -L\left(\phi_C^t\right) \tag{13.45}$$

$$\text{Backward Euler} \rightarrow \frac{(\rho_C\phi_C)^{t+\Delta t}-(\rho_C\phi_C)^{t}}{\Delta t}V_C = -L\left(\phi_C^t\right) \tag{13.46}$$

Forward Euler + Backward Euler:

$$\begin{aligned}
&\rightarrow \frac{(\rho_C\phi_C)^{t}-(\rho_C\phi_C)^{t-\Delta t}}{\Delta t}V_C + \frac{(\rho_C\phi_C)^{t+\Delta t}-(\rho_C\phi_C)^{t}}{\Delta t}V_C = -L\left(\phi_C^t\right) - L\left(\phi_C^t\right)\\
&\rightarrow \frac{(\rho_C\phi_C)^{t+\Delta t}-(\rho_C\phi_C)^{t-\Delta t}}{2\Delta t}V_C + L\left(\phi_C^t\right) = 0\\
&\rightarrow \text{Crank} - \text{Nicolson}
\end{aligned} \tag{13.47}$$

This formulation points to a simple implementation of the CN scheme within an implicit scheme framework as a two-step procedure. In the first step a Backward Euler formulation is used to implicitly find $(\rho\phi)^t$ from

$$(\rho_C\phi_C)^t + \frac{\Delta t}{V_C}L\left(\phi_C^t\right) = (\rho_C\phi_C)^{t-\Delta t} \tag{13.48}$$

while in the second step the CN value at time step $t + \Delta t$ is found explicitly as

$$\begin{aligned}
\frac{(\rho_C\phi_C)^{t+\Delta t}-(\rho_C\phi_C)^{t}}{\Delta t}V_C &= -L\left(\phi_C^t\right) = \frac{(\rho_C\phi_C)^{t}-(\rho_C\phi_C)^{t-\Delta t}}{\Delta t}V_C\\
&\Rightarrow (\rho_C\phi_C)^{t+\Delta t} = 2(\rho_C\phi_C)^{t}-(\rho_C\phi_C)^{t-\Delta t}
\end{aligned} \tag{13.49}$$

In this derivation it was assumed that the transient time step Δt is divided into two equal local time steps (Δt_{local}), with Δt_{local} equals half the set time step Δt.

It is important to note that while the CN scheme is second order accurate, it is still an explicit scheme, which is constrained by a CFL like condition, as explained above.

13.2.6 Adams-Moulton Scheme

The development of the second order Adams-Moulton scheme [15, 16] requires expanding the values of T at $t - \Delta t$ and $t - 2\Delta t$ using Taylor series expansions around t, yielding

$$\begin{aligned}
T(t-2\Delta t) &= T(t) - \frac{\partial T(t)}{\partial t}2\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{4\Delta t^2}{2!} + \cdots\\
T(t-\Delta t) &= T(t) - \frac{\partial T(t)}{\partial t}\Delta t + \frac{\partial^2 T(t)}{\partial t^2}\frac{\Delta t^2}{2!} + \cdots
\end{aligned} \tag{13.50}$$

The first derivative is obtained by combining the two equations in such a way that the second order derivative is eliminated, resulting in the following equation:

$$\frac{\partial T(t)}{\partial t} = \frac{3T(t) - 4T(t-\Delta t) + T(t-2\Delta t)}{2\Delta t} \tag{13.51}$$

which, upon substituting in Eq. (13.3), yields

$$\frac{3(\rho_C\phi_C)^t - 4(\rho_C\phi_C)^{t-\Delta t} + (\rho_C\phi_C)^{t-2\Delta t}}{2\Delta t} V_C + L\left(\phi_C^t\right) = 0 \tag{13.52}$$

Expanding the spatial term, the final form of the algebraic equation is obtained as

$$\left(a_C^{\bullet} + a_C\right)\phi_C + \sum_{F \sim NB(C)} a_F\phi_F = b_C - a_C^{\circ}\phi_C^{\circ} - a_C^{\circ\circ}\phi_C^{\circ\circ} \tag{13.53}$$

with the coefficients given by

$$\begin{aligned} a_C^{\bullet} &= \frac{3\rho_C V_C}{2\Delta t} \\ a_C^{\circ} &= -\frac{2\rho_C^{\circ} V_C}{\Delta t} \\ a_C^{\circ\circ} &= \frac{\rho_C^{\circ\circ} V_C}{2\Delta t} \end{aligned} \tag{13.54}$$

It is clear that the $a_C^{\circ\circ}$ coefficient has a positive sign implying that an increase in $\phi_C^{\circ\circ}$ would lead to a decrease in ϕ_C. This is mitigated by the large a_C° coefficient, which has the right influence. Thus while the scheme is stable, it is not bounded with unphysical oscillations expected in certain circumstances.

Example 1

The thermal conductivity of a solid sphere of volume 1 m^3 *is so high that its resistance to conduction is very small as compared to its resistance to convection heat transfer with the surroundings. Thus temperature gradients within the sphere are negligible and the temperature of the sphere is spatially uniform at any instant. The initial temperature of the sphere is* T_h *and that of the surroundings is* T_∞*. The density, specific heat, sphere surface area, and convection heat transfer coefficient with the surroundings are* ρ, c, A_s, *and* h_∞, *respectively. Neglecting heat transfer by radiation, the energy equation for the sphere is given by*

$$\rho c V \frac{dT}{dt} = -h_\infty A_S (T - T_\infty)$$

Defining a dimensionless temperature as

$$\phi = \frac{T - T_\infty}{T_h - T_\infty}$$

the energy equation and initial condition become

$$\frac{d\phi}{dt} = -\frac{h_\infty A_S}{\rho c V}\phi \text{ and } \phi(0) = 1$$

For a value of $h_\infty A_S/\rho cV = 1$, *compare dimensionless temperature values obtained analytically with numerical ones generated using the first order explicit scheme, the first order implicit scheme, and the second order CN scheme applied via the two-step procedure at times* 0.1, 0.2, *and* 0.3 *using a time step with size of* 0.1.

Solution

The governing equation reduces to

$$\frac{d\phi}{dt} = -\phi$$

subject to

$$\phi(0) = 1$$

By separation of variables and application of the initial condition the analytical solution is found as

$$\phi(t) = e^{-t}$$

Thus the analytical solution at times 0.1, 0.2, and 0.3 are given by

$$\begin{aligned}\phi_{exact}(0.1) &= e^{-0.1} = 0.9048\\ \phi_{exact}(0.2) &= e^{-0.2} = 0.8187\\ \phi_{exact}(0.3) &= e^{-0.3} = 0.7408\end{aligned}$$

The numerical solution is obtained with $V = 1$, $L(\phi^n) = -\phi^n$, and $L(\phi^{n+1}) = -\phi^{n+1}$.

The error in the numerical solution is found using

$$error = |\phi_{numerical} - \phi_{exact}|$$

Numerical solution using the first order explicit scheme

$$\phi^{t+\Delta t} = (1 - \Delta t)\phi^t$$

$$\left.\begin{aligned}\phi_{explicit}(0.1) &= (1-0.1)\phi(0) = 0.9*1 = 0.9\\ \phi_{explicit}(0.2) &= (1-0.1)\phi(0.1) = 0.9*0.9 = 0.81\\ \phi_{explicit}(0.3) &= (1-0.1)\phi(0.2) = 0.9*0.81 = 0.729\end{aligned}\right\} \Rightarrow \begin{cases} error_{explicit}(0.1) = 4.8\times 10^{-3}\\ error_{explicit}(0.2) = 8.7\times 10^{-3}\\ error_{explicit}(0.3) = 1.18\times 10^{-2}\end{cases}$$

Numerical solution using the first order implicit scheme

$$\phi^{t+\Delta t} = \frac{1}{1+\Delta t}\phi^{t}$$

$$\left.\begin{aligned}\phi_{implicit}(0.1) &= \frac{1}{1+0.1}(1) = 0.9091\\ \phi_{implicit}(0.2) &= \frac{1}{1+0.1}(0.9091) = 0.8264\\ \phi_{implicit}(0.3) &= \frac{1}{1+0.1}(0.8264) = 0.7513\end{aligned}\right\} \Rightarrow \begin{cases} error_{implicit}(0.1) = 4.3\times 10^{-3}\\ error_{implicit}(0.2) = 7.7\times 10^{-3}\\ error_{implicit}(0.3) = 1.05\times 10^{-2}\end{cases}$$

Numerical solution using the second order CN scheme

In this case the solution is obtained using Eqs. (13.48) and (13.49) that are reduced to

$$\begin{aligned}\phi^{*}(t+\Delta t/2) &= \frac{1}{1+\Delta t/2}\phi(t)\\ \phi_{CN}(t+\Delta t) &= 2\phi^{*}(t+\Delta t/2) - \phi(t)\end{aligned}$$

where the total time step Δt has been divided into two equal time steps of value $\Delta t/2$. Applying the above equations, the solutions are found as

$$\left.\begin{aligned}\phi^{*}(0.05) &= \frac{1}{1+0.05}\phi(0) = 0.95238\\ \phi_{CN}(0.1) &= 2\phi^{*}(0.05) - \phi(0) = \boxed{0.90476}\\ \phi^{*}(0.15) &= \frac{1}{1+0.05}\phi(0.1) = 0.861678\\ \phi_{CN}(0.2) &= 2\phi^{*}(0.15) - \phi(0.1) = \boxed{0.81859}\\ \phi^{*}(0.25) &= \frac{1}{1+0.05}\phi(0.2) = 0.779615\\ \phi_{CN}(0.3) &= 2\phi^{*}(0.25) - \phi(0.2) = \boxed{0.7406}\end{aligned}\right\} \Rightarrow \begin{cases} error_{CN}(0.1) = 7.551\times 10^{-5}\\ error_{CN}(0.2) = 1.366\times 10^{-4}\\ error_{CN}(0.3) = 1.854\times 10^{-4}\end{cases}$$

13.3 The Finite Volume Approach

The Finite Volume approach for the discretization of the transient term is very similar to the discretization of the convective term [4], except that the integration is carried over temporal rather than spatial element (Fig. 13.10).

Integration of Eq. (13.3) over the time interval $[t - \Delta t/2,\ t + \Delta t/2]$ yields

$$\underbrace{\int_{t-\Delta t/2}^{t+\Delta t/2} \frac{\partial(\rho_C \phi_C)}{\partial t} V_C dt}_{Term\ I} + \underbrace{\int_{t-\Delta t/2}^{t+\Delta t/2} L(\phi_C) dt}_{Term\ II} = 0 \tag{13.55}$$

With V_C treated as a constant, *Term I* turned into a difference of face fluxes, and *Term II* evaluated as a volume integral using the mid point rule, Eq. (13.55) becomes

$$V_C(\rho_C \phi_C)^{t+\Delta t/2} - V_C(\rho_C \phi_C)^{t-\Delta t/2} + L\left(\phi_C^t\right)\Delta t = 0 \tag{13.56}$$

Equation (13.56) is the semi-discretized transient equation, which can be written in the more standard form by dividing all terms by the temporal element volume, Δt, leading to

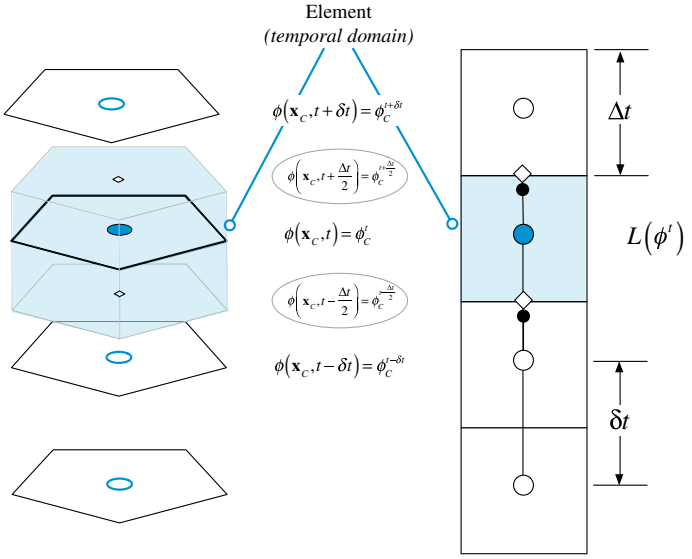


Fig. 13.10 Element in the transient domain

$$\frac{(\rho_C\phi_C)^{t+\Delta t/2}-(\rho_C\phi_C)^{t-\Delta t/2}}{\Delta t}V_C + L\left(\phi_C^t\right) = 0 \tag{13.57}$$

To derive the full discretized equation, an interpolation profile expressing the face values at $(t-\Delta t/2)$ and $(t+\Delta t/2)$ in terms of the element values at (t), $(t-\Delta t)$, etc., is needed. The selection of this profile can heavily rely on the understanding gained from the discretization of the convection term. The choice will obviously affect the accuracy and robustness of the method. In that regard, it is worth mentioning that the integration of the spatial operator is second order in time, but the accuracy of the operator itself is determined by the options used during its discretization.

Independent of the profile used, the flux will be linearized based on old and new values as

$$FluxT = FluxC\phi_C + FluxC^\circ\phi_C^\circ + FluxV \tag{13.58}$$

where again superscript ° refers to old values. With the linearization completed, the coefficients of the algebraic equation can then be assembled into

$$\begin{aligned} a_C &\leftarrow a_C + FluxC \\ b_C &\leftarrow b_C - FluxC^\circ\phi_C^\circ - FluxV \end{aligned} \tag{13.59}$$

In what follows the discretization for a number of interpolation profiles is presented.

13.3.1 First Order Transient Schemes

The first order implicit and explicit Euler schemes will be constructed next by adopting an upwind [14, 17] and a downwind [4, 18] transient interpolation profile, respectively.

13.3.2 First Order Implicit Euler Scheme

The transient first order implicit Euler scheme is obtained by using a first-order "upwind" interpolation profile [14, 17]. As shown in Fig. 13.11, the value of $\rho\phi$ at the temporal element face is set equal to the value at the centroid of the upwind element to give

$$(\rho_C\phi_C)^{t+\Delta t/2} = (\rho_C\phi_C)^t \text{ and } (\rho_C\phi_C)^{t-\Delta t/2} = (\rho_C\phi_C)^{t-\Delta t} \tag{13.60}$$

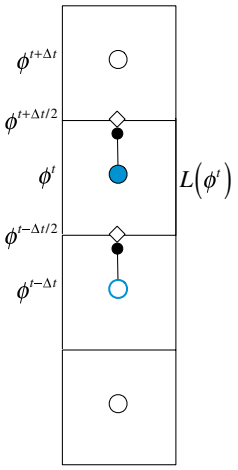


Fig. 13.11 First order transient upwind interpolation profile resulting in the implicit first order transient Euler scheme

Using Eq. (13.60), Eq. (13.57) becomes

$$\frac{(\rho_C\phi_C)^t-(\rho_C\phi_C)^{t-\Delta t}}{\Delta t}V_C+L\left(\phi_C^t\right)=0 \tag{13.61}$$

which is the first order implicit Euler scheme. The scheme is linearized as follows:

$$\begin{aligned} FluxC &= \frac{\rho_C V_C}{\Delta t} \\ FluxC^{\circ} &= -\frac{\rho_C^{\circ} V_C}{\Delta t} \\ FluxV &= 0 \end{aligned} \tag{13.62}$$

13.3.2.1 Numerical Diffusion

As this is a first order scheme, it is expected, based on the knowledge gained from convection schemes, to produce numerical diffusion. Its value can be determined by trying to recover the original governing equation using a Taylor series expansion around time t. The value of $(\rho\phi)^{t-\Delta t}$ can be expressed as

$$(\rho\phi)^{t-\Delta t}=(\rho\phi)^t-\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t\Delta t+\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t\frac{\Delta t^2}{2}+O\left(\Delta t^3\right) \tag{13.63}$$

which can be rearranged into

$$\frac{(\rho\phi)^t-(\rho\phi)^{t-\Delta t}}{\Delta t}=\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t-\underbrace{\left(\frac{\Delta t}{2}\right)\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t}_{\substack{\textit{Numerical}\\ \textit{diffusion}\\ \textit{term}}}-O\left(\Delta t^2\right) \tag{13.64}$$

Substituting Eq. (13.64) into the discretized equation gives

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t+\frac{1}{V_C}L\left(\phi_C^t\right)=\underbrace{\left(\frac{\Delta t}{2}\right)\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t}_{\substack{\textit{Numerical}\\ \textit{diffusion}\\ \textit{term}}}+O\left(\Delta t^2\right) \tag{13.65}$$

In effect a numerical diffusion term has been added to the equation that scales with the time step in a similar fashion to the upwind scheme for the advection term. So while the scheme is unconditionally stable, the solution it yields is really a stationary solution for large time steps.

13.3.3 First Order Explicit Euler Scheme

The transient first order explicit Euler scheme is obtained by using a first-order "downwind" interpolation profile [4, 18]. As shown in Fig. 13.12, the value of $\rho\phi$ at

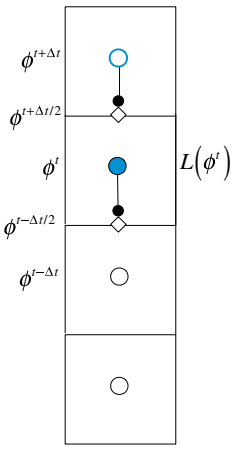


Fig. 13.12 First order transient downwind interpolation profile resulting in the explicit first order transient Euler scheme

the temporal element face is set equal to the value at the downwind element centroid, yielding

$$(\rho_C\phi_C)^{t+\Delta t/2} = (\rho_C\phi_C)^{t+\Delta t} \quad \text{and} \quad (\rho_C\phi_C)^{t-\Delta t/2} = (\rho_C\phi_C)^{t} \tag{13.66}$$

Using Eq. (13.66), Eq. (13.57) becomes

$$\frac{(\rho_C\phi_C)^{t+\Delta t} - (\rho_C\phi_C)^{t}}{\Delta t} V_C + L\left(\phi_C^t\right) = 0 \tag{13.67}$$

which is the first order explicit Euler scheme. The scheme is linearized as follows:

$$\begin{aligned} FluxC &= \frac{\rho_C V_C}{\Delta t} \\ FluxC^{\circ} &= -\frac{\rho_C^{\circ} V_C}{\Delta t} \\ FluxV &= 0 \end{aligned} \tag{13.68}$$

Note that now the new time is at $t + \Delta t$ and that the spatial operator of Eq. (13.67) has to be evaluated at time t. Thus, it is possible to evaluate the right hand side completely and find the value of $\rho\phi$ at time $t + \Delta t$ without the need to solve a set of linear algebraic equations. This is the explicit scheme and corresponds to the assumption that $\rho\phi$ prevails over the entire time step.

13.3.3.1 Numerical Anti-Diffusion

Again performing a simple Taylor expansion around time t yields

$$(\rho\phi)^{t+\Delta t} = (\rho\phi)^{t} + \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t \Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t \frac{\Delta t^2}{2} + O\left(\Delta t^3\right) \tag{13.69}$$

which can be rewritten as

$$\frac{(\rho\phi)^{t+\Delta t} - (\rho\phi)^{t}}{\Delta t} = \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \left(\frac{\Delta t}{2}\right)\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t + O\left(\Delta t^2\right) \tag{13.70}$$

Substitution into Eq. (13.67) gives

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \frac{1}{V_C} L\left(\phi_C^t\right) = \underbrace{-\left(\frac{\Delta t}{2}\right)\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t}_{\substack{Numerical \\ anti-diffusion \\ term}} + O\left(\Delta t^2\right) \tag{13.71}$$

Where now the second order differential term has a negative sign, akin to a negative diffusion or anti-diffusion, with compression effects on profiles, very similar to the Downwind scheme in advection. Again the anti-diffusion term scales with the time step. When used in combination with an upwind convection scheme and a Courant number of 1, it can be shown that the numerical diffusion of the advection scheme and the numerical anti-diffusion of the explicit Euler scheme for a CFL^{conv} equals to 1 are of equal magnitudes and of opposite signs. Thus they cancel each other producing nearly an exact solution. Nonetheless this is not practical as ensuring a CFL^{conv} of 1 on anything but simple one dimensional grids is not an option for real problems.

A related issue to the anti-diffusion behavior is numerical instabilities, which increases with increasing Δt placing a very strong restriction on the time step. This can be evaluated by applying the negative neighboring coefficient rule.

13.3.4 Second Order Transient Euler Schemes

Similar to advection schemes, second order transient schemes can be constructed with a linear interpolation profile. The choice could be a symmetric profile (central difference) yielding the Crank-Nicolson (CN) scheme [2], or an upwind one (second order upwind scheme) [4, 19, 20] resulting in the Adams-Moulton scheme [15, 16], an implicit scheme also known as the Second Order Upwind Euler (SOUE).

13.3.5 Crank-Nicholson (Central Difference Profile)

With the $\rho\phi$ computed using linear interpolation between the "Upwind" and "Downwind" nodes, the Crank-Nicholson scheme shown in Fig. 13.13 is obtained.

For a uniform time step, this is expressed mathematically as

$$\begin{aligned}(\rho_C\phi_C)^{t+\Delta t/2} &= \frac{1}{2}(\rho_C\phi_C)^{t+\Delta t}+\frac{1}{2}(\rho_C\phi_C)^{t}\\ (\rho_C\phi_C)^{t-\Delta t/2} &= \frac{1}{2}(\rho_C\phi_C)^{t}+\frac{1}{2}(\rho_C\phi_C)^{t-\Delta t}\end{aligned} \tag{13.72}$$

Substituting in Eq. (13.57), the discretized equation becomes

$$\frac{(\rho_C\phi_C)^{t+\Delta t}-(\rho_C\phi_C)^{t-\Delta t}}{2\Delta t}V_C + L\left(\phi_C^t\right) = 0 \tag{13.73}$$

The linearization coefficients for the CN scheme can be written as

$$\begin{aligned}FluxC &= \frac{\rho_C V_C}{2\Delta t}\\ FluxC^{\circ} &= 0\\ FluxV &= -\frac{\rho_C^{\circ\circ} V_C}{2\Delta t}\phi_C^{\circ\circ}\end{aligned} \tag{13.74}$$

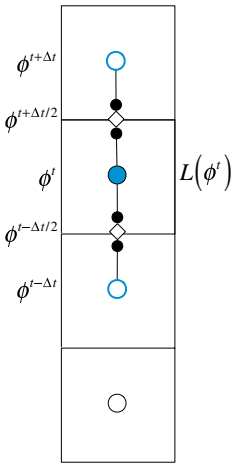


Fig. 13.13 Second order transient central difference interpolation profile resulting in the transient CN scheme

The stencil shown in Fig. 13.9 indicates that the scheme is explicit with the value at level $t + \Delta t$ computed explicitly from the values at times t and $t - \Delta t$. Thus its stability is constrained by a CFL limit.

Again in a similar fashion to the finite difference formulation, it can be reformulated in a two-step procedure using Eqs. (13.48) and (13.49), i.e., a first order implicit Euler step followed by a modified explicit Euler step in the form of extrapolation.

13.3.5.1 Numerical Accuracy

Expanding $(\rho\phi)$ at $t + \Delta t$ and $t - \Delta t$ via Taylor expansions around time t yields

$$(\rho\phi)^{t+\Delta t} = (\rho\phi)^t + \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t \Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t \frac{\Delta t^2}{2} + \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^3}{6} + O(\Delta t^4) \quad (13.75)$$

$$(\rho\phi)^{t-\Delta t} = (\rho\phi)^t - \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t \Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t \frac{\Delta t^2}{2} - \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^3}{6} + O(\Delta t^4) \quad (13.76)$$

Subtracting Eq. (13.75) from Eq. (13.76), the following equation is obtained:

$$\frac{(\rho\phi)^{t+\Delta t} - (\rho\phi)^{t-\Delta t}}{2\Delta t} = \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^2}{6} - O(\Delta t^3) \quad (13.77)$$

Substitution into Eq. (13.73) gives

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \frac{1}{V_C}L\left(\phi_C^t\right) = -\left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^2}{6} + O\left(\Delta t^3\right) \tag{13.78}$$

confirming that the scheme is second order accurate. The third order derivative is a dispersive term that results in instability.

13.3.6 Second Order Upwind Euler (SOUE) Scheme

Using the second-order "upwind" interpolation profile depicted in Fig. 13.14, the interface $\rho\phi$ values are approximated as

$$\begin{aligned}(\rho_C\phi_C)^{t+\Delta t/2} &= \frac{3}{2}(\rho_C\phi_C)^t - \frac{1}{2}(\rho_C\phi_C)^{t-\Delta t} \\ (\rho_C\phi_C)^{t-\Delta t/2} &= \frac{3}{2}(\rho_C\phi_C)^{t-\Delta t} - \frac{1}{2}(\rho_C\phi_C)^{t-2\Delta t}\end{aligned} \tag{13.79}$$

Substituting in Eq. (13.57), the discretized $\rho\phi$ field equation is obtained as

$$\frac{3(\rho_C\phi_C)^t - 4(\rho_C\phi_C)^{t-\Delta t} + (\rho_C\phi_C)^{t-2\Delta t}}{2\Delta t}V_C + L\left(\phi_C^t\right) = 0 \tag{13.80}$$

which is the implicit second order upwind Euler (SOUE) scheme. In this scheme the values of $\rho\phi$ have to be stored for two of the older time steps, with its linearization coefficients given by

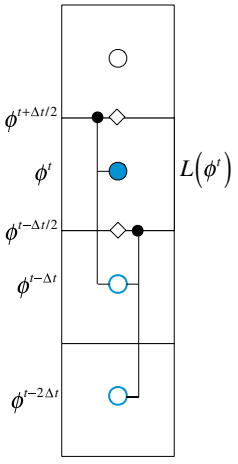


Fig. 13.14 Second order upwind Euler scheme

$$\begin{aligned} FluxC &= \frac{3\rho_C V_C}{2\Delta t} \\ FluxC^{\circ} &= -\frac{2\rho_C^{\circ} V_C}{\Delta t} \\ FluxV &= \frac{\rho_C^{\circ\circ} V_C \phi_C^{\circ\circ}}{2\Delta t} \end{aligned} \tag{13.81}$$

13.3.6.1 Numerical Accuracy

The scheme is second order as can be shown from a Taylor series evaluation. Expanding $(\rho\phi)^{t-\Delta t}$ and $(\rho\phi)^{t-2\Delta t}$ around time t give

$$(\rho\phi)^{t-\Delta t} = (\rho\phi)^{t} - \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t \Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t \frac{\Delta t^2}{2} - \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^3}{6} + O\left(\Delta t^4\right) \tag{13.82}$$

$$(\rho\phi)^{t-2\Delta t} = (\rho\phi)^{t} - \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t 2\Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t 2\Delta t^2 - \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{8\Delta t^3}{6} + O\left(\Delta t^4\right) \tag{13.83}$$

Multiplying Eq. (13.82) by 4 and subtracting the resulting equation from Eq. (13.83), an expression for the SOUE is obtained as

$$\frac{3(\rho\phi)^{t} - 4(\rho\phi)^{t-\Delta t} + (\rho\phi)^{t-2\Delta t}}{2\Delta t} = \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t - \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^2}{3} - O\left(\Delta t^3\right) \tag{13.84}$$

Combining Eq. (13.84) and Eq. (13.80), the recovered equation for $\rho\phi$ becomes

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \frac{1}{V_C} L\left(\phi_C^t\right) = \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_t \frac{\Delta t^2}{3} + O\left(\Delta t^3\right) \tag{13.85}$$

which has a third order numerical dispersion term but no numerical diffusion.

13.3.7 Initial Condition for the FV Approach

The implementation of the finite volume formulation is straight forward except for the initial time step. As shown in Fig. 13.15, the first temporal element is a boundary element in time, as such it does not have an upwind neighbor. Rather the value at the lower element face is used directly at the face resulting in a gradient that is half the correct numerical value. This comes about because it is computed as the difference between the values at $\phi_C^{t_{initial}+\Delta t/2}$ and $\phi_C^{t_{initial}}$, which are located half a

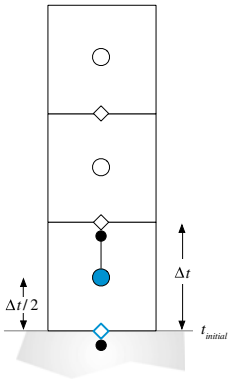


Fig. 13.15 Boundary temporal element

time step $(\Delta t/2)$ apart, while dividing their difference by a full time step (Δt) leading to a non-negligible initial error.

This is easily demonstrated by considering the first temporal element in the discretized equation of the first order implicit Euler scheme. Using Eq. (13.57) the discretized $\rho\phi$ field equation is obtained as

$$\frac{(\rho_C\phi_C)^{t_{initial}+\Delta t/2}-(\rho_C\phi_C)^{t_{initial}}}{\Delta t}V_C + L\left(\phi_C^{t_{initial}+\Delta t/2}\right) = 0 \tag{13.86}$$

For the first temporal element, the upwind interpolation yields a gradient computed as the difference between the $\rho\phi$ values at $t_{initial} + \Delta t/2$ and $t_{initial}$ divided by Δt. However for the case of a regular element (Fig. 13.16), the gradient is actually between the $\rho\phi$ values at $t_{initial} + 3\Delta t/2$ and $t_{initial} + \Delta t/2$, divided by Δt. The

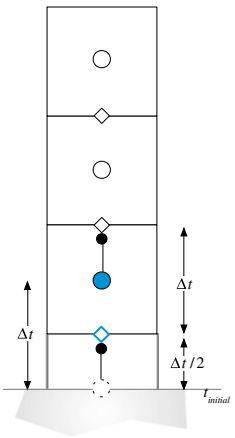


Fig. 13.16 Treatment of initial condition and the virtual element of centroid $t_{initial}$

difference between the two gradients is substantial, and any scheme that starts with the gradient of Eq. (13.86) will result in a large initial error that will affect the solution at the following steps. This error can be avoided if a grid similar to Fig. 13.16 is adopted. In this case the solution of the finite difference and finite volume methods will be basically similar, as for a regular grid.

Adopting this approach, the upwind values at the faces of the first temporal element spanning the time interval $[t_{initial} + \Delta t/2,\ t_{initial} + 3\Delta t/2]$ are obtained as

$$\begin{aligned}(\rho_C\phi_C)^{t_{initial}+3\Delta t/2} &= (\rho_C\phi_C)^{t_{initial}+\Delta t}\\ (\rho_C\phi_C)^{t_{initial}+\Delta t/2} &= (\rho_C\phi_C)^{t_{initial}}\end{aligned} \tag{13.87}$$

Substituting in Eq. (13.57), the discretized $\rho\phi$ field equation becomes

$$\frac{(\rho_C\phi_C)^{t_{initial}+\Delta t} - (\rho_C\phi_C)^{t_{initial}}}{\Delta t} V_C + L\left(\phi_C^{t_{initial}+\Delta t}\right) = 0 \tag{13.88}$$

which is similar to the equation obtained for any internal element.

Example 2

Repeat example 1 *using the CN scheme applied via Eq.* (13.73) *and the SOUE scheme. Use a time step* 0.05 *to find the values at* 0.1, 0.2, *and* 0.3.

Solution

The analytical solution at times 0.1, 0.2, and 0.3 were found in example 1.

Since two old values are needed, the value at the first time step is found using the first order backward Euler scheme. Thus the numerical solution is obtained using Eqs. (13.61), (13.73), and (13.80), which are reduced to

$$\begin{aligned}\phi_{EU}(t+\Delta t) &= \frac{1}{1+\Delta t}\phi^{\circ}\\ \phi_{CN}(t+\Delta t) &= \phi^{\circ\circ} - 2\Delta t\phi^{\circ}\\ \phi_{SOEU}(t+\Delta t) &= \frac{4\phi^{\circ} - \phi^{\circ\circ}}{3 + 2\Delta t}\end{aligned}$$

Based on the suggested implementation note, the first temporal element spans the time interval [0.025, 0.075], the second element spans the interval [0.075, 0.125], and so on.

Numerical solution using the second order CN scheme

Applying the above equations, the solutions are found as

$$\left.\begin{aligned}
&\phi_{EU}(0.05) = \frac{1}{1+0.05}\phi(0) = 0.95238 \\
&\phi_{CN}(0.1) = \phi^{\circ\circ} - 2\Delta t\phi^{\circ} = 1 - 2 \times 0.05 \times 0.95238 = \boxed{0.90476} \\
&\phi_{CN}(0.15) = 0.95238 - 2 \times 0.05 \times 0.90476 = 0.861904 \\
&\phi_{CN}(0.2) = 0.90476 - 2 \times 0.05 \times 0.861904 = \boxed{0.81857} \\
&\phi_{CN}(0.25) = 0.861904 - 2 \times 0.05 \times 0.81857 = 0.780047 \\
&\phi_{CN}(0.3) = 0.81857 - 2 \times 0.05 \times 0.780047 = \boxed{0.74056}
\end{aligned}\right\}$$

$$\Rightarrow \begin{cases} error_{CN}(0.1) = 4 \times 10^{-5} \\ error_{CN}(0.2) = 1.3 \times 10^{-4} \\ error_{CN}(0.3) = 2.4 \times 10^{-4} \end{cases}$$

It is clear that the solution error indicates second order accuracy. The slight differences between the error values obtained here and those reported in example 1 are due to the number of decimal values carried during computations.

Numerical solution using the SOUE scheme

$$\left.\begin{aligned}
&\phi_{EU}(0.05) = \frac{1}{1+0.05}\phi(0) = 0.9524 \\
&\phi_{SOUE}(0.1) = \frac{4\phi^{\circ} - \phi^{\circ\circ}}{3+2\Delta t} = \frac{4 \times 0.9524 - 1}{3.1} = \boxed{0.90632} \\
&\phi_{SOUE}(0.15) = (4 \times 0.90632 - 0.9524)/3.1 = 0.86219 \\
&\phi_{SOUE}(0.2) = (4 \times 0.86219 - 0.90632)/3.1 = \boxed{0.82014} \\
&\phi_{SOUE}(0.25) = (4 \times 0.82014 - 0.86219)/3.1 = 0.780119 \\
&\phi_{SOUE}(0.3) = (4 \times 0.780119 - 0.82014)/3.1 = \boxed{0.74204}
\end{aligned}\right\}$$

$$\Rightarrow \begin{cases} error_{SOUE}(0.1) = 1.5 \times 10^{-3} \\ error_{SOUE}(0.2) = 1.44 \times 10^{-3} \\ error_{SOUE}(0.3) = 1.24 \times 10^{-3} \end{cases}$$

The solution is second order accurate, however it is less accurate than the CN solution.

13.4 Non-Uniform Time Steps

So far a uniform time step was considered. In practical applications it is common to use variable time steps mainly to reduce the computational cost by selecting, at every time step, the maximum allowable time step value that does not violate the CFL condition.

For first order schemes, the discretization is not affected by whether the time step is variable or constant. The situation is different for second order transient schemes since they use a stencil involving two time step values. For the case of the two step implementation of the Crank-Nicolson transient scheme nothing changes except that for each of the two steps a different time step is used. This affects the accuracy as the spatial derivative is no longer at the center of the temporal element. For other second order schemes, the interpolation profile has to be modified to account for the non equal time steps. In what follows a non uniform transient grid is used in the discretization of the transient term for the standard CN [2] and the SOUE [4, 19, 20] schemes. While the finite volume and finite difference methods yield equivalent algebraic relations in a uniform grid, this is not the case for variable time steps as demonstrated in the derivations to follow.

13.4.1 Non-Uniform Time Steps with the Finite Difference Approach

13.4.1.1 Crank-Nicolson Scheme

The CN scheme with non uniform time steps is derived, as shown in Fig. 13.17, by expressing the values of $\rho\phi$ at times $t + \Delta t$ and $t - \Delta t^{\circ}$ in terms of its value and the values of its derivatives at time t using Taylor series as

$$(\rho\phi)^{t+\Delta t} = (\rho\phi)^{t} + \left.\frac{\partial(\rho\phi)}{\partial t}\right|_{t} \Delta t + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_{t} \frac{\Delta t^2}{2!} + \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_{t} \frac{\Delta t^3}{3!} + \cdots \tag{13.89}$$

$$(\rho\phi)^{t-\Delta t^{\circ}} = (\rho\phi)^{t} - \left.\frac{\partial(\rho\phi)}{\partial t}\right|_{t} \Delta t^{\circ} + \left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_{t} \frac{(\Delta t^{\circ})^2}{2!} - \left.\frac{\partial^3(\rho\phi)}{\partial t^3}\right|_{t} \frac{(\Delta t^{\circ})^3}{3!} + \cdots \tag{13.90}$$

Then, multiplying Eq. (13.89) by $(\Delta t^{\circ})^2$ and Eq. (13.90) by Δt^2 and subtracting the resulting equations from each other, an equation for the first derivative is obtained as

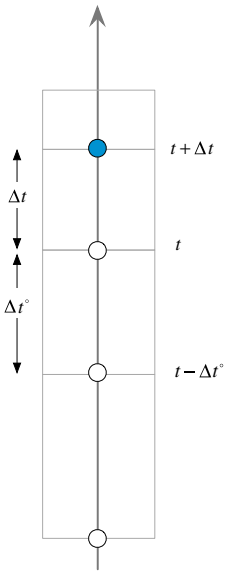


Fig. 13.17 The finite difference temporal mesh of the CN scheme with non uniform time steps

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t \approx \frac{(\Delta t^{\circ})^2(\rho\phi)^{t+\Delta t} - \left[(\Delta t^{\circ})^2 - \Delta t^2\right](\rho\phi)^{t} - \Delta t^2(\rho\phi)^{t-\Delta t}}{\left[\Delta t(\Delta t^{\circ})^2 + \Delta t^{\circ}\Delta t^2\right]} \tag{13.91}$$

Substituting the expression for the gradient from Eq. (13.91) in Eq. (13.3), the discretized equation for the CN scheme with non uniform time steps is given by

$$\frac{(\Delta t^{\circ})^2(\rho\phi) - \left[(\Delta t^{\circ})^2 - \Delta t^2\right](\rho\phi)^{\circ} - \Delta t^2(\rho\phi)^{\circ\circ}}{\Delta t^{\circ}\,\Delta t(\Delta t + \Delta t^{\circ})} V_C + L\left(\phi_C^{\circ}\right) = 0 \tag{13.92}$$

Expanding the spatial term, the final form of the algebraic equation becomes

$$\left(a_C^{\bullet} + a_C\right)\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C - a_C^{\circ}\phi_C^{\circ} - a_C^{\circ\circ}\phi_C^{\circ\circ} \tag{13.93}$$

with the time dependent coefficients computed from

$$\begin{aligned} a_C^{\bullet} &= \frac{\Delta t^{\circ}}{\Delta t(\Delta t + \Delta t^{\circ})}\rho_C V_C \\ a_C^{\circ} &= \frac{\Delta t - \Delta t^{\circ}}{\Delta t + \Delta t^{\circ}}\rho_C^{\circ} V_C \\ a_C^{\circ\circ} &= \frac{\Delta t}{\Delta t^{\circ}(\Delta t + \Delta t^{\circ})}\rho_C^{\circ\circ} V_C \end{aligned} \tag{13.94}$$

For uniform time steps, the coefficients in Eq. (13.39) are recovered.

13.4.2 Adams-Moulton (or SOUE) Scheme

Referring to Fig. 13.18, the Adams-Moulton scheme, also denoted by the SOUE scheme, with non uniform time steps is derived by expressing the values of the dependent variable ϕ at times $t - \Delta t$ and $t - \Delta t - \Delta t^\circ$ in terms of its value and the values of its derivatives at time t using Taylor series as

$$(\rho\phi)^{t-\Delta t} = (\rho\phi)^t - \Delta t \left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \frac{\Delta t^2}{2}\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t + O\left(\Delta t^3\right) \tag{13.95}$$

$$(\rho\phi)^{t-\Delta t-\Delta t^\circ} = (\rho\phi)^t - (\Delta t + \Delta t^\circ)\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t + \frac{(\Delta t + \Delta t^\circ)^2}{2}\left.\frac{\partial^2(\rho\phi)}{\partial t^2}\right|_t + O\left(\Delta t^3\right) \tag{13.96}$$

Multiplying Eq. (13.95) by $(\Delta t + \Delta t^\circ)^2/\Delta t^2$ and subtracting the resulting equation from Eq. (13.96), a second order representation of the first derivative (i.e., the SOUE scheme) is obtained as

$$\left.\frac{\partial(\rho\phi)}{\partial t}\right|_t = \frac{1}{\Delta t}\left[\left(1 + \frac{\Delta t}{\Delta t + \Delta t^\circ}\right)(\rho\phi)^t - \left(1 + \frac{\Delta t}{\Delta t^\circ}\right)(\rho\phi)^{t-\Delta t} + \left(\frac{\Delta t^2}{\Delta t^\circ(\Delta t + \Delta t^\circ)}\right)(\rho\phi)^{t-\Delta t-\Delta t^\circ}\right] \tag{13.97}$$

Substituting the expression for the gradient from Eq. (13.97) in Eq. (13.3), the discretized equation becomes

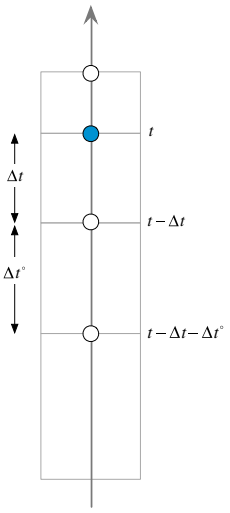


Fig. 13.18 The finite difference temporal mesh of the SOUE scheme with non uniform time steps

$$V_C\left(\frac{1}{\Delta t}+\frac{1}{\Delta t+\Delta t^{\circ}}\right)(\rho_C\phi_C)-V_C\left(\frac{1}{\Delta t}+\frac{1}{\Delta t^{\circ}}\right)(\rho_C\phi_C)^{\circ}+V_C\left(\frac{\Delta t}{\Delta t^{\circ}(\Delta t+\Delta t^{\circ})}\right)(\rho_C\phi_C)^{\circ\circ}+L\left(\phi_C^t\right)=0 \tag{13.98}$$

Expanding the spatial term, the final form of the algebraic equation is written as

$$\left(a_C^{\bullet}+a_C\right)\phi_C+\sum_{F\sim NB(C)}a_F\phi_F=b_C-a_C^{\circ}\phi_C^{\circ}-a_C^{\circ\circ}\phi_C^{\circ\circ} \tag{13.99}$$

with the time dependent coefficients obtained from

$$\begin{aligned}a_C^{\bullet}&=\left(\frac{1}{\Delta t}+\frac{1}{\Delta t+\Delta t^{\circ}}\right)\rho_C V_C\\ a_C^{\circ}&=-\left(\frac{1}{\Delta t}+\frac{1}{\Delta t^{\circ}}\right)\rho_C^{\circ}V_C\\ a_C^{\circ\circ}&=\frac{\Delta t}{\Delta t^{\circ}(\Delta t+\Delta t^{\circ})}\rho_C^{\circ\circ}V_C\end{aligned} \tag{13.100}$$

For uniform time steps the coefficients given in Eq. (13.54) are recovered.

13.4.3 Non-Uniform Time Steps with the Finite Volume Approach

Following the terminology used with the FVM, the size of a temporal element is denoted by Δt, while the distance between the centroids of two consecutive temporal elements is designated by δt. For uniform time steps both are equal and the time between two consecutive computed fields is $\Delta t=\delta t$ for both the finite difference and finite volume methods. For non-uniform time steps the time remains Δt for the finite difference method, however it becomes $\delta t=(\Delta t+\Delta t^{\circ})/2$ for the finite volume method leading to different formulations.

As for the finite difference method, with non-uniform time steps, the current and old time step values affect the scheme interpolation profile and hence its finite volume discretization. This is similar to writing the profile for a convection scheme over a structured non-uniform grid. The procedure used will be illustrated by considering the CN and SOUE schemes. Extension to other profiles is straightforward.

13.4.4 Crank-Nicolson Scheme

The CN scheme is obtained by calculating the value of $\rho\phi$ at an interface as the average of the $\rho\phi$ values at the main points straddling the interface (Fig. 13.19), i.e.,

$$\begin{aligned}(\rho_C\phi_C)^{t-\Delta t/2} &= \frac{\Delta t^\circ}{\Delta t+\Delta t^\circ}(\rho_C\phi_C)^t + \frac{\Delta t}{\Delta t+\Delta t^\circ}(\rho_C\phi_C)^{t-(\Delta t^\circ+\Delta t)/2}\\ (\rho_C\phi_C)^{t-\Delta t/2-\Delta t^\circ} &= \frac{\Delta t^{\circ\circ}}{\Delta t^\circ+\Delta t^{\circ\circ}}(\rho_C\phi_C)^{t-(\Delta t^\circ+\Delta t)/2} + \frac{\Delta t^\circ}{\Delta t^\circ+\Delta t^{\circ\circ}}(\rho_C\phi_C)^{t-\Delta t^\circ-(\Delta t+\Delta t^{\circ\circ})/2}\end{aligned} \tag{13.101}$$

Substituting in Eq. (13.57), the discretized $\rho\phi$ field equation is obtained as

$$\begin{aligned}&\frac{\Delta t^\circ}{\Delta t+\Delta t^\circ}\frac{V_C}{\Delta t}(\rho_C\phi_C) + \left(\frac{\Delta t}{\Delta t+\Delta t^\circ} - \frac{\Delta t^{\circ\circ}}{\Delta t^\circ+\Delta t^{\circ\circ}}\right)\frac{V_C}{\Delta t}(\rho_C\phi_C)^\circ\\ &\quad - \frac{\Delta t^\circ}{\Delta t^\circ+\Delta t^{\circ\circ}}\frac{V_C}{\Delta t}(\rho_C\phi_C)^{\circ\circ} + L\left(\phi_C^\circ\right) = 0\end{aligned} \tag{13.102}$$

The linearization coefficients for the CN scheme with non uniform time steps are inferred to be

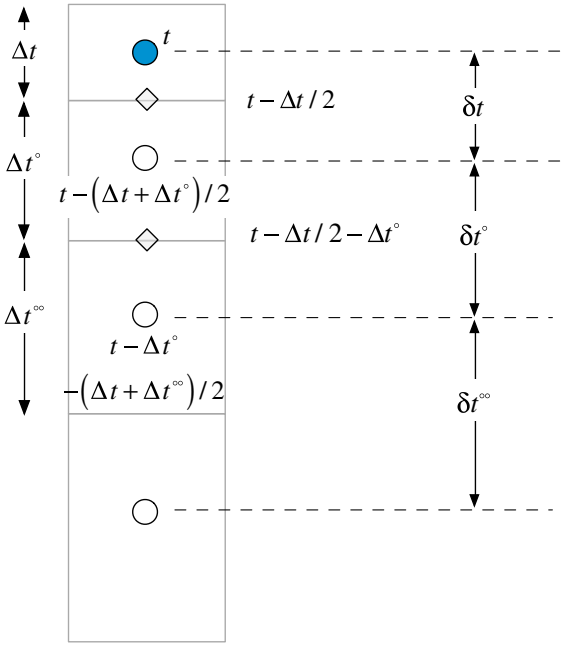


Fig. 13.19 The finite volume temporal mesh of the CN scheme with non uniform time steps

$$
\begin{aligned}
FluxC &= \frac{\Delta t^{\circ}}{\Delta t + \Delta t^{\circ}} \frac{\rho_C V_C}{\Delta t} \\
FluxC^{\circ} &= \left(\frac{\Delta t}{\Delta t + \Delta t^{\circ}} - \frac{\Delta t^{\circ\circ}}{\Delta t^{\circ} + \Delta t^{\circ\circ}} \right) \frac{\rho_C^{\circ} V_C}{\Delta t} \\
FluxV &= -\frac{\Delta t^{\circ\circ}}{\Delta t^{\circ} + \Delta t^{\circ\circ}} \frac{\rho_C^{\circ\circ} V_C \phi_C^{\circ\circ}}{\Delta t}
\end{aligned}
\tag{13.103}
$$

As in the constant time step case, the method is explicit necessitating storing values of the two previous time steps. Moreover the uniform time steps formulation can be recovered by setting $\Delta t = \Delta t^{\circ} = \Delta t^{\circ\circ}$ in Eqs. (13.102) and (13.103).

13.4.5 Adams-Moulton (or SOUE) Scheme

With the second-order "upwind" interpolation profile given by Eq. (11.84), the interface $\rho\phi$ values at the faces $t + \Delta t/2$ and $t - \Delta t/2$ displayed in Fig. 13.20 are found to be

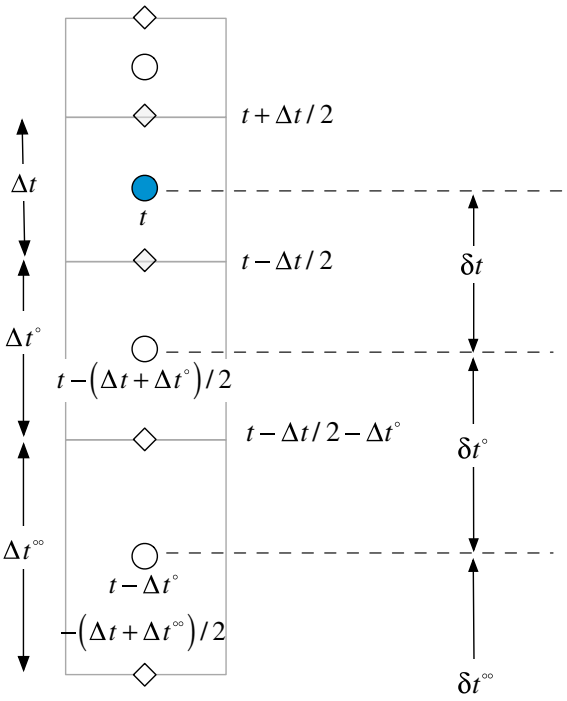


Fig. 13.20 The finite volume temporal mesh of the SOUE scheme with non uniform time steps

$$(\rho\phi)^{t+\Delta t/2} = (\rho\phi)^{t} + \left[(\rho\phi)^{t} - (\rho\phi)^{t-(\Delta t+\Delta t^{\circ})/2}\right]\frac{\Delta t}{\Delta t + \Delta t^{\circ}}$$
$$(\rho\phi)^{t-\Delta t/2} = (\rho\phi)^{t-(\Delta t+\Delta t^{\circ})/2} + \left[(\rho\phi)^{t-(\Delta t+\Delta t^{\circ})/2} - (\rho\phi)^{t-\Delta t^{\circ}-(\Delta t+\Delta t^{\circ\circ})/2}\right]\frac{\Delta t^{\circ}}{\Delta t^{\circ} + \Delta t^{\circ\circ}} \tag{13.104}$$

Using this profile approximation, the discretized form of Eq. (13.1) over the element C shown in Fig. 13.2 is obtained by substituting Eq. (13.104) in Eq. (13.57) and is given by

$$\left(1 + \frac{\Delta t}{\Delta t + \Delta t^{\circ}}\right)\frac{V_C}{\Delta t}(\rho_C\phi_C) - \left(1 + \frac{\Delta t}{\Delta t + \Delta t^{\circ}} + \frac{\Delta t^{\circ}}{\Delta t^{\circ} + \Delta t^{\circ\circ}}\right)\frac{V_C}{\Delta t}(\rho_C\phi_C)^{\circ}$$
$$+ \frac{\Delta t^{\circ}}{\Delta t^{\circ} + \Delta t^{\circ\circ}}\frac{V_C}{\Delta t}(\rho_C\phi_C)^{\circ\circ} + L(\phi_C) = 0 \tag{13.105}$$

The linearization coefficients for the SOUE scheme with non uniform time steps are inferred to be

$$\begin{aligned} FluxC &= \left(\frac{1}{\Delta t} + \frac{1}{\Delta t + \Delta t^{\circ}}\right)\rho_C V_C \\ FluxC^{\circ} &= -\left(\frac{1}{\Delta t} + \frac{1}{\Delta t + \Delta t^{\circ}} + \frac{\Delta t^{\circ}/\Delta t}{\Delta t^{\circ} + \Delta t^{\circ\circ}}\right)\rho_C^{\circ} V_C \\ FluxV &= \left(\frac{\Delta t^{\circ}/\Delta t}{\Delta t^{\circ} + \Delta t^{\circ\circ}}\right)\rho_C^{\circ\circ} V_C \phi_C^{\circ\circ} \end{aligned} \tag{13.106}$$

Similar to the constant time step case, the method is implicit as it requires solving a system of equations to obtain the ϕ field at every time step. The uniform time step form of the equation given by Eq. (13.80) is obtained by setting $\Delta t = \Delta t^{\circ} = \Delta t^{\circ\circ}$ in Eq. (13.105).

13.5 Computational Pointers

13.5.1 uFVM

The discretization of the transient term in uFVM follows the finite volume method implemented within an implicit framework. The assembly of the transient fluxes resulting from the first order backward Euler scheme is shown in Listing 13.1.

```
%
theDensityField = cfdGetMeshField(['Density' theFluidTag]);
density = theDensityField.phi(iElements);
density_old = theDensityField.phi_old(iElements);

volumes = [theMesh.elements(iElements).volume]';

theFluxes.FLUXCE(iElements)    =   volumes .* density / dt;
theFluxes.FLUXCEOLD(iElements) = - volumes .* density_old / dt;
theFluxes.FLUXTE(iElements)    = theFluxes.FLUXCE .* phi;
theFluxes.FLUXTEOLD(iElements) = theFluxes.FLUXCEOLD .* phi_old ;
```

Listing 13.1 Assembly of the transient fluxes resulting from the implicit Euler scheme

13.5.2 OpenFOAM®

In OpenFOAM®, explicit and implicit time derivatives [21] are defined via the namespaces fvm, fvc and the corresponding functions **fvc::ddt(rho, phi)** and **fvm::ddt(rho, phi)**, respectively. Moreover the first and second order upwind Euler schemes in addition to the second order Crank-Nicholson scheme are available, with the latter implemented following the two-step approach.

The files of the transient schemes are located in the directory "$FOAM_SRC/finiteVolume/finiteVolume/ddtSchemes". A base class denoted by **ddtScheme <Type>** is defined from which all time discretization schemes have to be derived.

The first order Euler scheme is implemented in the class **EulerDdtScheme.** The class is declared on top of the base class **ddtScheme <Type>**, as shown in Listing 13.2.

```
template<class Type>
class EulerDdtScheme
:
    public ddtScheme<Type>
```

Listing 13.2 Decleration of the EulerDdtScheme class

The implementation of the associated **fvc** and **fvm** namespaces are defined in the file **EulerDdtScheme.C.** The implicit evaluation of the Euler scheme is defined via the following function in Listing 13.3:

```
template<class Type>
tmp<fvMatrix<Type> >
EulerDdtScheme<Type>::fvmDdt
(
    const volScalarField& rho,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
```

Listing 13.3 Definition of the function needed for implicit solution using the Euler scheme

As depicted in Listing 13.4, the first step in this function is the definition of the **fvMatrix** in which only the diagonal coefficient vector is filled.

```
{
    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            vf.dimensions()*dimVol/dimTime
        )
    );
    fvMatrix<Type>& fvm = tfvm();
```

Listing 13.4 Definition of the fvMatrix

After allocating the needed space for storing the diagonal coefficients and sources, the values to be stored are defined and computed. As shown in Listing 13.5, this is accomplished by first defining the reciprocal of the time step as **rDeltaT**, then calculating $a_t = \rho_C V_C / \Delta t$ and storing its value in the **fvm.diag()** vector. The source contribution is computed as the product of $a_t^o = -\rho_C^o V_C / \Delta t$ and the old value of **vf** and stored in the **fvm.source()** vector, where **vf** is the generic variable used while applying the time scheme.

```
  scalar rDeltaT = 1.0/mesh().time().deltaTValue();
  fvm.diag() = rDeltaT*rho*mesh().V();
  fvm.source()=rDeltaT*rho.oldTime()*vf.oldTime().internalField()*mesh
().V();
  return tfvm;
}
```

Listing 13.5 Calculation of the terms added to the diagonal and source vectors

The script in Listing 13.6 shows that OpenFOAM® allows explicit evaluation of the unsteady term using the Euler scheme. In this case, a **GeometricField** object containing the value of $(\rho\phi - \rho^o\phi^o)/\Delta t$ is returned (here the value is per unit volume).

```
tmp<GeometricField<Type, fvPatchField, volMesh> >
EulerDdtScheme<Type>::fvcDdt
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
        return tmp<GeometricField<Type, fvPatchField, volMesh> >
        (
            new GeometricField<Type, fvPatchField, volMesh>
            (
                ddtIOobject,
                rDeltaT*(vf*rho - vf.oldTime()*rho.oldTime())
            )
        );
```

Listing 13.6 Explicit calculation of the unsteady term using the Euler scheme

The SOUE scheme is implemented in OpenFOAM® under the class **backwardDdtScheme.** The definition of the class is on top of the base class, as shown in Listing 13.7.

```
template<class Type>
class backwardDdtScheme
:
    public fv::ddtScheme<Type>
{
    // Private Member Functions

        //- Return the current time-step
        scalar deltaT_() const;

        //- Return the previous time-step
        scalar deltaT0_() const;
```

Listing 13.7 Script used to define the **backwardDdtScheme** class for the implementation of the SOUE scheme

In this case OpenFOAM® uses information from the current and the previous time steps. In the general case, the time steps are different necessitating the use of two variables to store their values.

The implicit time discretization is defined in a way similar to the first order transient scheme using the function shown in Listing 13.8 through Listing 13.10.

```
template<class Type>
tmp<fvMatrix<Type> >
backwardDdtScheme<Type>::fvmDdt
(
    const volScalarField& rho,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
```

Listing 13.8 Script used to define the implicit discretization using the SOUE scheme

The part of the function displayed in Listing 13.9 calculates the transient coefficients that multiply the current, old, and old-old values of the dependent variable and stores the contribution to the diagonal coefficients in **fvm.diag()** vector.

```
    scalar rDeltaT = 1.0/deltaT_();

    scalar deltaT = deltaT_();
    scalar deltaT0 = deltaT0_(vf);

    scalar coefft   = 1 + deltaT/(deltaT + deltaT0);
    scalar coefft00 = deltaT*deltaT/(deltaT0*(deltaT + deltaT0));
    scalar coefft0  = coefft + coefft00;

    fvm.diag() = (coefft*rDeltaT)*rho.internalField()*mesh().V();
```

Listing 13.9 Script used to calculate the unsteady coefficients and to store the contribution to the diagonal coefficients in the **fvm.diag()** vector

In the last part of the function shown in Listing 13.10, the contribution of the unsteady term is computed and stored in the fvm.source() vector.

```
fvm.source() = rDeltaT*mesh().V()*
(
    coefft0*rho.oldTime().internalField()
   *vf.oldTime().internalField()
  - coefft00*rho.oldTime().oldTime().internalField()
   *vf.oldTime().oldTime().internalField()
);
```

Listing 13.10 Script used to calculate and store contribution to the source in the **fvm.source()** vector

By comparing the coefficients of the SOUE scheme with the ones given in Eq. (13.100) it is easily seen that OpenFOAM® adopts a finite difference approach for the discretization of the unsteady term whereby the time derivative is approximated via Taylor series expansions.

13.6 Closure

The chapter covered the discretization of the transient term in the unsteady conservation equation. For that purpose, two general methodologies were discussed. One method is based on a finite difference discretization while the other follows a finite volume approach in which the conservation equation is integrated over a temporal element. The first order fully implicit and fully explicit transient schemes were presented. The formulation of higher order approximations was also investigated. This included the CN and the SOUE schemes for uniform and non uniform time steps. The next chapter is devoted to the discretization of the source term, relaxation of the algebraic system of equations, and other related details.

13.7 Exercises

Exercise 1
Transient heat transfer for the one dimensional body shown in Fig. 13.21 is governed by the following energy equation:

$$\frac{\partial\left(\rho c_p T\right)}{\partial t} = \frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right)$$

The body is insulated at one end while subjected to convective heat transfer at the second end. Other parameters include $T_R = 330$ K, $h_R = 400$ W/m^{2}K, $k = 55$ W/mK, $\rho = 7000\,\text{kg/m}^3$, and $c_p = 400$ J/Kg K.

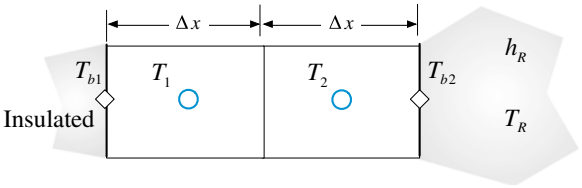


Fig. 13.21 One-dimensional domain used for Exercise 1

(a) Compute the temperature field using the Euler Explicit method for three time steps. Note that the initial temperature is $T_i = 273$ K with $\Delta t = 20$ s and $\Delta x = 0.015$ m.
(b) Repeat part a using an implicit Euler scheme.
(c) Explain the difference in temperatures between the two methods at time $t = 60$ s.

Exercise 2

The body described in Exercise 1 is now insulated at one end while subjected to a Dirichlet boundary condition at the second end. The initial and boundary conditions are $T_i = 273$ K, $T_{b2} = 330$ K while values of other parameters are given by

$$\Delta x = 0.015 \text{ m},\ k = 55 \text{ W/mK},\ \rho = 7000 \text{ kg/m}^3 \text{ and } c_p = 400 \text{ J/Kg K}.$$

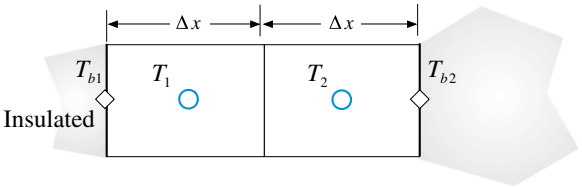


Fig. 13.22 One-dimensional domain used for Exercise 2

Compute the temperature field for three time steps using:

(a) The Adams-Moulton method with uniform time steps ($\Delta t = 20$ s).
(b) The finite difference form of the Adams-Moulton method with non-uniform time steps ($\Delta t_1 = 10$ s, $\Delta t_2 = 20$ s, and $\Delta t_3 = 30$ s).
(c) The finite volume form of the Adams-Moulton method with non-uniform time steps ($\Delta t_1 = 10$ s, $\Delta t_2 = 20$ s, and $\Delta t_3 = 30$ s). (Fig. 13.22).

Use the implicit Euler scheme for the first time step.

Exercise 3

The body described in Exercise 1 is again subjected to a Dirichlet boundary condition at one end and to a convective heat transfer at the second end. The parameters involved are $\Delta x = 0.015$ m, $T_i = 273$ K, $T_{b1} = 260$ K, $T_R = 330$ K, $h_R = 400$ W/m^{2}K, $k = 55$ W/mK, $\rho = 7000$ kg/m^3, and $c_p = 400$ J/Kg K (Fig. 13.23).

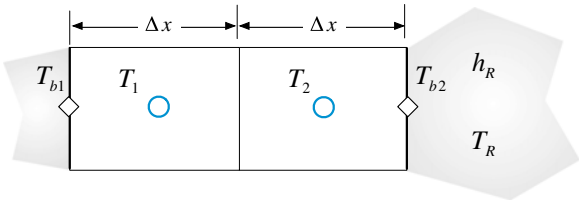


Fig. 13.23 One-dimensional domain used for Exercise 3

Compute the temperature field for three time steps using:

(a) The Crank-Nicolson method with uniform time steps ($\Delta t = 20$ s).
(b) The finite difference form of the Crank-Nicolson method with non-uniform time steps ($\Delta t_1 = 10$ s, $\Delta t_2 = 20$ s, and $\Delta t_3 = 30$ s).
(c) The finite volume form of the Crank-Nicolson method with non-uniform time steps ($\Delta t_1 = 10$ s, $\Delta t_2 = 20$ s, and $\Delta t_3 = 30$ s).

Use the implicit Euler scheme for the first time step.

Exercise 4

Consider the following equation defined over the one dimensional grid shown in Fig. 13.24:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot \Gamma \nabla \phi - \beta \phi$$

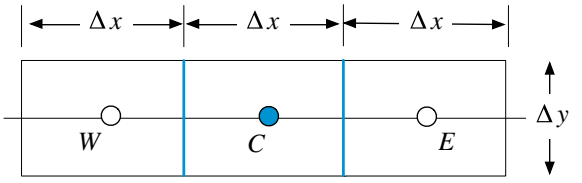


Fig. 13.24 One dimensional domain used for Exercise 4

(a) Derive the algebraic equation for element C. Use a first order Euler Explicit scheme for the transient term and linearize the source term given that β is positive.
(b) Is there a step limitation for the equation derived in (a)? If so derive its expression in terms of the appropriate variables.

Exercise 5
Use the implicit backward Euler scheme to integrate in time the linear advection equation given by

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0\, u > 0$$

and the second order central difference approximation for the spatial derivative.

(a) Derive the discretized equation.
(b) Find the accuracy of the scheme
(c) Determine the stability of the scheme.

Exercise 6
Use the fully implicit Euler scheme in time and the central difference scheme in space to discretize the one dimensional convection diffusion heat equation given by

$$\frac{\partial \left(\rho c_p T\right)}{\partial t} + \frac{\partial (\rho u T)}{\partial x} = \frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right) + S$$

over a uniform mesh of spacing Δx and write it down in the standard form.

There are two issues that should be considered when choosing the time step: stability and accuracy. What are the limits for the time step of the two schemes to achieve stable and accurate solutions? Are these limits similar for both stability and accuracy?

Exercise 7 (OpenFOAM®)
List from Doxygen [22] all derived classes of the *ddtScheme <Type>* class.

Exercise 8 (OpenFOAM®)
Find in OpenFOAM® the fvm implementation of the first order implicit Euler scheme. Compare the implemented algorithm with Eq. (13.28) and the contribution to the matrix of coefficients with Eq. (13.62).

Exercise 9 (OpenFOAM®)
Compare in OpenFOAM® the fvm implementation of the second order Crank-Nicolson transient scheme with Eqs. (13.43) and (13.44). The C file is located in "*$FOAM_SRC/finiteVolume/finiteVolume/ddtSchemes/CrankNicolsonDdtScheme /CrankNicolsonDdtScheme.C*". Hint: In the fvm member function, just check the *if* statement when *mesh().moving()* is false.

References

1. Faires JD, Burden RL (1993) Numerical methods. PWS, Boston, pp 152–153
2. Crank J, Nicolson P (1947) A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. Proc Cambr Phil Soc 43:50–67
3. Shyy W (1985) A study of finite difference approximations to steady state convection dominated flows. J Comput Phys 57:415–438
4. Moukalled F, Darwish M (2012) Transient schemes for capturing interfaces of free-surface flows. Numer Heat Transf Part B Fundam 61(3):171–203
5. Ascher U, Ruuth S, Spiteri RJ (1997) Implicit-explicit runge-kutta methods for time-dependent partial differential equations. Appl Numer Math 25:151–167
6. Ames WF (1977) Numerical methods for partial differential equations. Academic Press, Orlando
7. Milne WE (1953) Numerical solution of differential equations. Wiley, New York
8. Richtmyer RD (1967) Difference methods for initial value problems, 2nd edn. Wiley, New York
9. Birkhoff G, Rota G (1989) Ordinary differential equations. Wiley, New York
10. Burden R, Faires JD (2010) Numerical analysis, 9th edn. Brooks, Cole
11. Chapra S, Canale R (2014) Numerical methods for engineers. 7th ed., McGraw Hill, New York
12. Cheney W, Kincaid D (2013) Numerical mathematics and computing, 7th edn. Brooks/Cole, Boston
13. Courant R, Friedrichs K, Lewy H (1928) Über die partiellen Differenzengleichungen der mathematischen Physik. Math Ann (in German) 100:32–74
14. Patankar SV (1980) Numerical heat transfer and fluid flow, Hemisphere, New York
15. Peinado J, Ibáñez J, E. Arias E, V. Hernández V (2010) Adams–Bashforth and Adams–Moulton methods for solving differential Riccati equations. Comput Math With Appl 60 (11):3032–3045
16. Ferziger JH, Peric M (2013) Computational methods for fluid dynamics, 3rd edn. Springer, Germany
17. Courant R, Isaacson E, Rees M (1952) On the solution of nonlinear hyperbolic differential equations by finite differences. Commun Pure Appl Math 5:243–255
18. Darwish M, Moukalled F (2006) Convective schemes for capturing interfaces of free-surface flows on unstructured grids. Numer Heat Transf Part B Fundam 49(1):19–42
19. Darwish M, Moukalled F (1994) Normalized variable and space formulation methodology for high-resolution schemes. Numer Heat Transf Part B Fundam 26(1):79–96
20. Leonard BP (1981) A survey of finite differences with unwinding for numerical modeling of the incompressible convection diffusion equation. In Taylor C, Morgan K (eds.) Computational techniques in transient and turbulent flow, Pineridge Press, Swansea, UK, 2:1–35
21. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org
22. OpenFOAM Doxygen, 2015 Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 14
Discretization of the Source Term, Relaxation, and Other Details

Abstract This chapter is devoted to a number of "small" numerical details that may have "big" effects on the solution behavior. First the treatment of the source term in the general case when it is solution dependent (i.e., when $Q^{\phi} = Q^{\phi}(\phi)$) is examined. The source is linearized in terms of the dependent variable and split into two parts, one treated explicitly and the second treated implicitly. This is followed by a discussion of explicit and implicit techniques for under-relaxing the algebraic equations. Several implicit under-relaxing approaches are presented, starting with the well known implicit under relaxation method of Patankar (Numerical heat transfer and fluid flow, 1980) [1], the E-factor method of van Doormaal and Raithby (Numerical Heat Transfer 7:147–163, 1984) [2], and the false transient approach of Mallinson and de Vahl Davis (Journal of Computational Physics 12 (4):435–461, 1973) [3]. Then the residual form of the discretized algebraic equation is introduced. The chapter ends with the presentation of convergence indicators used to evaluate the solution convergence status.

14.1 Source Term Discretization

Source terms (sink and source) appear in the governing equations of many flow and transport phenomena problems. Examples include the equations of turbulence models, chemical reactions, radiation heat transfer, mass transfer, and multiphase flows, to cite a few. These source terms affect not only the physics of the problem, but also the numerical stability of computations. However, if properly handled, source terms may yield improvement in robustness. A general recommendation is to treat negative values (sinks) implicitly, while positive values (sources) should be evaluated explicitly.

The treatment of source terms can be clarified by considering the discretized form of the general conservation equation for the variable ϕ over the element of centroid C and volume V_C with a source explicitly displayed (Fig. 14.1). This equation is given by

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_14

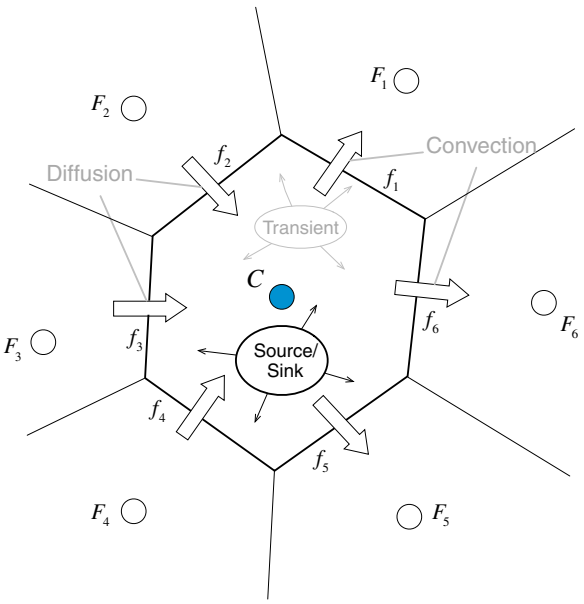


Fig. 14.1 An element ***C*** with a source term Q^{ϕ}

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = Q_C^{\phi}V_C \tag{14.1}$$

where $Q_C^{\phi}V_C$ represents the source term integrated over the element C.

In general the source term is a function of the dependent variable ϕ with its functional relationship expressed as

$$Q_C^{\phi} = Q(\phi_C) \tag{14.2}$$

In this form the source can be explicitly calculated based on the available ϕ values, which in an iterative process represent values from the previous iteration. Whereas this approach is acceptable if the value of Q_C^{ϕ} is constant or relatively small, when the variation in Q_C^{ϕ} is large in comparison with other terms in the equation the rate of convergence can be negatively affected. In such situations, the rate of convergence may be improved by linearizing Q_C^{ϕ} using a Taylor-like series expansion. Denoting values at the previous iteration with a superscript *, the value of the source term Q_C^{ϕ} at the current iteration can be expressed as

$$\begin{aligned} Q(\phi_C) &= Q(\phi_C^*) + \left(\frac{\partial Q}{\partial \phi_C}\right)^* (\phi_C - \phi_C^*) \\ &= \underbrace{\left(\frac{\partial Q}{\partial \phi_C}\right)^* \phi_C}_{\textit{Implicit part}} + \underbrace{Q(\phi_C^*) - \left(\frac{\partial Q}{\partial \phi_C}\right)^* \phi_C^*}_{\textit{Explicit part calculated based on values from previous iteration}} \end{aligned} \tag{14.3}$$

In a control volume context, the right hand side of Eq. (14.1) can be written as

$$\begin{aligned} Q_C^\phi V_C &= \iint_{V_C} Q^\phi dV \\ &= \iint_{V_C} \left(\frac{\partial Q_C^*}{\partial \phi_C}\phi_C\right) dV + \iint_{V_C} \left(Q_C^* - \frac{\partial Q_C^*}{\partial \phi_C}\phi_C^*\right) dV \\ &= \left(\frac{\partial Q_C^*}{\partial \phi_C} V_C\right)\phi_C + \left(Q_C^* - \frac{\partial Q_C^*}{\partial \phi_C}\phi_C^*\right) V_C \\ &= FluxC_C\ \phi_C + FluxV_C \end{aligned} \tag{14.4}$$

Substituting back in Eq. (14.1), the algebraic equation becomes

$$[a_C - FluxC_C]\phi_C + \sum_{F \sim NB(C)} a_F \phi_F = FluxV_C \tag{14.5}$$

In this formulation the implicit part of the source, $FluxC_C$ [defined in Eq. (14.3)], is required to be negative to guarantee diagonal dominance or else the Scarborough criterion may not be fulfilled causing divergence. In addition, for the case when the variable ϕ is positive-definite, the explicit part, $FluxV_C$ [Eq. (14.3)], must be positive to ensure positive ϕ predictions.

Example 1

In problems involving heat transfer by radiation, the source term in the energy equation takes the form

$$Q^T = A\left(T_\infty^4 - T^4\right)$$

where A is a constant, T is the temperature at any grid point, and T_∞ represents the non-varying ambient temperature. Integrate this source term over an element of centroid C and volume V_C, then linearize it using different alternatives and explain their consequences on convergence.

Solution

$$\int_{V_C} Q^T dV = Q_C^T V_C = FluxV_C + FluxC_C T_C$$

Several arbitrary choices can be selected to linearize Q^T.

Option 1

$$FluxC_C = 0$$
$$FluxV_C = A\left(T_\infty^4 - T_C^4\right)V_C$$

Adopting this alternative may cause the solution to diverge as it results in negative values for $FluxV_C$, whenever $T_C > T_\infty$, leading to possible unphysical negative absolute temperature values during the iterative process.

Option 2
Linearizing with respect to the value of temperature of the previous iteration T_C^*, the expanded form of the source term is obtained as

$$Q_C^T = \left(Q_C^T\right)^* + \left(\frac{dQ_C^T}{dT_C}\right)^* \left(T_C - T_C^*\right)$$
$$= A\left(T_\infty^{*4} - T_C^{*4}\right) - 4AT_C^{*3}\left(T_C - T_C^*\right)$$

Comparing with Eq. (14.9), $FluxC_C$ and $FluxV_C$ are found to be

$$FluxC_C = -4AT_C^{*3}V_C$$
$$FluxV_C = A\left(T_\infty^{*4} + 3T_C^{*4}\right)V_C$$

This is the ideal approach resulting in a positive $FluxV_C$ and a negative $FluxC_C$ and giving the best rate of convergence as the introduced implicitness in the solution is the optimum one.

14.2 Under-Relaxation of the Algebraic Equations

As described in previous chapters, the end product of the discretization process is a set of algebraic equations of the form given by Eq. (14.1), in which a_F refers to a neighboring coefficient (Fig. 14.2) representing the effect of the neighboring variable ϕ_F on the cell variable ϕ_C, b_C is the right hand side of the equation that usually includes the source terms and the effects of other variables, while a_C is the main

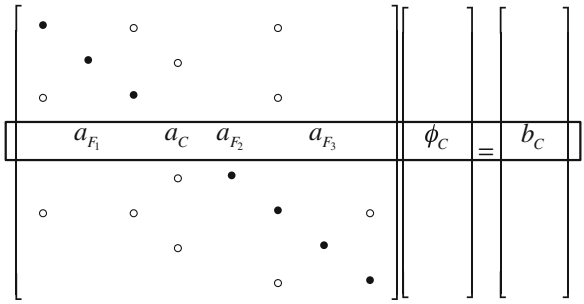


Fig. 14.2 Matrix representation of Eq. (14.1)

coefficient of the algebraic equation and contains the effects of various influences, including the spatial discretization effects, the transient effects, etc. The set of equations represented by Eq. (14.1) is usually diagonally dominant.

In the iterative solution of the system of algebraic equations it is often desirable to slow down the changes in the values of the dependent variable from iteration to iteration. This is needed to improve the convergence of non-linear problems but also to avoid divergence when starting with a guessed initial field that could be far from the solution. The non-linearities can arise because of the non-orthogonality of the grid system, the presence of source terms, the non-linear nature of the modeled equations, etc. One method commonly used to promote convergence by "slowing down" ("relaxing") the (sometimes excessive) changes made to the values of the variable during solution is the relaxation method. The standard relaxation method used in many CFD codes is the implicit under relaxation method of Patankar [1], which was briefly presented in Chap. 8. Other under-relaxation methods have been presented in the literature such as the E-Factor [2] relaxation method and the False transient method [3]. Van Doormaal and Raithby [2] have shown that these different relaxation methods are somewhat related and that the under-relaxation in any of the methods can be related to the under-relaxation in the other methods, as they all basically retard the effect of neighboring elements and sources on the under-relaxed element value. In other words, under-relaxation affects equally the source term in the concerned element and its spatial coefficients. Some of these relaxation methods are presented next.

14.2.1 Under-Relaxation Methods

Solution relaxation may be performed either explicitly after the solution at any iteration is obtained or implicitly by incorporating its effect into the equation before the solution is obtained. Both methods are outlined below.

14.2.2 Explicit Under-Relaxation

In the explicit under-relaxation method, at the end of every iteration after a new solution is obtained, all cells in the computational domain are visited and the predicted value $\left(\phi_C^{new,\,predicted}\right)$ in any cell C is modified according to

$$\phi_C^{new,\,used} = \phi_C^{old} + \lambda^{\phi}\left(\phi_C^{new,\,predicted} - \phi_C^{old}\right) \tag{14.6}$$

where λ^{ϕ} is the relaxation factor, which for both explicit and implicit relaxation can be interpreted according to its assigned value as follows:

1. A value of $\lambda^{\phi} < 1$ results in under-relaxation. This may slow down the speed of convergence but increases the stability of the calculation, i.e., it decreases the possibility of divergence or oscillations in the solution.
2. A value of $\lambda^{\phi} = 1$ corresponds to no relaxation. If applied, then the predicted values during an iteration are the ones used at the next iteration.
3. A value of $\lambda^{\phi} > 1$ leads to over-relaxation. It can sometimes be used to accelerate convergence but usually decreases the stability of the calculations.

Explicit under-relaxation is used to under-relax pressure in the SIMPLE algorithm, which will be introduced in the next chapter. Further, in problems where the fluid properties depend on the solution and are iteratively updated, explicit under-relaxation may be necessary to promote convergence. Examples include, but are not limited to, the turbulent viscosity in turbulent flows, the density in compressible flows, and computed interface values using HR schemes. In addition, it may be used to under-relax individual terms in the conservation equation such as the source term, and in some cases gradients of solution variables.

14.2.3 Implicit Under-Relaxation Methods

Several approaches in this category have been developed. The standard method that was presented in Chap. 8 is Patankar's approach [1], a summary of which is given here for completeness. Other methods include the E-factor approach and the false transient technique, which are also discussed.

14.2.3.1 Patankar's Under-Relaxation

As mentioned above, the iterative solution of a system of equations can be under-relaxed by introducing a relaxation factor λ^{ϕ} and expressed via Eq. (14.6). To simplify the notation used with implicit under-relaxation, Eq. (14.6) is modified to

$$\phi_C = \phi_C^* + \lambda^\phi \left(\phi_C^{new\ iteration} - \phi_C^*\right) \tag{14.7}$$

where ϕ_C^* is the value of ϕ_C from the previous iteration. In Patankar's relaxation approach, $\phi_C^{new\ iteration}$ in Eq. (14.7) is replaced by its equivalent expression from Eq. (14.1) to yield

$$\phi_C = \phi_C^* + \lambda^\phi \left(\left(\frac{-\sum\limits_{F \sim NB(C)} a_F \phi_F + b_C}{a_C} \right) - \phi_C^* \right) \tag{14.8}$$

re-arranging, the equation becomes

$$\frac{a_C}{\lambda^\phi}\phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C + \frac{\left(1 - \lambda^\phi\right)}{\lambda^\phi} a_C \phi_C^* \tag{14.9}$$

In Eq. (14.9) the relaxation factor λ^ϕ modifies the diagonal coefficient and the right hand side without modifying the equation mathematically. Since $\lambda^\phi < 1$, under relaxation increases the diagonal dominance of the algebraic system and enhances the stability of the iterative linear solver. This is an important advantage when compared to the explicit approach.

However it is worth noting that the implicit relaxation applies a relation that is proportional to the diagonal coefficient. Thus the relaxation will be larger for a larger diagonal coefficient, which translates into a larger relaxation of higher importance for smaller control volumes. This is demonstrated in the next section.

14.2.3.2 E-Factor Relaxation

The E-Factor method [2] is a reformulation of Patankar's method. It is derived by rewriting Eq. (14.1) in the following form:

$$a_C \phi_C = b_C - \sum_{F \sim NB(C)} a_F \phi_F \tag{14.10}$$

under-relaxing, the right hand side of Eq. (14.10) is transformed to

$$a_C \phi_C = \lambda^\phi \left(b_C - \sum_{F \sim NB(C)} a_F \phi_F \right) + \left(1 - \lambda^\phi\right) a_C \phi_C^* \tag{14.11}$$

Replacing the under-relaxation factor with $\frac{E^\phi}{1+E^\phi}$, Eq. (14.11) becomes

$$a_C\phi_C = \frac{E^\phi}{1+E^\phi}\left(b_C - \sum_{F\sim NB(C)} a_F\phi_F\right) + \left(1 - \frac{E^\phi}{1+E^\phi}\right)a_C\phi_C^* \tag{14.12}$$

which can be reformulated as

$$a_C\left(1+\frac{1}{E^\phi}\right)\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C + \frac{1}{E^\phi}a_C\phi_C^* \tag{14.13}$$

With this formulation the under-relaxation effect can be readily interpreted in term of some artificial transient time scale that advances ϕ_C at each solver iteration. The time step Δt can be shown to be proportional to the characteristic time interval Δt^* according to

$$\Delta t = E^\phi \Delta t^* \tag{14.14}$$

where

$$\Delta t^* = \frac{\rho_C V_C}{a_C} \tag{14.15}$$

In Eq. (14.15) ρ_C is the density of the fluid in cell C of volume V_C. The characteristic time interval is related to the time required to diffuse and convect a change of ϕ_C across the element. Thus the E-factor is equivalent to an element CFL number.

It is clear from Eq. (14.15) that the time step advancement of the E-Factor relaxation is dependent on the cell volume, with the solution in a smaller element advancing more slowly than in a coarser element. This can be detrimental to the convergence rate for a steady state solution since it is very common to use highly stretched elements with small volumes near boundaries, thus forcing a critical region in the computational domain to advance at a very small time step compared to the remainder of the domain. This is also a characteristic of the Patankar relaxation method.

The relation between E^ϕ and λ^ϕ can be shown to be

$$E^\phi = \frac{1}{1-\lambda^\phi}. \tag{14.16}$$

In general values of E^ϕ are chosen in the range of 4–10, corresponding to values between 0.75 and 0.9 for λ^ϕ.

Example 2

In the figure below an illustrative boundary mesh is shown. Elements A and D represent stretched elements near a wall boundary, with the volume ratio of about $V_{C_A}/V_{C_D} \approx 0.1$. *The value of the diagonal coefficients for such meshes are usually dominated by the diffusion coefficient and in this case would be around* $a_{C_A}/a_{C_D} \approx 2$ *since element A has a boundary face.*

Compute the relative pseudo transient time for elements A and D if an under-relaxation factor of 0.8 is applied (Fig. 14.3).

Fig. 14.3 Schematic of a boundary mesh with stretched elements

Solution

Solution starts by computing the equivalent E factor for the applied relaxation factor

$$E = \frac{1}{1-\lambda} = \frac{1}{1-0.8} = 5$$

thus the pseudo time step for each of the elements is

$$\begin{aligned}\Delta t &= E^{\phi}\Delta t^{*} \\ &= 5\frac{\rho_C V_C}{a_C}\end{aligned}$$

Therefore the relative pseudo time steps for elements A and D can be computed as

$$\frac{\Delta t_A}{\Delta t_B} = \left(\frac{V_{C_A}}{a_{C_A}}\right)/\left(\frac{V_{C_D}}{a_{C_D}}\right) = \left(\frac{V_{C_A}}{V_{C_D}}\right)\left(\frac{a_{C_D}}{a_{C_A}}\right) \approx 0.1\left(\frac{1}{2}\right) = 0.05$$

implying that the pseudo time step for element D is nearly 20 times that of element A.

14.2.3.3 False Transient Relaxation

The false transient relaxation method [3] is a modification of the Euler first order implicit transient method, wherein the previous iteration values are used instead of the old time step values. As in the Euler method the diagonal dominance of the algebraic equation is increased through the addition of the pseudo-transient term, $a_C^o \phi_C$, to the diagonal coefficient and the pseudo old time step term, $a_C^o \phi_C^*$, to the right hand side. With these modifications the equation becomes

$$\left(a_C + a_C^o\right)\phi_C + \sum_{F \sim NB(C)} a_F \phi_F = b_C + a_C^o \phi_C^* \tag{14.17}$$

where the coefficient a_C^o is computed as

$$a_C^o = \frac{\rho_C V_C}{\Delta t} \tag{14.18}$$

and is basically equal to the transient coefficient obtained from the first order implicit Euler discretization of the transient term, ρ_C is the density, V_C the element volume, and Δt a user-defined false time step. For large values of Δt, the added term is negligible, and under-relaxation effects are negligible. Therefore the solution of the equation is the same as the original unrelaxed one. For very small values of Δt, the value of a_C^o becomes large dominating other terms. The solution is heavily under-relaxed leading to minute change in the value of ϕ_C (i.e., $\phi_C \approx \phi_C^*$).

In addition to allowing the solution to advance consistently over the entire computational domain, the false transient method ensures the addition of a non-zero contribution to the diagonal coefficient even in extreme cases when the diagonal coefficient is zero.

There is no general rule for assigning optimum under-relaxation factors as values used in one case may not work properly for another case. In addition, different equations may be assigned different under-relaxation factors. Further, it is not necessary to use the same under-relaxation value throughout the computational domain. Furthermore, under-relaxation values may vary from iteration to iteration. For the SIMPLE algorithm to be described in Chaps. 15 and 16, Raithby and Schneider [4] derived an optimum relation between the under-relaxation factors for the velocity and pressure fields, which will be presented in the next chapter.

14.3 Residual Form of the Equation

The discretized algebraic equation has been written so far in its "direct" or "standard" form as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C, \tag{14.19}$$

which is the form used in OpenFOAM®.

Equation (14.19) can also be written in "correction" or "residual" form by rearranging its terms so as to solve for the correction needed to satisfy the equation. Thus if ϕ_C^* and ϕ_C' are the previous iteration value of ϕ_C and the correction needed to satisfy Eq. (14.19), respectively, then the solution is given by

$$\phi_C = \phi_C^* + \phi_C' \tag{14.20}$$

and Eq. (14.19) can be rewritten as

$$a_C\left(\phi_C^* + \phi_C'\right) + \sum_{F\sim NB(C)} a_F\left(\phi_F^* + \phi_F'\right) = b_C \tag{14.21}$$

or

$$a_C\phi_C' + \sum_{F\sim NB(C)} a_F\phi_F' = b_C - \left(a_C\phi_C^* + \sum_{F\sim NB(C)} a_F\phi_F^*\right) \tag{14.22}$$

Note that the right hand side of Eq. (14.22) represents the residual error of the equation for the field ϕ_C^*. Denoting this residual over element C by Res_C^ϕ, Eq. (14.22) becomes

$$a_C\phi_C' + \sum_{F\sim NB(C)} a_F\phi_F' = Res_C^\phi \tag{14.23}$$

It is worth noting that for the exact field Res_C^ϕ would be zero.

While mathematically equivalent to Eq. (14.19), Eq. (14.23) has one numerical advantage. In this form, numerical errors during the solution of the equation are slightly less than those associated with the standard form for cases when small variations are expected for large values of ϕ.

14.3.1 Residual Form of Patankar's Under-Relaxation

The residual form of the implicit Patankar relaxation equation is derived by rewriting Eq. (14.19) in the following form:

$$a_C\left(\phi_C^* + \phi_C'\right) = \lambda^\phi \left(b_C - \sum_{F \sim NB(C)} a_F\left(\phi_F^* + \phi_F'\right) \right) + \left(1 - \lambda^\phi\right) a_C \phi_C^* \quad (14.24)$$

which can be simplified to

$$a_C \phi_C' + \lambda^\phi \sum_{F \sim NB(C)} a_F \phi_F' = \lambda^\phi \left[b_C - \left(a_C \phi_C^* + \sum_{F \sim NB(C)} a_F \phi_F^* \right) \right] \quad (14.25)$$

Noticing that the right hand side of the above equation represents the residual of the original equation, Eq. (14.25) can be written as

$$\frac{a_C}{\lambda^\phi} \phi_C' + \sum_{F \sim NB(C)} a_F \phi_F' = Res_C^\phi \quad (14.26)$$

implying that under-relaxing the equation in residual form necessitates modifying only the diagonal coefficient.

14.4 Residuals and Solution Convergence

In any iterative solution process it is important to be able to determine when the solution can be considered good enough, or when the error can be estimated to be below a certain tolerance, or even to what precision the conservation equations have been satisfied. Having the tools to answer any of the above questions is an important ingredient for any CFD code. This can be rephrased as how to evaluate the degree of convergence of the solution field without knowing the final solution. To this end a number of indicators were proposed over the years, from the simple monitoring of a point at a location over a number of iterations, to the monitoring of an integrated value such as the drag coefficient, the total mass flow, the wall shear stress, and so on, or more commonly the monitoring of some type of equation residual. The challenge being that the method has to be used for a wide range of flow parameters and for a variety of geometries and boundary conditions.

14.4.1 Residuals

As a solution to the discretized system of equations represented by Eq. (14.1) is sought, the error in the balance equation is quantified by defining the element residual error as

$$Res_C^{\phi} = b_C - \left(a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F \right). \tag{14.27}$$

It is clear that when the solution is reached and the equation satisfied, the value of Res_C^{ϕ} will be zero. Using Res_C^{ϕ} a number of residual indicators for the whole domain can be derived as detailed next.

14.4.2 Absolute Residual

As defined by Eq. (14.27), the residual may be a positive or a negative quantity. Since the sign is immaterial, the absolute value of the Res_C^{ϕ}, denoted by R_C^{ϕ}, is used to decide whether a solution has converged or not. If R_C^{ϕ} decreases with iterations then the solution will be converging otherwise it will be diverging. The value of R_C^{ϕ} at point C is defined as

$$R_C^{\phi} = \left| b_C - \left(a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F \right) \right| \tag{14.28}$$

14.4.3 Maximum Residual

The solution is assumed to have converged when the maximum value of the absolute residuals, defined as,

$$R_{C,\max}^{\phi} = \max_{all\ cells} \left| b_C - \left(a_C \phi_C + \sum_{F \sim NB(C)} a_F \phi_F \right) \right| = \max_{all\ cells} \left(R_C^{\phi} \right) \tag{14.29}$$

over the domain drops below a vanishing quantity ε, i.e.,

$$R_{C,\max}^{\phi} \leq \varepsilon \Rightarrow \text{solution has converged.} \tag{14.30}$$

14.4.4 Root-Mean Square Residual

Another parameter used as a convergence indicator, is the average of the square root of the sum of the squares of the absolute residuals $R_{C,rms}^{\phi}$, mathematically given by

$$R^{\phi}_{C,\,rms} = \sqrt{\frac{\sum\limits_{C\sim all\ elements}\left(b_C - \left(a_C\phi_C + \sum\limits_{F\sim NB(C)} a_F\phi_F\right)\right)^2}{number\ of\ elements}} = \sqrt{\frac{\sum\limits_{C\sim all\ cells}\left(R^{\phi}_{C}\right)^2}{number\ of\ elements}}. \tag{14.31}$$

In this case the convergence criteria is written as

$$R^{\phi}_{C,\,rms} \leq \varepsilon \Rightarrow \text{solution has converged.} \tag{14.32}$$

14.4.5 Normalization of the Residual

The level of the absolute residual is a strong function of the variable ϕ. Therefore different variables result in different levels of R^{ϕ}_{C}. This makes it difficult to discern whether a solution has converged or not. In such cases a better insight can be gained through scaling the different residuals by dividing them by their respective maximum fluxes. Recalling that a_C represents the sum of the fluxes over the element, the residuals are scaled relative to the local value of the property ϕ to obtain a relative error by dividing them by the maximum value of $a_C\phi_C$ over the domain such that

$$R^{\phi}_{C,\,scaled} = \frac{\left|a_C\phi_C + \sum\limits_{F\sim NB(C)} a_F\phi_F - b_C\right|}{\max\limits_{all\ cells}|a_C\phi_C|}. \tag{14.33}$$

The solution is assumed to have converged when the maximum value of the scaled absolute residuals has dropped below a vanishing quantity ε, i.e.,

$$\max_{all\ cells}\left(R^{\phi}_{C,scaled}\right) \leq \varepsilon \Rightarrow \text{solution has converged} \tag{14.34}$$

It is common to require ε for scaled residuals to be of the order of 10^{-3} to 10^{-5} or less for convergence.

In addition to using either the absolute or scaled residuals, it is always insightful to monitor integrated quantities, as mentioned above, before concluding that the solution has converged. Always ensure proper convergence before declaring that a solution has converged as non-converged solutions can be misleading.

14.5 Computational Pointers

In this section, the source term linearization and relaxation techniques used in uFVM and OpenFOAM® are discussed.

14.5.1 uFVM

14.5.1.1 Source Term Linearization

The linearization and assembly of the source term in uFVM is implemented in the function cfdAssembleSourceTerm displayed in Listing 14.1. The function relies on the linearization set by the user, which has to supply the constant part of the source, Sb, and the linearized part, Sc. These terms are then added to FLUXV and FLUXC, as in Eq. (14.4).

It is worth noting that in uFVM the equations are solved in residual form, which explains the implementation of the total source in FLUXTE, rather than its constant part only.

```
theEquationField = cfdGetMeshField(theEquationName);
phi = theEquationField.phi(iElements);
%
Sb = cfdComputeFormulaAtLocale(theTerm.Sb,'Interior Elements')';
Sc = cfdComputeFormulaAtLocale(theTerm.Sc,'Interior Elements')';
%
volume = [theMesh.elements.volume];
%
% Assemble Source Term
%
pos = zeros(1,size(phi));
pos(Sc<0) = 1;
theFluxes.FLUXCE = -pos .* Sc .* volume;
theFluxes.FLUXTE = -(Sb +Sc .*phi) .* volume;
```

Listing 14.1 Linearization and implementation of the source term

14.5.1.2 Under-Relaxation

The implicit under-relaxation method of Patankar is implemented in ufvm. Since the equations are solved in residual form, their under-relaxation (Listing 14.2) requires modifying their diagonal coefficients only, as demonstrated by Eq. (14.26).

```
function cfdApplyURF(theEquationName)
%=====================================================
%  written by the CFD Group @ AUB, Fall 2006
%=====================================================
theEquation = cfdGetModel(theEquationName);
urf = theEquation.urf;
theCoefficients = cfdGetCoefficients;
theCoefficients.ac = theCoefficients.ac/urf;
cfdSetCoefficients(theCoefficients);
```

Listing 14.2 Implementation of Patankar's implicit under-relaxation method

14.5.2 OpenFOAM®

14.5.2.1 Source Term Linearization

In Sect. 14.1, the treatment of the source in the transport equation of a generic variable ϕ was discussed. The suggested linearization (or implicit treatment) can be viewed as imposing an artificial time step onto the matrix of coefficients, thus affecting the characteristic time of advancing the solution. In addition it increases diagonal dominance and enhances the solution robustness of the algebraic system of equations by allowing for an inherent relaxation to come into action when needed. That is, whenever the negative source term changes substantially, the system self-adapt the time step resolution in order to capture the characteristics of the modeled phenomenon. This is in contrast with the non-linearization approach (explicit treatment), which necessitates heavier under-relaxation of the system of equations with the relaxation factor being generally not optimal.

For the discretization of the source term OpenFOAM® [5] uses the implicit fvm:: and explicit fvc: : operators. Specifically the implementation of the fvc:: operator can be found in the directory "$FOAM_SRC/finiteVolume/finiteVolume/fvc/" in the corresponding files fvcSup.H and fvcSup.C. However it is usually the norm to define the explicit source term into the equation without the need for using the fvc:: operator. For example considering the case of a generic scalar transport equation with a source term that does not depend directly on the main variable and thus cannot be linearized, given by

$$\nabla \cdot (\rho U \phi) - \nabla \cdot (k \nabla(\phi)) = aU^2 \tag{14.35}$$

In OpenFOAM® Eq. (14.35) can be implemented, as shown in Listing 14.3,

```
fvMatrix<scalar> phiEqn
(
fvm::div(mDot,phi) - laplacian(k,phi) == a*magSqr(U)
);
```

Listing 14.3 Defining an explicit source term without invoking the **fvc** operator

with the source term not requiring any special wrapping function or operator.

The implementation of the **fvm::** functions can be found in the directory "$FOAM_SRC/finiteVolume/finiteVolume/fvm/" in the corresponding files **fvmSup.H** and **fvmSup.C**. The discretization of the linearized source is set in function **fvm::Sp**. As per Eq. (14.5), the implicit part of the source term is added as a contribution to the main diagonal of the coefficient matrix. For that purpose the function **Sp** is defined in Listing 14.4.

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type> >
Foam::fvm::Sp
(
    const DimensionedField<scalar, volMesh>& sp,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = vf.mesh();

    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            dimVol*sp.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm();

    fvm.diag() += mesh.V()*sp.field();

    return tfvm;
}
```

Listing 14.4 Script used for the definition and implementation of the **Sp** function

It is worth mentioning that the function **Sp** treats the source term irrespective of the sign of the slope of its linearized form. This means that for the case when the slope of the linearized term is positive the operation may lead to divergence of the solution algorithm as it destroys the diagonal dominance of the set of algebraic equations. Thus it is always important to ensure that the implicit treatment is used only when it results in a negative slope of the linearized term. For the case when the

slope of the linearized source term can assume, in different regions of the domain, different values (positive and negative), the negative contributions should be treated as implicit and positive contribution as explicit. For that purpose OpenFOAM® provides a special source term function denoted by **fvm::SuSp** in which the implicit/explicit treatment is automatically performed. The script of this function is given in Listing 14.5.

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type> >
Foam::fvm::SuSp
(
    const DimensionedField<scalar, volMesh>& susp,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = vf.mesh();

    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            dimVol*susp.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm();

    fvm.diag() += mesh.V()*max(susp.field(), scalar(0));

    fvm.source() -= mesh.V()*min(susp.field(), scalar(0))
        *vf.internalField();

    return tfvm;
}
```

Listing 14.5 Script used for the definition and implementation of the **SuSp** function

In this function both diagonal and source term vectors are filled depending on the local sign of the slope of the linearized source term. In fact the use of the **max/min** function achieves the selective discretization. For instance, if in a generic cell of the domain the slope of the linearized source term assumes a negative value (here taken as positive) the contribution to the source vector is zero (i.e., min(SuSp.field(), scalar(0)) = 0) and the opposite for the diagonal contribution.

14.5.2.2 Under-Relaxation

The under-relaxation methods used in OpenFOAM® include both the implicit technique of Patankar and the explicit variable relaxation. More specifically the

implicit under relaxation is applied only to the fvMatrix object (i.e., the actual finite volume discretization matrix) while the explicit relaxation is defined only for **GeometricField** objects.

The explicit relaxation described by Eq. (14.6) can be found in the file GeometricField.C in the "$FOAM_SRC/OpenFOAM/fields/GeometricFields/GeometricField" directory. The dedicated function (Listing 14.6) inside the GeometricField class for performing this task is given by

```
template<class Type, template<class> class PatchField, class GeoMesh>
void   Foam::GeometricField<Type,   PatchField,   GeoMesh>::relax(const
scalar alpha)
{
    if (debug)
    {
        InfoIn
        (
            "GeometricField<Type, PatchField, GeoMesh>::relax"
            "(const scalar alpha)"
        )  << "Relaxing" << endl << this->info() << " by " << alpha <<
endl;
    }
    operator==(prevIter() + alpha*(*this - prevIter()));
}
```

Listing 14.6 Script of the function in the **GeometricField** class for explicit under relaxation

where the operator "==" defines the new value of the **GeometricField** itself based on the current value and the previous one in accordance with Eq. (14.6).

In general to explicitly relax a variable, first its value is stored in the **`prevIter()`** array, after that calculations are performed to obtain the new predicted value, and finally relaxation is applied. For example, using the pressure "p" as the GeometricField variable the following should be written (Listing 14.7):

```
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
… //any preliminary operation
p.storePrevIter();
p=…   //Perform the operation for the new predicted pressure
p.relax();
```

Listing 14.7 Explicit under relaxation of the pressure field

In this case the value of the relaxation factor is directly read from the **fvSolution** dictionary. In case the developer requires a certain constant value, say 0.5, the last line in Listing 14.5 should be replaced by "**`p.relax(0.5);`**".

The Patankar relaxation is applied directly to the matrix of coefficients and in OpenFOAM® it is implemented inside the **fvMatrix** class. The file **fvMatrix.C** in the directory "$FOAM_SRC/finiteVolume/fvMatrices/fvMatrix" contains the definition of the implicit relaxation given in Listing 14.8.

```
template<class Type>
void Foam::fvMatrix<Type>::relax(const scalar alpha)
{
    if (alpha <= 0)
    {
        return;
    }
…
```

Listing 14.8 Script for the definition of implicit relaxation

The function definition is quite long, mainly due to the constraints imposed by the diagonal dominance requirement of the matrix of coefficients, but the relevant code reads (Listing 14.9)

```
    Field<Type>& S = source();
    scalarField& D = diag();

     // Store the current unrelaxed diagonal for use in updating the
source
    scalarField D0(D);
…
    // ... then relax
    D /= alpha;
    // Finally add the relaxation contribution to the source.
    S += (D - D0)*psi_.internalField();
```

Listing 14.9 Excerpts of the script used for implicit under relaxation

In the first part, after checking the sign of the slope of the linearized source, references to the diagonal and source vectors are setup. The original diagonal is stored under the "**D0**" scalar field and then divided by the relaxation factor "**alpha**". An additional contribution is added to the source vector of the matrix where "**psi_**" is the variable associated with the **fvMatrix** class. The source term looks slightly different from the right hand side of Eq. (14.4) but defines exactly the same contribution, as shown in the following equation:

$$\frac{(1-\lambda)}{\lambda} a_C \phi_C^* = \left(\frac{a_C}{\lambda} - a_C\right) \phi_C^* \tag{14.36}$$

14.6 Closure

The chapter dealt with the treatment of the source term in the general conservation equation and discussed several methods used for under-relaxing the algebraic system of equations. The chapter also discussed some of the indicators used for checking convergence. The next chapter is devote for the solution of incompressible flow problems.

14.7 Exercises

Exercise 1
Linearize the following source term where the dependent variable is ϕ:

$$Q_C^{\phi} = A + B|\phi_C|\phi_C$$

Exercise 2
Consider the following equation:

$$\frac{\partial \phi}{\partial t} - \nabla \cdot \Gamma \nabla \phi = -\beta(\phi - 1)^{1/3} - \kappa\left(\phi^4 - 1\right)$$

Derive the algebraic equation for an element C. Use a first order Euler scheme for the transient term and linearize the source terms given that β and κ are both positive.

Exercise 3
Consider the steady convection diffusion equation given by

$$\nabla \cdot (\rho \mathbf{v} \phi) - \nabla \cdot \Gamma \nabla \phi = 2 - \phi^3$$

a. Discretize the above equation using the SMART scheme with a deferred correction approach for the convection term, and linearize the source term.
b. Then apply under-relaxation to the discretized equation using
   (i) Patankar's under-relaxation method
   (ii) The E-Factor relaxation method
   (iii) The false transient method

Exercise 4

In the solution of turbulent flows (which will be the subject of Chap. 17), turbulence models are introduced. One such models is the well-known two-equation $k-\varepsilon$ turbulence model, with k and ε governed by conservation equations that have the form of the general conservation equation. The ε equation in the model has the following source term:

$$Q_C^{\varepsilon} = C_{\varepsilon 1}\frac{\varepsilon}{k}P_k - C_{\varepsilon 2}\rho\frac{\varepsilon^2}{k}$$

where ρ is the fluid density, $C_{\varepsilon 1}$ and $C_{\varepsilon 2}$ are positive constants, and P_k, k, and ε are all positive quantities. Suggest a linearization for Q_C^{ε}.

Exercise 5

In solving the momentum equation in $r\theta Z$ coordinates, the momentum equation in the θ direction has the following source term:

$$Q_C^{v_\theta} = -\rho\frac{v_r v_\theta}{r}$$

Suggest a linearization for $Q_C^{v_\theta}$.

Exercise 6

The Fithigh-Nagumo equations, presented below, model the evolution of animal coat pattern formation. The equations represent concentrations of two chemical substances influencing skin pigmentation whose reaction-diffusion interaction result in the formation of patterns that are reminiscent of the zebra or jaguars skins. The variations in the resulting patterns depend on the constants used in the model. The model equations are given by

$$\frac{\partial\phi}{\partial t} = \nabla\cdot a\nabla\phi + \phi - \phi^3 - \varphi + k$$
$$\tau\frac{\partial\varphi}{\partial t} = \nabla\cdot b\nabla\varphi + \phi - \varphi$$

Solve these equations using uFVM and OpenFOAM® for the following values of the constants:

$$a = 2.8\times 10^{-4}$$
$$b = 5\times 10^{-3}$$
$$\tau = 0.1$$
$$k = -0.005$$

Let the computational domain be a square of dimension [4, 4] discretized with a mesh of size 100×100 elements. As boundary conditions, use a zero gradient over all boundaries with a random initial conditions for the ϕ (values within [0, 1]) and

initial conditions for $\varphi = 1 - \phi$. Solve the problem over 5 s with a time step $\Delta t \leq 10^{-4}$ s, using the first order Euler Implicit Scheme and the second order Crank-Nicholson scheme and compare results. Make sure that the source term is linearized.

References

1. Patankar S (1980) Numerical heat transfer and fluid flow. McGraw Hill, New York
2. Van Doormaal JP, Raithby GD (1984) Enhancement to the SIMPLE method for predicting incompressible fluid flows. Numer Heat Transf 7:147–163
3. Mallinson GD, de Vahl Davis G (1973) The method of the false transient for the solution of coupled elliptic equations. J Comput Phys 12(4):435–461
4. Raithby GD, Schneider GE (1979) Numerical solution of problems in incompressible fluid flow: treatment of the velocity-pressure coupling. Numer Heat Transf 2:417–440
5. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org

Part III
Algorithms

Chapter 15
Fluid Flow Computation: Incompressible Flows

Abstract In previous chapters the procedure for discretizing and solving the general transport equation for the variable ϕ in the presence of a known velocity field was formulated. In general, the velocity field is not known and has to be computed by solving the set of Navier-Stokes equations. For incompressible flows this task is complicated by the strong coupling that exist between pressure and velocity and by the fact that pressure does not appear as a primary variable in either the momentum or continuity equations. The focus of this chapter is on presenting a method that addresses these two issues, and computes the flow field for incompressible fluid flows. This is accomplished initially on a one dimensional staggered grid, then on a collocated one dimensional grid and finally on a collocated three dimensional unstructured grid. In addition to fully deriving the SIMPLE, SIMPLEC, PRIME and PISO algorithms, the Rhie-Chow interpolation and its extension to transient, relaxation and body force terms are clearly formulated. Finally, the implementation details for a number of frequently encountered boundary conditions are presented.

15.1 The Main Difficulty

The general conservation equation dealt with in previous chapters can be reformed into an equation similar to the continuity and momentum equations. Yet the numerical techniques presented up till now are not enough to allow for the resolution of the Navier-Stokes equations. Solving general fluid flows requires an algorithm [1] that can deal with the pressure velocity coupling. To understand this issue, the continuity and momentum equations are reproduced below.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_15

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{15.1}$$

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = -\nabla p + \nabla \cdot \left\{ \mu \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^{\mathrm{T}} \right] \right\} + \mathbf{f}_b \tag{15.2}$$

That Eqs. (15.1) and (15.2) are nonlinear is not by itself an unsurmountable difficulty, since such a problem is usually handled by adopting an iterative approach. Moreover, Eq. (15.2) is a vector equation, which when written in terms of its components results in a system of scalar equations that can be solved sequentially. Furthermore, the stress tensor can be reformulated into a diffusion-like term and treated implicitly, with its second part (i.e., the transpose of the velocity gradient) evaluated explicitly based on previous iteration values and added to the source. The main issue that cannot be addressed directly with the numerics of the general scalar equation, is the unavailability of an explicit equation for computing the pressure field that appears in the momentum equation.

A review of Eqs. (15.1) and (15.2) reveals that while the velocity field can be computed using the momentum equation, the pressure field appearing in the momentum equation cannot be computed directly from the continuity equation. This strong yet implicit coupling can be made more evident by rewriting the set of equations in a matrix form as

$$\mathbf{A}\mathbf{u} = \begin{pmatrix} \mathbf{F} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{P} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_b \\ \mathbf{0} \end{pmatrix}. \tag{15.3}$$

In this form, Eq. (15.3) shows a zero diagonal block in the system, which is a characteristic of saddle point problems, indicating that it cannot sustain the solution of the pressure and velocity fields by any iterative mean. Consequently, an equation for pressure is required and should be derived.

One approach is to simply reformulate the system of momentum and continuity equations by decomposing matrix $\mathbf{A}$ into a lower ($\mathbf{L}$) and an upper ($\mathbf{U}$) triangular matrices as

$$\mathbf{A} = \begin{pmatrix} \mathbf{F} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{B} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{F} & 0 \\ \mathbf{B} & -\mathbf{B}\mathbf{F}^{-1}\mathbf{B}^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{F}^{-1}\mathbf{B}^{\mathrm{T}} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} = \mathbf{L}\mathbf{U} \tag{15.4}$$

where the term $-\mathbf{B}\mathbf{F}^{-1}\mathbf{B}^{\mathrm{T}}$ is the Schur complement matrix.

This is in essence the approach that needs to be followed in order to iteratively solve the Navier-Stokes equations. This technique is embodied in the classical segregated SIMPLE (Semi Implicit Method for Pressure Linked Equations) algorithm of Patankar and Spalding [1–3].

The solution procedure is based on reformulating the Navier-Stokes equations in terms of a momentum and a pressure equation, which are then discretized and solved sequentially. The pressure equation is constructed by combining the semi-discretized momentum and continuity equations (approximation of the Schur complement matrix).

The algorithm is driven by a Picard type iterative procedure during which the momentum equation is solved using the pressure field of the previous iteration. The resulting velocity field conserves momentum but not necessarily mass. This velocity field is then used to construct the pressure equation whose solution is used to correct both the pressure and velocity fields so as to enforce mass conservation. A new iteration is then started and the sequence is repeated until the velocity and pressure fields satisfy both mass and momentum conservation.

This algorithm can be described in matrix form as

$$\begin{pmatrix} \mathbf{I} & \mathbf{D}^{-1}\mathbf{B}^{\mathrm{T}} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{*} \\ \mathbf{p}^{*} \end{pmatrix} \tag{15.5}$$

followed by an update to the velocity field using

$$\begin{pmatrix} \mathbf{F} & \mathbf{0} \\ \mathbf{B} & -\mathbf{B}\mathbf{D}^{-1}\mathbf{B}^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \mathbf{v}^{*} \\ \mathbf{p}^{*} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_b \\ \mathbf{0} \end{pmatrix} \tag{15.6}$$

where in Eqs. (15.5) and (15.6) $\mathbf{F}^{-1}$ is approximated by its inverse diagonal, $\mathbf{D}^{-1}$, and the superscript $(^{*})$ refers to intermediate values at the current iteration. The steps required are summarized as follows:

- Solve: $\mathbf{F}\mathbf{v}^{*} = \mathbf{f}_b$
- Solve: $-\mathbf{B}\mathbf{D}^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{p}^{*} = -\mathbf{B}\mathbf{v}^{*}$
- Update: $\mathbf{v} = \mathbf{v}^{*} - \mathbf{D}^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{p}^{*}$
- Update: $\mathbf{p} = \mathbf{p}^{*}$

This kind of splitting is similar to that used in the SIMPLE family of algorithms, which is the subject of this chapter.

15.2 A Preliminary Derivation

The difficulties faced in developing a solution algorithm for incompressible flow problems will be highlighted by performing the discretization in a one dimensional space over the uniform grid displayed in Fig. 15.1. For simplicity, the flow is assumed to be steady. The simplified continuity and momentum equations (written in conservative form) are given by

$$\frac{\partial(\rho u)}{\partial x} = 0 \tag{15.7}$$

$$\frac{\partial(\rho u u)}{\partial x} = \frac{\partial}{\partial x}\left(\mu \frac{\partial u}{\partial x}\right) - \frac{\partial p}{\partial x} \tag{15.8}$$

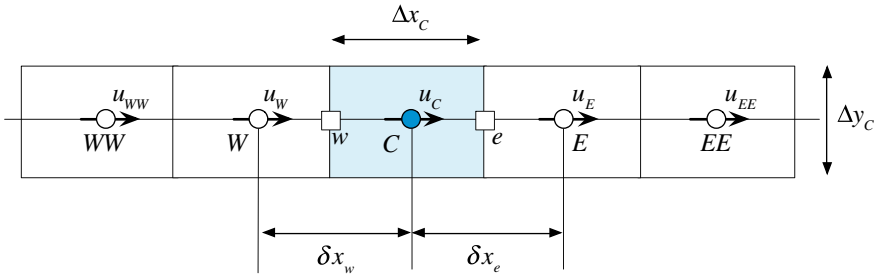


Fig. 15.1 One dimensional domain

15.2.1 Discretization of the Momentum Equation

The discretization of the momentum equation starts by integrating Eq. (15.8) over element C shown in Fig. 15.1 to yield

$$\int_{V_C} \frac{\partial(\rho uu)}{\partial x} dV = \int_{V_C} \frac{\partial}{\partial x}\left(\mu \frac{\partial u}{\partial x}\right) dV - \int_{V_C} \frac{\partial p}{\partial x} dV \tag{15.9}$$

The volume integrals of the convection and diffusion terms in Eq. (15.9) are then transformed into surface integrals by invoking the divergence theorem to give

$$\int_{\partial V_C} (\rho uu) dy = \int_{\partial V_C} \mu \frac{\partial u}{\partial x} dy - \int_{V_C} \frac{\partial p}{\partial x} dV \tag{15.10}$$

Representing the surface integrals by summation of fluxes over the faces of the element, and using a single Gaussian point for the face integrals, the semi-discretized forms of the left and right hand sides of Eq. (15.8) become

$$\underbrace{(\rho u \Delta y)_e}_{\dot{m}_e} u_e + \underbrace{-(\rho u \Delta y)_w}_{\dot{m}_w} u_w = \left(\mu \frac{\partial u}{\partial x} \Delta y\right)_e - \left(\mu \frac{\partial u}{\partial x} \Delta y\right)_w - \int_{V_C} \frac{\partial p}{\partial x} dV \tag{15.11}$$

which can be rewritten as

$$\underbrace{\dot{m}_e u_e + \dot{m}_w u_w}_{Convection} - \underbrace{\left[\left(\mu \frac{\partial u}{\partial x} \Delta y\right)_e - \left(\mu \frac{\partial u}{\partial x} \Delta y\right)_w\right]}_{Diffusion} = - \int_{V_C} \frac{\partial p}{\partial x} dV \tag{15.12}$$

The convection and diffusion terms can be discretized using any of the techniques described in previous chapters to yield an algebraic equation of the form

$$a_C^u u_C + \sum_{F\sim NB(C)} \left(a_F^u u_F\right) = b_C^u - \int_{V_C} \frac{\partial p}{\partial x} dV \tag{15.13}$$

The discretization of the pressure term is deferred till after the discretization of the continuity equation.

15.2.2 Discretization of the Continuity Equation

The discretized form of the continuity equation is obtained by integrating Eq. (15.7) over element C displayed in Fig. 15.1 to give

$$\int_{V_C} \frac{\partial(\rho u)}{\partial x} dV = 0 \tag{15.14}$$

Again making use of the divergence theorem to transform the volume integral into a surface integral and then into summation of fluxes over the faces of the element, the discrete form of the continuity equation is obtained as

$$\sum_{f\sim nb(C)} (\rho u \Delta y)_f = (\rho u \Delta y)_e - (\rho u \Delta y)_w = 0 \tag{15.15}$$

or

$$\sum_{f\sim nb(C)} \dot{m}_f = \dot{m}_e + \dot{m}_w = 0 \tag{15.16}$$

15.2.3 The Checkerboard Problem

The discretization of the pressure term may be accomplished by adopting either of the following two approaches. In the first approach, the volume integral is computed via a single Gaussian integration point resulting in

$$\int_{V_C} \frac{\partial p}{\partial x} dV = \left(\frac{\partial p}{\partial x}\right)_C V_C \tag{15.17}$$

Using a central difference scheme, the discretized form of Eq. (15.17) is obtained as

$$\int_{V_C} \frac{\partial p}{\partial x} dV = \frac{p_E - p_W}{2\Delta x} V_C \tag{15.18}$$

In the second approach, the volume integral of the pressure gradient term is transformed into a surface integral such that

$$\int_{V_C} \frac{\partial p}{\partial x} dV = \int_{\partial V_C} p \, dy \tag{15.19}$$

Rewriting the surface integral as a summation of fluxes over the faces of the element, Eq. (15.19) becomes

$$\int_{V_C} \frac{\partial p}{\partial x} dV = \int_{\partial V_C} p \, dy = p_e \Delta y_e - p_w \Delta y_w = (p_e - p_w)\Delta y = (p_e - p_w) \frac{V_C}{\Delta x} \tag{15.20}$$

Selecting a linear interpolation profile for the variation of pressure, the pressure gradient term can be rewritten as a function of pressure values at the main grid points as

$$\int_{V_C} \frac{\partial p}{\partial x} dV = \left[\frac{1}{2}(p_E + p_C) - \frac{1}{2}(p_C + p_W) \right] \frac{V_C}{\Delta x} = \frac{p_E - p_W}{2\Delta x} V_C \tag{15.21}$$

Thus either approach leads to the same expression involving the pressure difference between the alternating points E and W.

In a similar way, using a linear interpolation profile and noticing that the density is constant and $(\Delta y)_e = (\Delta y)_w = (\Delta y)_C$, the continuity equation can be expressed as

$$u_E - u_W = 0 \tag{15.22}$$

which also relates the velocity at two alternating grid points.

In Eq. (15.21) the pressure gradient term in element C depends on the values of pressure at the two alternating, not consecutive, grid points straddling the element. The same is true for the continuity equation, which enforces conservation only for alternating velocity elements. This implies that non-physical zigzag (or checkerboard) pressure and velocity fields, like the ones shown in Fig. 15.2, will be sensed as uniform fields by the numerical scheme.

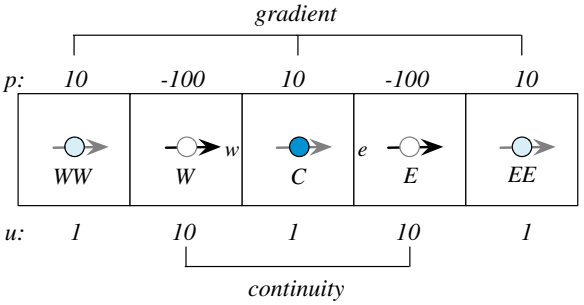


Fig. 15.2 A checkerboard pressure and velocity fields

For the pressure and velocity values shown in Fig. 15.2, the pressure gradient at points W, C, and E are found to be

$$\int_{V_W} \frac{\partial p}{\partial x} dV = (p_C - p_{WW}) \frac{V_W}{2\Delta x_W} = (10 - 10) \frac{V_W}{2\Delta x_W} = 0$$

$$\int_{V_C} \frac{\partial p}{\partial x} dV = (p_E - p_W) \frac{V_C}{2\Delta x_C} = (-100 + 100) \frac{V_C}{2\Delta x_C} = 0$$

$$\int_{V_E} \frac{\partial p}{\partial x} dV = (p_{EE} - p_C) \frac{V_E}{2\Delta x_E} = (10 - 10) \frac{V_E}{2\Delta x_E} = 0$$

and the continuity equation seems to be enforced for each element since

$$\int_{V_W} \frac{\partial u}{\partial x} dV = (u_C - u_{WW}) \frac{V_W}{2\Delta x_W} = (1 - 1) \frac{V_W}{2\Delta x_W} = 0$$

$$\int_{V_C} \frac{\partial u}{\partial x} dV = (u_E - u_W) \frac{V_C}{2\Delta x_C} = (10 - 10) \frac{V_C}{2\Delta x_C} = 0$$

$$\int_{V_E} \frac{\partial u}{\partial x} dV = (u_{EE} - u_C) \frac{V_E}{2\Delta x_E} = (1 - 1) \frac{V_E}{2\Delta x_E} = 0$$

In multi dimensional situations a similar non-physical behavior can arise even if it is harder to visualize. This sets the ground for the next step that presents one approach to resolve this problem.

15.2.4 The Staggered Grid

The culprit in the previous formulation is the uncoupling between the pressure and velocity fields. Coupling can be enforced if the different variables are stored at

staggered locations such that no interpolation is needed to calculate the pressure gradient in the momentum equation and the velocity field in the continuity equation. Such a staggered grid is shown in Fig. 15.3a, b. In the staggered grid the velocity field is stored at cell faces (Fig. 15.3a), while pressure and all other variables are stored at cell centroids (Fig. 15.3b).

With this formulation, the discretized continuity equation for element C becomes

$$\sum_{f\sim nb(C)} \dot{m}_f = \dot{m}_e + \dot{m}_w = 0 \quad \text{or} \quad u_e - u_w = 0 \tag{15.23}$$

with no need for interpolation as the velocity values are available at the e and w locations. Moreover, the momentum equation is integrated over elements similar to element e resulting in the following discretized momentum equation:

$$a_e^u u_e + \sum_{f\sim NB(e)} a_f^u u_f = b_e^u - V_e(\nabla p)_e = b_e^u - V_e \frac{p_E - p_C}{\delta x_e} \tag{15.24}$$

The pressure gradient is related to values at the consecutive grid points straddling the element face with no interpolation needed. Therefore checkerboard pressure and velocity field solutions are inadmissible as they will be easily detected and eliminated by the numerical method.

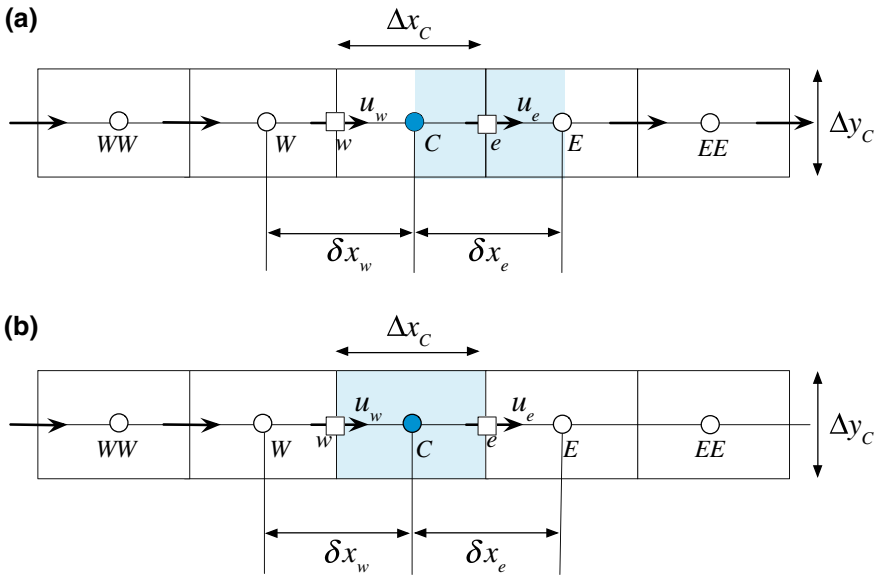


Fig. 15.3 An element for **a** the momentum equation and **b** the continuity equation in a one dimensional staggered grid arrangement

15.2.5 The Pressure Correction Equation

The derivations presented next are based on the work of Patankar and Spalding [2, 3], who developed the initial implementation of the SIMPLE (Semi Implicit Method for Pressure Linked Equations) algorithm.

Starting with the continuity and momentum equations given respectively by (Fig. 15.3)

$$\sum_{f\sim nb(C)} \dot{m}_f = 0 \tag{15.25}$$

$$a_e^u u_e + \sum_{f\sim NB(e)} a_f^u u_f = b_e^u - V_e \left(\frac{\partial p}{\partial x}\right)_e \tag{15.26}$$

the solution proceeds by providing an initial guess for the velocity and pressure fields. Denoting the initial guess or the solution at the starts of any iteration with a superscript (n), then the tentative velocity and pressure fields are given by $u^{(n)}$ and $p^{(n)}$. At any iteration, solving the momentum equation first for the velocity field, the solution obtained is denoted by a superscript * as it is not the final solution at the current iteration. Thus, the momentum equation satisfies

$$a_e^u u_e^* + \sum_{f\sim NB(e)} a_f^u u_f^* = b_e^u - V_e \left(\frac{\partial p^{(n)}}{\partial x}\right)_e \tag{15.27}$$

where the pressure field is still based on values from the previous iteration. The computed velocity field u^* satisfies the momentum equation but not necessarily the continuity equation, since the pressure field is not exact. Therefore a correction is sought to ensure that the velocity (or the mass flow rate) and pressure fields satisfy the continuity equation.

Denoting the correction fields with a superscript prime, i.e., (u', p'), then the sought after velocity and pressure are given by

$$\begin{aligned} u &= u^* + u' \\ p &= p^* + p' \end{aligned} \tag{15.28}$$

Note that the mass flow rate at cell faces will also be corrected according to

$$\begin{aligned} \dot{m}_f &= \dot{m}_f^* + \rho u' S_f^x \\ &= \dot{m}_f^* + m_f' \end{aligned} \tag{15.29}$$

such that the exact mass flow rate satisfies the continuity equation, i.e.,

$$\dot{m}_e + \dot{m}_w = \dot{m}_e^* + \dot{m}_e' + \dot{m}_w^* + \dot{m}_w' = 0 \tag{15.30}$$

which can be rewritten as

$$\dot{m}_e' + \dot{m}_w' = -\dot{m}_e^* - \dot{m}_w^* \tag{15.31}$$

This is an interesting form of the continuity equation showing that once the computed mass flow rate reaches the exact solution and satisfies the continuity equation, then the RHS becomes zero leading to a zero correction field. Thus it is the mass conservation error of the current fields that drives the correction field. The mass flow rates and mass flow rate corrections at an element faces are given by

$$\begin{aligned} \dot{m}_e &= \rho \mathbf{v}_e^* \cdot \mathbf{S}_e = \rho u_e^* S_e^x = \rho u_e^* \Delta y_e \\ \dot{m}_w &= \rho \mathbf{v}_w^* \cdot \mathbf{S}_w = \rho u_w^* S_w^x = -\rho u_w^* \Delta y_w \end{aligned} \tag{15.32}$$

and

$$\begin{aligned} \dot{m}_e' &= \rho \mathbf{v}_e' \cdot \mathbf{S}_e = \rho u_e' S_e^x = \rho u_e' \Delta y_e \\ \dot{m}_w' &= \rho \mathbf{v}_w' \cdot \mathbf{S}_w = \rho u_w' S_w^x = -\rho u_w' \Delta y_w \end{aligned} \tag{15.33}$$

where in Eqs. (15.32) and (15.33) the fact that $S_e^x = \Delta y_e$ and $S_w^x = -\Delta y_w$ has been used.

The pressure field does not appear in Eq. (15.31) and to bring it into the equation, the discrete form of the momentum equation is used. The process starts by rewriting Eq. (15.26) in a more compact form as

$$u_e + H_e(u) = B_e^u - D_e^u \left(\frac{\partial p}{\partial x} \right)_e \tag{15.34}$$

where

$$H_e(u) = \sum_{f \sim NB(e)} \frac{a_f^u}{a_e^u} u_f \quad B_e^u = \frac{b_e^u}{a_e^u} \quad \text{and} \quad D_e^u = \frac{V_e}{a_e^u} \tag{15.35}$$

For the case of the computed velocity field, the above equation is written as

$$u_e^* + H_e(u^*) = B_e^u - D_e^u \left(\frac{\partial p^{(n)}}{\partial x} \right)_e \tag{15.36}$$

Subtracting the computed momentum equation, Eq. (15.36), from the exact one, Eq. (15.34), an equation for the correction field is obtained as

$$u'_e + \underline{H_e(u')} = -D^u_e \left(\frac{\partial p'}{\partial x}\right)_e \tag{15.37}$$

A similar approach is used for the w face yielding

$$u'_w + \underline{H_w(u')} = -D^u_w \left(\frac{\partial p'}{\partial x}\right)_w \tag{15.38}$$

Substituting Eq. (15.33) into the continuity equation, Eq. (15.31), its expanded form becomes

$$\rho_e u'_e \Delta y_e + \left(-\rho_w u'_w \Delta y_w\right) = -(\dot{m}^*_e + \dot{m}^*_w). \tag{15.39}$$

Then replacing the discrete forms of u'_e and u'_w computed from Eqs. (15.37) and (15.38), respectively, in Eq. (15.39), an equation involving pressure correction is obtained and is given by

$$\begin{aligned} &\rho_e \left[-H_e(u') - D^u_e \left(\frac{\partial p'}{\partial x}\right)_e\right] \Delta y_e \\ &\quad - \rho_w \left[-H_w(u') - D^u_w \left(\frac{\partial p'}{\partial x}\right)_w\right] \Delta y_w = -\left(\dot{m}^*_e + \dot{m}^*_w\right) \end{aligned} \tag{15.40}$$

In this equation the pressure field appears in a diffusion like form, which after discretization becomes

$$\begin{aligned} &\rho_e \left[-H_e(u') - D^u_e \left(\frac{p'_E - p'_C}{\Delta x}\right)_e\right] \Delta y_e \\ &\quad + \rho_w \left[-H_w(u') - D^u_w \left(\frac{p'_C - p'_W}{\Delta x}\right)_w\right] (-\Delta y_w) = -\left(\dot{m}^*_e + \dot{m}^*_w\right) \end{aligned} \tag{15.41}$$

or

$$\begin{aligned} &- \rho_e D^u_e \left(\frac{\Delta y_w}{\Delta x_w}\right) \left(p'_E - p'_C\right) - \rho_w D^u_w \left(-\frac{\Delta y_w}{\Delta x_w}\right) \left(p'_C - p'_W\right) \\ &\quad = -\left(\dot{m}^*_e + \dot{m}^*_w\right) + \left(\rho_e H_e(u') \Delta y_e + \rho_w H_w(u')(-\Delta y_w)\right) \end{aligned} \tag{15.42}$$

Rearranging, the pressure correction equation is formulated as

$$a^{p'}_C p'_C + a^{p'}_E p'_E + a^{p'}_W p'_W = b^{p'}_C \tag{15.43}$$

where

$$\begin{aligned} a_E^{p'} &= -\frac{\rho_e D_e^u \Delta y_e}{\delta x_e} \\ a_W^{p'} &= -\frac{\rho_w D_w^u \Delta y_w}{\delta x_w} \\ a_C^{p'} &= -\left(a_E^{p'} + a_W^{p'}\right) \\ b_C^{p'} &= -\left(\dot{m}_e^* + \dot{m}_w^*\right) + \underline{\left[\rho_e \Delta y_e H_e(u') - \rho_w \Delta y_w H_w(u')\right]} \end{aligned} \tag{15.44}$$

The underlined terms in Eqs. (15.37), (15.38), and (15.44) involve corrections which become zero at the state of convergence. Therefore they have no effect on the final solution. Different approximations to these terms result in different algorithms as will be explained later. In the original SIMPLE algorithm these terms are simply neglected. Moreover for one dimensional constant area situations Δy may be set to 1 and dropped from the equations.

15.2.6 The SIMPLE Algorithm on Staggered Grid

Using the momentum and pressure correction equations, a solution to the flow problem can be obtained. In the SIMPLE algorithm this solution is found iteratively by generating pressure and velocity fields that consecutively satisfy the momentum and continuity equations, while approaching the final solution (which satisfies both equations) at every iteration [4–6]. This sequential, rather than simultaneous, solution of the equations is denoted in the literature by the segregated approach. The sequence of events in the segregated SIMPLE algorithm can be summarized as follows:

1. Start with a guessed pressure and velocity fields $p^{(n)}$ and $u^{(n)}$, respectively.
2. Solve the momentum equation given by Eq. (15.27) to obtain a new velocity field u_f^*.
3. Update the mass flow rates using the momentum satisfying velocity field to obtain the $\dot{m}_f^*$ field.
4. Using the new mass flow rates solve the pressure correction equation to obtain a pressure correction field p'.
5. Update the pressure and velocity fields to obtain continuity-satisfying fields using the following equations:

$$
\begin{aligned}
u_f^{**} &= u_f^* + u_f' \quad u_f' = -D_f^u \left(\frac{\partial p'}{\partial x}\right)_f \\
p_C^* &= p_C^{(n)} + p_C' \\
\dot{m}_f^{**} &= \dot{m}_f^* + \dot{m}_f' \quad \dot{m}_f' = -\rho_f D_f^u \Delta y_f \left(\frac{\partial p'}{\partial x}\right)_f
\end{aligned}
\tag{15.45}
$$

6. set $u^{(n)} = u^{**}$ and $p^{(n)} = p^*$
7. Go back to step 2 and repeat until convergence.

The SIMPLE algorithm is best illustrated via the example presented next.

Example 1

Flow in a Pipe Network

A portion of a water pipe system is shown in Fig. 15.4. *The momentum equation for the flow in the pipes can be written as*

$$\dot{m} = \rho u A = -D\Delta P$$

where $D_A = 0.5$, $D_B = D_F = 0.4$, $D_C = D_E = 0.3$, $D_D = 0.19$, $D_G = 0.1875$, *and* $D_H = 0.35$. *Using the SIMPLE algorithm, calculate the unknown mass flow rates and pressures in the system.*

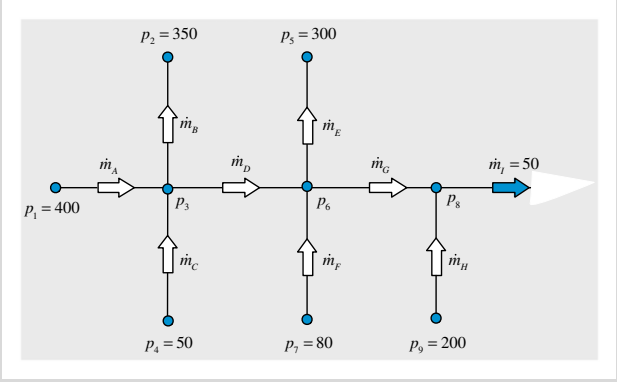


Fig. 15.4 A portion of a water pipe system

Solution

In this system, the mass flow rate field is used as a variable instead of the velocity field. This is not problematic since the momentum equation has been simplified by dropping the convection and diffusion terms as their values are negligible compared to the pressure head.

The solution using the SIMPLE algorithm starts by first computing an initial velocity field using the assumed pressure field, and then predicting a pressure field that enforces continuity on the just computed velocity field.

This procedure is summarized as

1. Start with a guessed pressure field.
2. Compute the mass flow rates using the given momentum equation.
3. Construct a pressure correction equation that enforces continuity (mass conservation) and use it to correct the pressure and velocity fields.

No iterations will be needed since there are no non-linear effects induced by a convection term.

step 1

Start by assigning guessed values to the pressure at the locations where solutions are to be found. Thus assume the following:

$p_3^{(n)} = 300 \quad p_6^{(n)} = 200 \quad p_8^{(n)} = 120$ (other values could have been used)

step 2

Based on the assumed pressure values calculate the various mass flow rates using the momentum equations according to

$$\dot{m}_A^* = D_A\,(p_1 - p_3^{(n)}) = 0.5 * (400 - 300) = 50$$
$$\dot{m}_B^* = D_B\,(p_3^{(n)} - p_2) = 0.4 * (300 - 350) = -20$$
$$\dot{m}_C^* = D_C\,(p_4 - p_3^{(n)}) = 0.3 * (50 - 300) = -75$$
$$\dot{m}_D^* = D_D\,(p_3^{(n)} - p_6^{(n)}) = 0.19 * (300 - 200) = 19$$
$$\dot{m}_E^* = D_E\,(p_6^{(n)} - p_5) = 0.3 * (200 - 300) = -30$$
$$\dot{m}_F^* = D_F\,(p_7 - p_6^{(n)}) = 0.4 * (80 - 200) = -48$$
$$\dot{m}_G^* = D_G\,(p_6^{(n)} - p_8^{(n)}) = 0.1875 * (200 - 120) = 15$$
$$\dot{m}_H^* = D_H\,(p_9 - p_8^{(n)}) = 0.35 * (200 - 120) = 28$$

step 3

Check whether the mass flow rates satisfy continuity by computing $\sum_{\sim k} \dot{m}_k^*$ at all interior points, i.e.,

$$\textit{Node } 3\text{:}\ \sum_{\sim k} \left(\dot{m}_k^*\right) = \dot{m}_B^* + \dot{m}_D^* - \dot{m}_A^* - \dot{m}_C^* = -20 + 19 - 50 + 75 = 24$$

$$\textit{Node } 6\text{:}\ \sum_{\sim k} \left(\dot{m}_k^*\right) = \dot{m}_G^* + \dot{m}_E^* - \dot{m}_D^* - \dot{m}_F^* = 15 - 30 - 19 + 48 = 14$$

$$\textit{Node } 8\text{:}\ \sum_{\sim k} \left(\dot{m}_k^*\right) = \dot{m}_I^* - \dot{m}_G^* - \dot{m}_H^* = 50 - 15 - 28 = 7$$

Since mass conservation is not satisfied, correction fields are needed and pressure correction equations are derived as follows:

$$\sum_{\sim k}\left(\dot{m}_k^* + \dot{m}_k'\right) = 0 \Rightarrow \sum_{\sim k}\left(\dot{m}_k'\right) = -\sum_{\sim k}\left(\dot{m}_k^*\right)$$

In term of pressure corrections, mass flow rate corrections can be expressed as

$$\begin{aligned}
\dot{m}_A' &= D_A\left(-p_3'\right)\\
\dot{m}_B' &= D_B\left(p_3'\right)\\
\dot{m}_C' &= D_C\left(-p_3'\right)\\
\dot{m}_D' &= D_D\left(p_3' - p_6'\right)\\
\dot{m}_E' &= D_E\left(p_6'\right)\\
\dot{m}_F' &= D_F\left(-p_6'\right)\\
\dot{m}_G' &= D_G\left(p_6' - p_8'\right)\\
\dot{m}_H' &= D_H\left(-p_8'\right)
\end{aligned}$$

Note that p_1', p_2', p_4', p_5' and p_7' are set to zero since the corresponding pressure values are known and hence represent the exact values.

The flow field at nodes 3, 6, and 8 are respectively given by

$$\begin{aligned}
\dot{m}_B' + \dot{m}_D' - \dot{m}_A' - \dot{m}_C' &= -\left(\dot{m}_B^* + \dot{m}_D^* - \dot{m}_A^* - \dot{m}_C^*\right)\\
\dot{m}_G' + \dot{m}_E' - \dot{m}_D' - \dot{m}_F' &= -\left(\dot{m}_G^* + \dot{m}_E^* - \dot{m}_D^* - \dot{m}_F^*\right)\\
-\dot{m}_G' - \dot{m}_H' &= -\left(\dot{m}_I^* - \dot{m}_G^* - \dot{m}_H^*\right)
\end{aligned}$$

and using pressure corrections as

$$\begin{aligned}
&D_B\left(p_3'\right) + D_D\left(p_3' - p_6'\right) - D_A\left(-p_3'\right) - D_C\left(-p_3'\right) = -\left(\dot{m}_B^* + \dot{m}_D^* - \dot{m}_A^* - \dot{m}_C^*\right)\\
&D_G\left(p_6' - p_8'\right) + D_E\left(p_6'\right) - D_D\left(p_3' - p_6'\right) - D_F\left(-p_6'\right) = -\left(\dot{m}_G^* + \dot{m}_E^* - \dot{m}_D^* - \dot{m}_F^*\right)\\
&-D_G\left(p_6' - p_8'\right) - D_H\left(-p_8'\right) = -\left(\dot{m}_I^* - \dot{m}_G^* - \dot{m}_H^*\right)
\end{aligned}$$

After simplification the above equations become

$$\begin{aligned}
1.39\left(p_3'\right) - 0.19\left(p_6'\right) &= -\left(\dot{m}_B^* + \dot{m}_D^* - \dot{m}_A^* - \dot{m}_C^*\right)\\
1.0775\left(p_6'\right) - 0.1875\left(p_8'\right) - 0.19\left(p_3'\right) &= -\left(\dot{m}_G^* + \dot{m}_E^* - \dot{m}_D^* - \dot{m}_F^*\right)\\
-0.1875\left(p_6'\right) + 0.5375\left(p_8'\right) &= -\left(\dot{m}_I^* - \dot{m}_G^* - \dot{m}_H^*\right)
\end{aligned}$$

Substituting the tentative mass flow rates, the various correction fields satisfy

$$\begin{aligned} &1.39\left(p_3'\right) - 0.19\left(p_6'\right) = -24 \\ &1.0775\left(p_6'\right) - 0.1875\left(p_8'\right) - 0.19\left(p_3'\right) = -14 \\ &-0.1875\left(p_6'\right) + 0.5375\left(p_8'\right) = -7 \end{aligned}$$

Solving the system of pressure correction equations yields

$$\begin{cases} p_3' = -20 \\ p_6' = -20 \\ p_8' = -20 \end{cases}$$

With the pressure correction computed, the velocity and pressure fields can now be updated to produce a mass conserving velocity field. The mass flow rates are computed as

$$\begin{aligned} \dot{m}_A^{**} &= \dot{m}_A^* + \dot{m}_A' = \dot{m}_A^* - 0.5p_3' = 50 - 0.5(-20) = 60 \\ \dot{m}_B^{**} &= \dot{m}_B^* + \dot{m}_B' = \dot{m}_B^* + 0.4p_3' = -20 + 0.4(-20) = -28 \\ \dot{m}_C^{**} &= \dot{m}_C^* + \dot{m}_C' = \dot{m}_C^* - 0.3p_3' = -75 - 0.3(-20) = -69 \\ \dot{m}_D^{**} &= \dot{m}_D^* + \dot{m}_D' = \dot{m}_D^* + 0.19\left(p_3' - p_6'\right) = 19 + 0.19(-20 + 20) = 19 \\ \dot{m}_E^{**} &= \dot{m}_E^* + \dot{m}_E' = \dot{m}_E^* + 0.3p_6' = -30 + 0.3(-20) = -36 \\ \dot{m}_F^{**} &= \dot{m}_F^* + \dot{m}_F' = \dot{m}_F^* - 0.4p_6' = -48 - 0.4(-20) = -40 \\ \dot{m}_G^{**} &= \dot{m}_G^* + \dot{m}_G' = \dot{m}_G^* + 0.1875\left(p_6' - p_8'\right) = 15 + 0.1875(-20 + 20) = 15 \\ \dot{m}_H^{**} &= \dot{m}_H^* + \dot{m}_H' = \dot{m}_H^* - 0.35p_8' = 28 - 0.35(-20) = 35 \end{aligned}$$

while the pressure is updated using

$$\begin{aligned} p_3^* &= p_3^{(n)} + p_3' = 300 - 20 = 280 \\ p_6^* &= p_6^{(n)} + p_6' = 200 - 20 = 180 \\ p_8^* &= p_8^{(n)} + p_8' = 120 - 20 = 100 \end{aligned}$$

Treat the corrected values as a new guess and repeat. Better estimate for the mass flow rates are computed using the momentum equations as

$$\dot{m}_A^* = D_A\left(p_1 - p_3^{(n)}\right) = 0.5 * (400 - 280) = 60$$
$$\dot{m}_B^* = D_B\left(p_3^{(n)} - p_2\right) = 0.4 * (280 - 350) = -28$$
$$\dot{m}_C^* = D_C\left(p_4 - p_3^{(n)}\right) = 0.3 * (50 - 280) = -69$$
$$\dot{m}_D^* = D_D\left(p_3^{(n)} - p_6^{(n)}\right) = 0.19 * (280 - 180) = 19$$
$$\dot{m}_E^* = D_E\left(p_6^{(n)} - p_5\right) = 0.3 * (180 - 300) = -36$$
$$\dot{m}_F^* = D_F\left(p_7 - p_6^{(n)}\right) = 0.4 * (80 - 180) = -40$$
$$\dot{m}_G^* = D_G\left(p_6^{(n)} - p_8^{(n)}\right) = 0.1875 * (180 - 100) = 15$$
$$\dot{m}_H^* = D_H\left(p_9 - p_8^{(n)}\right) = 0.35 * (200 - 100) = 35$$

The imbalance in the mass flow rate at nodes 3, 6, and 8 are computed as

$$\sum_{\sim k}\left(\dot{m}_k^*\right) = \dot{m}_B^* + \dot{m}_D^* - \dot{m}_A^* - \dot{m}_C^* = -28 + 19 - 60 + 69 = 0$$
$$\sum_{\sim k}\left(\dot{m}_k^*\right) = \dot{m}_G^* + \dot{m}_E^* - \dot{m}_D^* - \dot{m}_F^* = 15 - 36 - 19 + 40 = 0$$
$$\sum_{\sim k}\left(\dot{m}_k^*\right) = \dot{m}_I^* - \dot{m}_G^* - \dot{m}_H^* = 50 - 15 - 35 = 0$$

The pressure correction equations become

$$\begin{aligned} &1.39\left(p_3'\right) - 0.19\left(p_6'\right) = 0 \\ &1.0775\left(p_6'\right) - 0.1875\left(p_8'\right) - 0.19\left(p_3'\right) = 0 \\ &-0.1875\left(p_6'\right) + 0.5375\left(p_8'\right) = 0 \end{aligned}$$

The solution to the pressure correction field is found to be

$$\begin{cases} p_3' = 0 \\ p_6' = 0 \\ p_8' = 0 \end{cases}$$

Thus the solution is obtained in one iteration.

15.2.7 Pressure Correction Equation in Two Dimensional Staggered Cartesian Grids

In a two dimensional Cartesian grid, three grid systems are used. One for the u-velocity component, a second one for the v-velocity component, and a third grid system for the pressure and other variables as illustrated in Fig. 15.5.

The derivations presented above for the pressure correction equation in one dimensional domains can be easily extended into multi dimensional situations. For element C shown in Fig. 15.6, the pressure correction equation is obtained as

$$a_C^{p'} p'_C + a_E^{p'} p'_E + a_W^{p'} p'_W + a_N^{p'} p'_N + a_S^{p'} p'_S = b_C^{p'} \tag{15.46}$$

where

$$\begin{aligned} a_E^{p'} &= -\frac{\rho_e D_e^u \Delta y_C}{\delta x_e} \quad a_W^{p'} = -\frac{\rho_w D_w^u \Delta y_C}{\delta x_w} \\ a_N^{p'} &= -\frac{\rho_n D_n^v \Delta x_C}{\delta y_n} \quad a_S^{p'} = -\frac{\rho_s D_s^v \Delta x_C}{\delta y_s} \\ a_C^{p'} &= -\left(a_E^{p'} + a_W^{p'} + a_N^{p'} + a_S^{p'}\right) \\ b_C^{p'} &= -\left(\dot{m}_e^* + \dot{m}_w^* + \dot{m}_n^* + \dot{m}_s^*\right) \end{aligned} \tag{15.47}$$

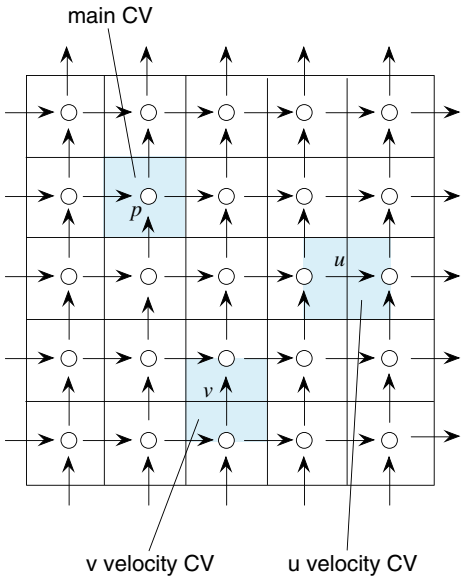


Fig. 15.5 u, v, and p elements in a two dimensional Cartesian staggered grid

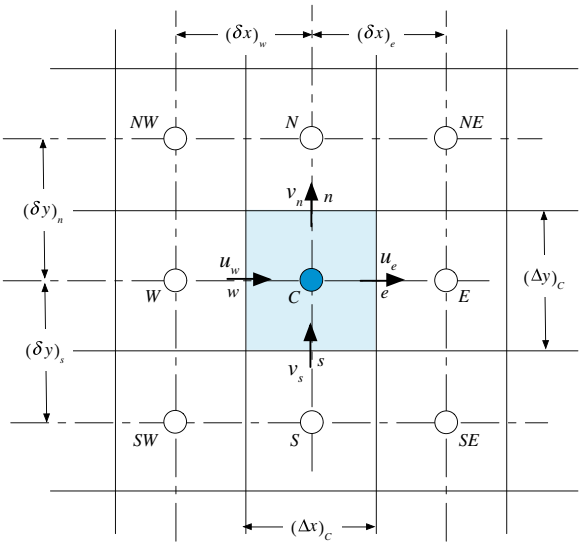


Fig. 15.6 A two dimensional Cartesian element for the derivation of the pressure correction equation

Example 2

In the two dimensional problem shown in Fig. 15.7*, the following quantities are given* $u_w = 50$, $v_s = 20$, $p_N = 0$ *and* $p_E = 10$.

The flow is steady and the density is uniform and equal to 1. *The momentum equations for* u_e *and* v_n *are given by*

$$u_e = -d_e(p_E - p_C)$$
$$v_n = -d_n(p_N - p_C)$$

where the constants $d_e = 1$ *and* $d_n = 0.25$. *The element shown has* $\Delta x = \Delta y = 1$. *Use the SIMPLE algorithm to compute the values of* u_e, v_n, *and* p_C.

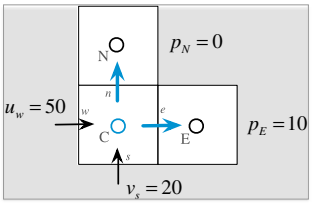


Fig. 15.7 The two dimensional domain used in Example 2

Solution

Start by assigning a guessed value to the pressure at the C location where solution is to be found. Thus assume the following:

$p_C^{(n)} = 100$ (other values could have been guessed)

Based on the assumed pressure value calculate the various velocities using the momentum equation according to

$$u_e^* = -d_e\left(p_E - p_C^{(n)}\right) = -1 * (10 - 100) = 90$$
$$v_n^* = -d_n\left(p_N - p_C^{(n)}\right) = -0.25 * (0 - 100) = 25$$

Since the density is uniform and equal to 1 and $\Delta x = \Delta y = 1$, then

$$\dot{m}_e^* = u_e^* = 90$$
$$\dot{m}_n^* = v_n^* = 25$$

Check whether the mass flow rates satisfy continuity over element C by computing $\sum_{\sim f} \dot{m}_f^*$ at the element faces.

$$\sum\nolimits_{\sim f}\left(\dot{m}_f^*\right) = \dot{m}_e^* - \dot{m}_w^* + \dot{m}_n^* - \dot{m}_s^* = 90 - 50 + 25 - 20 = 45$$

In the above mass conservation equation the negative sign for $\dot{m}_w^*$ and $\dot{m}_s^*$ is explicitly used. Since mass conservation is not satisfied, correction fields are needed and a pressure correction equation is derived as follows:

$$\sum_{\sim f}\left(\dot{m}_f^* + \dot{m}_f'\right) = 0 \Rightarrow \sum_{\sim f}\left(\dot{m}_f'\right) = -\sum_{\sim f}\left(\dot{m}_f^*\right)$$

In term of pressure corrections, mass flow rate corrections can be expressed as

$$\dot{m}_e' = u_e' = p_C' \quad \dot{m}_n' = v_n' = 0.25\, p_C'$$

The correction equation becomes

$$\dot{m}_e' + \dot{m}_n' = -\sum_{\sim f}\dot{m}_f^* \Rightarrow 1.25 p_C' = -45 \Rightarrow p_C' = -36$$

Applying the correction to the mass flow rate and pressure fields, continuity satisfying fields are obtained as

$$\dot{m}_e^{**} = \dot{m}_e^* + \dot{m}_e' = \dot{m}_e^* + p_C' = 90 - 36 = 54$$
$$\dot{m}_n^{**} = \dot{m}_n^* + \dot{m}_n' = \dot{m}_n^* + 0.25p_C' = 25 - 0.25(36) = 16$$
$$p_C^{**} = p_C^* + p_C' = 100 - 36 = 64$$

Treat the corrected values as a new guess and repeat.

$$\dot{m}_e^* = -d_e\left(p_E - p_C^*\right) = -1 * (10 - 64) = 54$$
$$\dot{m}_n^* = -d_n\left(p_N - p_C^*\right) = -0.25 * (0 - 64) = 16$$

The imbalance in the mass flow rate is computed as

$$\sum_{\sim f}\left(\dot{m}_f^*\right) = \dot{m}_e^* - \dot{m}_w^* + \dot{m}_n^* - \dot{m}_s^* = 54 - 50 + 16 - 20 = 0$$

The pressure correction equations become

$$1.25\left(p_c'\right) = 0$$

The solution to the pressure correction field is found to be

$$p_C' = 0$$

Thus the solution is obtained in one iteration as

$$u_e = 54$$
$$v_n = 16$$
$$p_C = 64$$

15.2.8 Pressure Correction Equation in Three Dimensional Staggered Cartesian Grid

Without going into details and for completeness of presentation, the pressure correction equation over the three dimensional staggered Cartesian grid shown in Fig. 15.8, where the u, v, and w velocity components are stored at the (e, w), (n, s), and (t, b) element faces, respectively, is given by

$$a_C^{p'}p_C' + a_E^{p'}p_E' + a_W^{p'}p_W' + a_N^{p'}p_N' + a_S^{p'}p_S' + a_T^{p'}p_T' + a_B^{p'}p_B' = b_C^{p'} \tag{15.48}$$

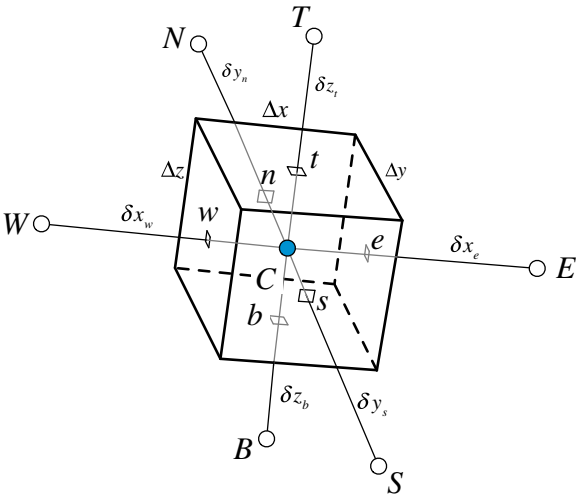


Fig. 15.8 A three dimensional Cartesian element for the derivation of the pressure correction equation

where

$$
\begin{aligned}
a_E^{p'} &= -\frac{\rho_e D_e^u \Delta y_e \Delta z_e}{\delta x_e} \quad a_W^{p'} = -\frac{\rho_w D_w^u \Delta y_w \Delta z_w}{\delta x_w} \\
a_N^{p'} &= -\frac{\rho_n D_n^v \Delta x_n \Delta z_n}{\delta y_n} \quad a_S^{p'} = -\frac{\rho_s D_s^v \Delta x_s \Delta z_s}{\delta y_s} \\
a_T^{p'} &= -\frac{\rho_t D_t^w \Delta x_t \Delta y_t}{\delta z_t} \quad a_B^{p'} = -\frac{\rho_b D_b^w \Delta x_b \Delta y_b}{\delta z_b} \\
a_C^{p'} &= -\left(a_E^{p'} + a_W^{p'} + a_N^{p'} + a_S^{p'} + a_T^{p'} + a_B^{p'}\right) \\
b_C^{p'} &= -\left(\dot{m}_e^* + \dot{m}_w^* + \dot{m}_n^* + \dot{m}_s^* + \dot{m}_t^* + \dot{m}_b^*\right)
\end{aligned}
\tag{15.49}
$$

15.3 Disadvantages of the Staggered Grid

The use of staggered grids was critical to the development of the SIMPLE algorithm. Nevertheless adopting a staggered grid arrangement has its disadvantages. As mentioned above, in two and three dimensions, three and four staggered grid systems, respectively, are required with the velocity components integrated over different elements, as shown in Fig. 15.5 for a two dimensional situation.

Besides the memory requirement to store a grid system for every velocity component and a grid system for pressure and other variables, the staggering procedure itself becomes an issue for non-Cartesian grids and more so for unstructured grids.

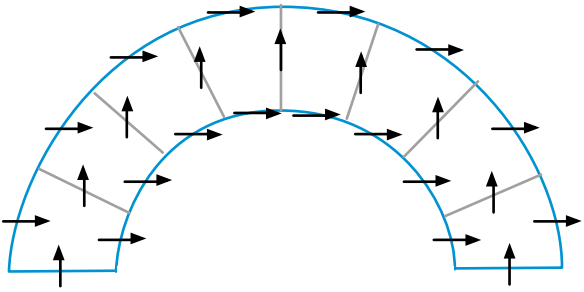


Fig. 15.9 Staggered Cartesian velocity components in a curvilinear grid system

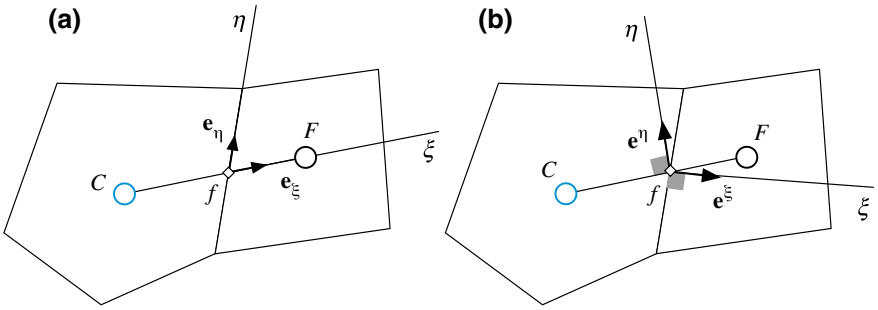


Fig. 15.10 Curvilinear coordinates **a** Covariant component and **b** contra-variant components

In curvilinear grids, the use of Cartesian velocity components can lead to problems when one or more of the surfaces become aligned with the staggered velocity component as shown in Fig. 15.9.

Therefore a better alternative in this case is to use either covariant or contra-variant curvilinear velocity components, as shown in Fig. 15.10a, b, respectively.

An example of staggering using contra-variant velocity components is shown in Fig. (15.11).

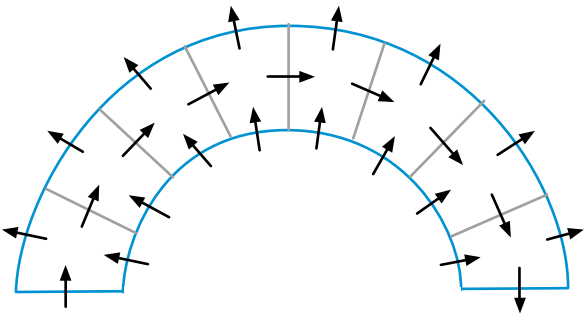


Fig. 15.11 Curvilinear velocities: staggering using contra-variant velocity components

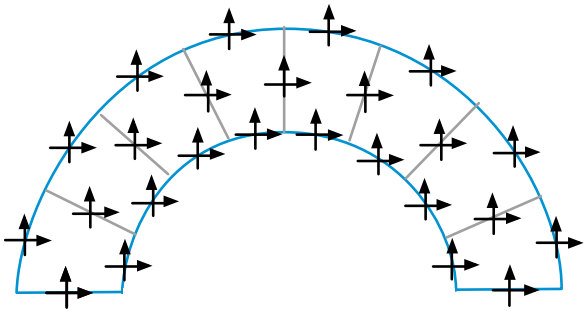


Fig. 15.12 Cartesian components defined at each face

Unfortunately complications arise when discretizing the momentum equations in curvilinear coordinates [1, 3, 4, 7], due to the increased complexity in the treatment of the diffusion term, and because the equations gain non-conservative terms.

Another option shown in Fig. 15.12 is to stagger all Cartesian velocity components in all directions so as to have all velocity components at all faces. This would double (in two dimensions) or triple (in three dimensions) the number of momentum equations to be solved. The problem is further complicated in the case of an unstructured grid. In this case there is no obvious staggering direction, and the only way for a staggering concept to apply is by changing the size of the cell elements used for the pressure and velocity components, or by resorting to staggering all velocity components along all the faces, again dramatically increasing the number of variables to be solved. Finally, the geometric information stored is more than doubled, as a new unstructured grid need to be used for the velocity components.

It turns out that the use of a cell-centered collocated grid system (Fig. 15.13), where all variables are stored at the same location (the cell centroid), is a more attractive solution. It is worth noting that while the velocity components are stored at the centroids of the elements as is the case for pressure or any other variable, the mass flux, a scalar value, in a collocated grid is stored at the element faces. The mass flux can actually be viewed as a contra-variant component, except that in this case it is computed using a custom interpolation of the discrete momentum equation, known as the Rhie-Chow interpolation, which is the subject of the next section.

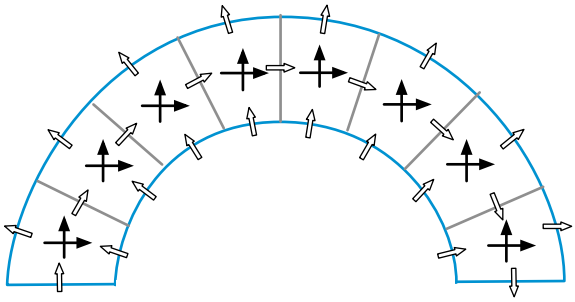


Fig. 15.13 Collocated grid arrangement

15.4 The Rhie-Chow Interpolation

The deficiency in the original collocated formulation presented earlier was in the linear interpolation used to calculate the velocities at the element faces. This interpolation resulted in decoupling the pressure and velocity values at the cell level giving rise to the checkerboard problem. In 1983, Rhie and Chow [8] reported on an interpolation procedure that allowed the formulation of the SIMPLE algorithm on a collocated grid [9–16]. In their method a dissipation term, representing the difference between two estimates of the cell face pressure gradient, is added to the linearly interpolated cell face velocity. As shown in Fig. 15.14 the two pressure gradient estimates are based on different grid stencils.

This procedure will be shown to be equivalent to constructing a pseudo-momentum equation at the element face with its coefficients linearly interpolated from the coefficients of the momentum equations at the centroids of the elements straddling the face and its pressure gradient computed using a small grid stencil. In that respect, the Rhie-Chow interpolation simply mimics the small stencil pressure-velocity coupling of the staggered grid arrangement.

Starting with the discretized x-momentum equations for cells C and F, which are

$$\begin{aligned} u_C + H_C[u] &= B_C^u - D_C^u \left(\frac{\partial p}{\partial x}\right)_C \\ u_F + H_F[u] &= B_F^u - D_F^u \left(\frac{\partial p}{\partial x}\right)_F \end{aligned} \tag{15.50}$$

A u_f velocity equation similar to that of Eq. (15.50), with the pressure gradient linked to the local neighboring pressure values, as illustrated in Fig. 15.14, will have the following form:

$$u_f + H_f[u] = B_f^u - D_f^u \left(\frac{\partial p}{\partial x}\right)_f . \tag{15.51}$$

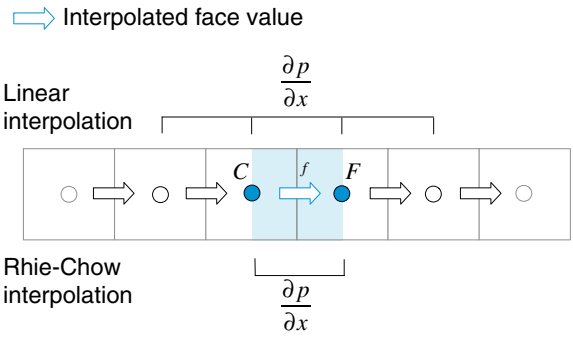


Fig. 15.14 The two pressure gradient estimates in the Rhie-Chow interpolation technique

Since in a collocated grid, the coefficients of this equation cannot be directly computed, they are approximated by interpolation from the coefficients of the neighboring nodes. Using a linear interpolation profile, these coefficients are computed as

$$\begin{aligned} H_f[u] &= \frac{1}{2}\left(H_C[u] + H_F[u]\right) = \overline{H_f}[u] \\ B_f^u &= \frac{1}{2}\left(B_C^u + B_F^u\right) = \overline{B_f^u} \\ D_f^u &= \frac{1}{2}\left(D_C^u + D_F^u\right) = \overline{D_f^u} \end{aligned} \tag{15.52}$$

Employing the values given in Eq. (15.52), the pseudo-momentum equation at the element face becomes

$$u_f + \overline{H_f}[u] = \overline{B_f^u} - \overline{D_f^u}\left(\frac{\partial p}{\partial x}\right)_f \tag{15.53}$$

This is in all practical sense the momentum equation on a "staggered" grid, which is reconstructed using the collocated grid momentum coefficients.

In all above equations and for later use, values with an over bar are obtained by linear interpolation between the values at points C and F according to

$$\overline{\square}_f = g_C \square_C + g_F \square_F \tag{15.54}$$

where g_C and g_F are geometric interpolation factors related to the position of the element face f with respect to the nodes C and F, as explained in previous chapters.

Using Eq. (15.50), $\overline{H_f}$ is rewritten as

$$\begin{aligned} \overline{H_f}[u] &= \frac{1}{2}\left(-u_C + B_C^u - D_C^u\left(\frac{\partial p}{\partial x}\right)_C - u_F + B_F^u - D_F^u\left(\frac{\partial p}{\partial x}\right)_F\right) \\ &= -\overline{u_f} - \overline{D_f^u}\overline{\left(\frac{\partial p}{\partial x}\right)_f} + \overline{B_f^u} \end{aligned} \tag{15.55}$$

where the coefficient approximation can be shown to be second order accurate, i.e.,

$$\begin{aligned} \overline{D_f^u\left(\frac{\partial p}{\partial x}\right)_f} - \overline{D_f^u}\overline{\left(\frac{\partial p}{\partial x}\right)_f} &= \frac{1}{2}\left(D_C^u\left(\frac{\partial p}{\partial x}\right)_C + D_F^u\left(\frac{\partial p}{\partial x}\right)_F\right) \\ &\quad - \frac{1}{2}\left(D_C^u + D_F^u\right) \times \frac{1}{2}\left(\left(\frac{\partial p}{\partial x}\right)_C + \left(\frac{\partial p}{\partial x}\right)_F\right) \\ &= \frac{1}{4}D_C^u\left(\left(\frac{\partial p}{\partial x}\right)_C - \left(\frac{\partial p}{\partial x}\right)_F\right) + \frac{1}{4}D_F^u\left(\left(\frac{\partial p}{\partial x}\right)_F - \left(\frac{\partial p}{\partial x}\right)_C\right) \\ &\approx O(\Delta x^2) \end{aligned} \tag{15.56}$$

Substituting Eq. (15.55) into Eq. (15.53), the velocity at the element face using the Rhie-Chow interpolation method is obtained as

$$u_f = -\overline{H_f}[u] + \overline{B_f^u} - \overline{D_f^u}\left(\frac{\partial p}{\partial x}\right)_f = \underbrace{\overline{u_f}}_{\substack{\text{average}\\ \text{velocity}}} \underbrace{- \overline{D_f^u}\left(\left(\frac{\partial p}{\partial x}\right)_f - \overline{\left(\frac{\partial p}{\partial x}\right)_f}\right)}_{\text{correction term}} \tag{15.57}$$

For a multi dimensional situation, similar interpolation formulae can be derived for the y and z velocity components and are given by

$$v_f = \overline{v_f} - \overline{D_f^v}\left(\left(\frac{\partial p}{\partial y}\right)_f - \overline{\left(\frac{\partial p}{\partial y}\right)_f}\right) \tag{15.58}$$

$$w_f = \overline{w_f} - \overline{D_f^w}\left(\left(\frac{\partial p}{\partial z}\right)_f - \overline{\left(\frac{\partial p}{\partial z}\right)_f}\right) \tag{15.59}$$

Equations (15.57)–(15.59) can be written in a vector form, more suitable for deriving the multi-dimensional pressure correction equation, as

$$\mathbf{v}_f = \overline{\mathbf{v}_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right) \tag{15.60}$$

where

$$\overline{\mathbf{D}_f^{\mathbf{v}}} = \begin{bmatrix} \overline{D_f^u} & 0 & 0 \\ 0 & \overline{D_f^v} & 0 \\ 0 & 0 & \overline{D_f^w} \end{bmatrix} \tag{15.61}$$

and where ∇p_f is computed as per Sect. 9.4 using

$$\nabla p_f = \overline{\nabla p_f} + \underbrace{\left[\frac{p_F - p_C}{d_{CF}} - \left(\overline{\nabla p_f} \cdot \mathbf{e}_{CF}\right)\right]\mathbf{e}_{CF}}_{\text{Correction to interpolated face gradient}} \tag{15.62}$$

and yielding a stencil in the **CF** direction formed only from the adjacent cell values p_F and p_C as

$$\begin{aligned} \nabla p_f \cdot \mathbf{e}_{CF} &= \overline{\nabla p_f} \cdot \mathbf{e}_{CF} + \left[\frac{p_F - p_C}{d_{CF}} - \left(\overline{\nabla p_f} \cdot \mathbf{e}_{CF}\right)\right]\mathbf{e}_{CF} \cdot \mathbf{e}_{CF} \\ &= \frac{p_F - p_C}{d_{CF}} \end{aligned} \tag{15.63}$$

With the face velocities closely linked to the pressure of adjacent cells, checkerboard fields are inadmissible rendering solutions on collocated grids viable.

15.5 General Derivation

Before proceeding with the development of the multidimensional collocated pressure correction equation, the discretized multidimensional momentum equation is first presented.

15.5.1 The Discretized Momentum Equation

The momentum equation given by Eq. (15.2) is slightly modified and written as

$$\underline{\frac{\partial}{\partial t}[\rho \mathbf{v}]} + \underline{\nabla \cdot \{\rho \mathbf{v}\mathbf{v}\}} = -\nabla p + \underline{\nabla \cdot \{\mu \nabla \mathbf{v}\}} + \nabla \cdot \{\mu (\nabla \mathbf{v})^{\mathrm{T}}\} + \mathbf{f}_b \qquad (15.64)$$

The discretized form of Eq. (15.64) in the time interval $[t - \Delta t/2, t + \Delta t/2]$ is sought over element C shown in Fig. 15.15.

In Eq. (15.64), the three underlined expressions represent, from left to right, the unsteady, convection, and diffusion term, respectively. The discretization of these terms proceeds as presented in previous chapters. The remaining terms are evaluated explicitly and treated as sources. The volume integral of the second part of the shear stress term is transformed into a surface integral using the divergence theorem and then into a summation of surface fluxes as

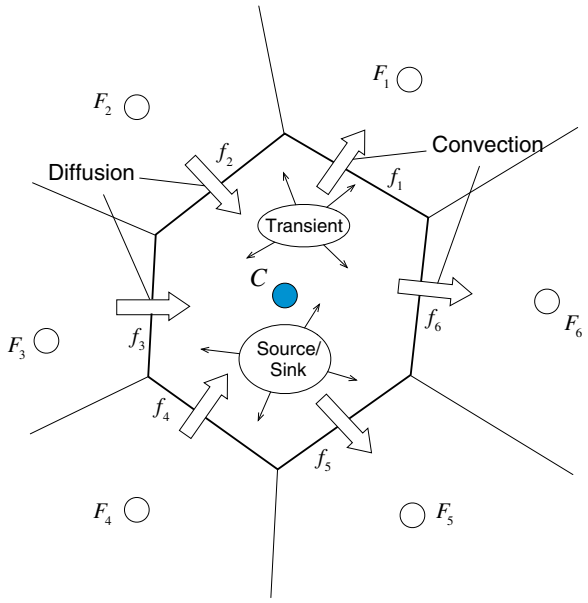


Fig. 15.15 Element C in a general unstructured grid system

$$\int_{V_C} \nabla \cdot \left\{ \mu (\nabla \mathbf{v})^{\mathrm{T}} \right\} dV = \int_{\partial V_C} \left\{ \mu (\nabla \mathbf{v})^{\mathrm{T}} \right\} \cdot d\mathbf{S} = \sum_{f \sim nb(C)} \mu (\nabla \mathbf{v})_f^{\mathrm{T}} \cdot \mathbf{S}_f \tag{15.65}$$

where the expanded form of $(\nabla \mathbf{v})_f^{\mathrm{T}} \cdot \mathbf{S}_f$ in a three dimensional coordinate system is given by

$$(\nabla \mathbf{v})_f^{\mathrm{T}} \cdot \mathbf{S}_f = \begin{bmatrix} \dfrac{\partial u}{\partial x} S_f^x + \dfrac{\partial u}{\partial y} S_f^y + \dfrac{\partial u}{\partial z} S_f^z \\ \dfrac{\partial v}{\partial x} S_f^x + \dfrac{\partial v}{\partial y} S_f^y + \dfrac{\partial v}{\partial z} S_f^z \\ \dfrac{\partial w}{\partial x} S_f^x + \dfrac{\partial w}{\partial y} S_f^y + \dfrac{\partial w}{\partial z} S_f^z \end{bmatrix} \tag{15.66}$$

The volume integral of the pressure gradient is also treated as a source term and evaluated explicitly as

$$\int_{V_C} \nabla p \, dV = (\nabla p)_C V_C \tag{15.67}$$

or transformed into a surface integral according to Eq. (2.85) and computed as

$$\int_{V_C} \nabla p \, dV = \int_{\partial V_C} p \, d\mathbf{S} = \sum_{f \sim nb(C)} p_f \mathbf{S}_f \tag{15.68}$$

The body force term is integrated directly over the control volume to yield

$$\int_{V_C} \mathbf{f}_b dV = (\mathbf{f}_b)_C V_C \tag{15.69}$$

Using a first order Euler scheme for the discretization of the unsteady term, a HR scheme for the convection term implemented via the deferred correction approach, and decomposing the diffusion flux into an implicit part aligned with the grid and an explicit cross diffusion part, the discretized momentum equation is written in vector form as

$$a_C^{\mathbf{v}} \mathbf{v}_C + \sum_{F \sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}_F = \mathbf{b}_C^{\mathbf{v}} \tag{15.70}$$

where the coefficients are given by

$$\begin{aligned} a_C^{\mathbf{v}} &= FluxC_C + \sum_{f\sim nb(C)} (FluxC_f) \\ a_F^{\mathbf{v}} &= FluxF_f \\ \mathbf{b}_C^{\mathbf{v}} &= -FluxV_C - \sum_{f\sim nb(C)} FluxV_f + \sum_{f\sim nb(C)} \mu_f (\nabla \mathbf{v})_f^{\mathrm{T}} \cdot \mathbf{S}_f - (\nabla p)_C V_C \end{aligned} \tag{15.71}$$

with the face fluxes calculated using

$$\begin{aligned} FluxC_f &= \underbrace{\left\| \dot{m}_f, 0 \right\|}_{\substack{convection \\ contribution}} + \underbrace{\mu_f \frac{E_f}{d_{CF}}}_{\substack{diffusion \\ contribution}} \\ FluxF_f &= \underbrace{-\left\| -\dot{m}_f, 0 \right\|}_{\substack{convection \\ contribution}} - \underbrace{\mu_f \frac{E_f}{d_{CF}}}_{\substack{diffusion \\ contribution}} \\ FluxV_f &= -\mu_f (\nabla \mathbf{v})_f \cdot \mathbf{T}_f + \dot{m}_f (\mathbf{v}_f^{HR} - \mathbf{v}_f^U) \end{aligned} \tag{15.72}$$

and the element fluxes computed from

$$\begin{aligned} FluxC_C &= \frac{\rho_C V_C}{\Delta t} \\ FluxV_C &= \underbrace{-\frac{\rho_C^\circ V_C}{\Delta t} \mathbf{v}_C^\circ}_{\substack{transient \\ contribution}} - \underbrace{(\mathbf{f}_b)_C V_C}_{\substack{source\ term \\ contribution}} \end{aligned} \tag{15.73}$$

Even though the algebraic form of the momentum equation, Eq. (15.70), is linear, its coefficients depend on the velocity and pressure fields. This nonlinearity is handled by an iterative process during which the coefficients are calculated at the start of every iteration based on values of the dependent variables obtained in the previous iteration. This change in the values of the coefficients results in large changes in $\mathbf{v}$ and affects the rate of convergence to the degree of even causing divergence. To slow down the changes, under-relaxation can be applied when the transient time steps used are large. Denoting the under relaxation factor by $\lambda^{\mathbf{v}}$ and adopting Patankar's implicit relaxation approach, the under relaxed momentum equation can be written as

$$\frac{a_C^{\mathbf{v}}}{\lambda^{\mathbf{v}}} \mathbf{v}_C + \sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}_F = \mathbf{b}_C^{\mathbf{v}} + \frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}} a_C^{\mathbf{v}} \mathbf{v}_C^{(n)} \tag{15.74}$$

By redefining $a_C^{\mathbf{v}}$ and $\mathbf{b}_C^{\mathbf{v}}$ such that

$$\begin{aligned} a_C^{\mathbf{v}} &\leftarrow \frac{a_C^{\mathbf{v}}}{\lambda^{\mathbf{v}}} \\ \mathbf{b}_C^{\mathbf{v}} &\leftarrow \mathbf{b}_C^{\mathbf{v}} + \frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}} a_C^{\mathbf{v}} \mathbf{v}_C^{(n)} \end{aligned} \tag{15.75}$$

the under-relaxed momentum equation can be rewritten as

$$a_C^{\mathbf{v}} \mathbf{v}_C + \sum_{F \sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}_F = \mathbf{b}_C^{\mathbf{v}} \tag{15.76}$$

For the derivation of the collocated pressure correction equation, the pressure gradient is taken out of the $\mathbf{b}_C^{\mathbf{v}}$ source term and displayed explicitly to yield

$$\mathbf{b}_C^{\mathbf{v}} = -V_C (\nabla p)_C + \hat{\mathbf{b}}_C^{\mathbf{v}} \tag{15.77}$$

Substituting back in Eq. (15.76), the momentum equation becomes

$$\mathbf{v}_C + \sum_{F \sim NB(C)} \frac{a_F^{\mathbf{v}}}{a_C^{\mathbf{v}}} \mathbf{v}_F = -\frac{V_C}{a_C^{\mathbf{v}}} (\nabla p)_C + \frac{\hat{\mathbf{b}}_C^{\mathbf{v}}}{a_C^{\mathbf{v}}}. \tag{15.78}$$

Defining the following vector operators:

$$\begin{aligned} \mathbf{H}_C[\mathbf{v}] &= \sum_{f \sim NB(C)} \frac{a_F^{\mathbf{v}}}{a_C^{\mathbf{v}}} \mathbf{v}_F \\ \mathbf{B}_C^{\mathbf{v}} &= \frac{\hat{\mathbf{b}}_C^{\mathbf{v}}}{a_C^{\mathbf{v}}} \\ \mathbf{D}_C^{\mathbf{v}} &= \frac{V_C}{a_C^{\mathbf{v}}} \end{aligned} \tag{15.79}$$

Equation (15.78) is reformulated as

$$\mathbf{v}_C + \mathbf{H}_C[\mathbf{v}] = -\mathbf{D}_C^{\mathbf{v}} (\nabla p)_C + \mathbf{B}_C^{\mathbf{v}}, \tag{15.80}$$

a form that will be useful in later derivations.

15.5.2 The Collocated Pressure Correction Equation

As in the case of a staggered grid, starting with guessed values or values obtained from the previous iteration $\left(\mathbf{v}^{(n)}, \dot{m}^{(n)}, p^{(n)}\right)$, the momentum equation, Eq. (15.80), is first solved to obtain a momentum conserving velocity field $\mathbf{v}^*$. Thus the obtained solution satisfies

$$\mathbf{v}_C^* + \mathbf{H}_C[\mathbf{v}^*] = -\mathbf{D}_C^{\mathbf{v}}\left(\nabla p^{(n)}\right)_C + \mathbf{B}_C^{\mathbf{v}} \tag{15.81}$$

while the final solution should satisfy Eq. (15.80). The difference between these two equations is that the velocity field in Eq. (15.80) satisfies both the momentum and continuity equations while the one in Eq. (15.79) does not necessarily satisfy the continuity equation because of the linearization in which pressure and velocity are based on the previous iteration values. Therefore corrections to the velocity, mass flow rate, and pressure fields are needed to enforce mass conservation. Denoting these corrections by $(\mathbf{v}', p', \dot{m}')$ the relations between the exact and computed fields can be written as

$$\begin{aligned} \mathbf{v} &= \mathbf{v}^* + \mathbf{v}' \\ p &= p^{(n)} + p' \\ m &= m^* + m' \end{aligned} \tag{15.82}$$

Substituting the mass flow rate given by Eq. (15.82) into Eq. (15.25), the continuity equation becomes

$$\sum_{f\sim nb(C)} \dot{m}_f' = -\sum_{f\sim nb(C)} \dot{m}_f^* \quad \text{where } \dot{m}_f^* = \rho_f \mathbf{v}_f^* \cdot \mathbf{S}_f \tag{15.83}$$

with the face velocity computed using the Rhie-Chow interpolation as

$$\mathbf{v}_f^* = \overline{\mathbf{v}_f^*} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f^{(n)} - \overline{\nabla p_f^{(n)}}\right). \tag{15.84}$$

When the computed mass flow rate field is conservative, the RHS of Eq. (15.83) is zero yielding a zero correction field. On the other hand, an incorrect velocity field leads to an imbalance in mass and a nonzero value of the RHS of Eq. (15.83) implying the need for a correction field for conservation to be enforced.

Mass flow rate corrections can be written in terms of velocity corrections, which can be derived by subtracting Eq. (15.81) from Eq. (15.80) to yield

$$\mathbf{v}'_C + \mathbf{H}_C[\mathbf{v}'] = -\mathbf{D}_C^{\mathbf{v}}(\nabla p')_C \tag{15.85}$$

A similar equation holds for element F and is given by

$$\mathbf{v}'_F + \mathbf{H}_F[\mathbf{v}'] = -\mathbf{D}_F^{\mathbf{v}}(\nabla p')_F \tag{15.86}$$

The mass flow rate correction at a cell face can be expressed as

$$\dot{m}'_f = \rho_f \mathbf{v}'_f \cdot \mathbf{S}_f \tag{15.87}$$

where the face velocity correction is obtained by subtracting Eq. (15.84) from Eq. (15.60) to give

$$\mathbf{v}'_f = \overline{\mathbf{v}'_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}(\nabla p'_f - \overline{\nabla p'_f}) \tag{15.88}$$

Substitution of Eqs. (15.87) and (15.88) in Eq. (15.83), leads to the following form of the pressure correction equation:

$$\underline{\sum_{f\sim nb(C)} (\rho_f \overline{\mathbf{v}'_f} \cdot \mathbf{S}_f) + \sum_{f\sim nb(C)} \left(\rho_f \overline{\mathbf{D}_f^{\mathbf{v}} \nabla p'_f} \cdot \mathbf{S}_f\right)} - \sum_{f\sim nb(C)} \left(\rho_f \overline{\mathbf{D}_f^{\mathbf{v}}} (\nabla p')_f \cdot \mathbf{S}_f\right) = -\sum_{f\sim nb(C)} \dot{m}_f^* \tag{15.89}$$

In this equation the underlined part represents the effects of the neighboring velocity corrections on the velocity correction of the element under consideration. This influence becomes clearer by interpolating Eqs. (15.85) and (15.86) to the face yielding the following equivalent expression for the underline terms:

$$\overline{\mathbf{v}'_f} + \overline{\mathbf{H}_f}[\mathbf{v}'] = -\overline{\mathbf{D}_f^{\mathbf{v}}}(\overline{\nabla p'})_f \Rightarrow \overline{\mathbf{v}'_f} + \overline{\mathbf{D}_f^{\mathbf{v}}}(\overline{\nabla p'})_f = -\overline{\mathbf{H}_f}[\mathbf{v}']. \tag{15.90}$$

Substituting Eq. (15.90) in Eq. (15.89), the pressure correction equation is rewritten as

$$\sum_{f\sim nb(C)} \left(-\rho_f \overline{\mathbf{D}_f^{\mathbf{v}}} (\nabla p')_f \cdot \mathbf{S}_f\right) = -\sum_{f\sim nb(C)} \dot{m}_f^* + \underline{\sum_{f\sim nb(C)} (\rho_f \overline{\mathbf{H}_f}[\mathbf{v}'] \cdot \mathbf{S}_f)} \tag{15.91}$$

or more explicitly in the form

$$\sum_{f\sim nb(C)} \left(-\rho_f \overline{\mathbf{D}_f^{\mathbf{v}}} (\nabla p')_f \cdot \mathbf{S}_f\right) = -\sum_{f\sim nb(C)} \dot{m}_f^* + \underline{\sum_{f\sim nb(C)} \left(\rho_f \left(\overline{\sum_{F\sim NB(C)} \frac{a_F^{\mathbf{v}}}{a_C^{\mathbf{v}}} \mathbf{v}'_F}\right) \cdot \mathbf{S}_f\right)}. \tag{15.92}$$

In Eq. (15.91) or (15.92) the treatment of the underlined term is critical to rendering the equation solvable. In the original SIMPLE algorithm it is neglected,

thus linking the velocity correction at a point directly to pressure corrections. Because this is a correction equation the modification or dropping of the term will not affect the final solution, since at convergence the corrections become zero. However it will affect the convergence rate in that the larger is the neglected term the higher will be the error present in the approximation at each iteration.

The remaining terms in Eq. (15.91) or (15.92) can be easily treated. The coefficients of the pressure correction equation are obtained as per the discretization of the diffusion term in Chap. 8, specifically the treatment of anisotropic diffusion.

Thus the term on the LHS is modified into a gradient dot product of the form

$$\begin{aligned}\left(\overline{\mathbf{D}_f^{\mathbf{v}}}(\nabla p')_f\right)\cdot\mathbf{S}_f &= \left((\nabla p')_f\overline{\mathbf{D}_f^{\mathbf{v}}}^T\right)\cdot\mathbf{S}_f \\ &= (\nabla p')_f\cdot\left(\overline{\mathbf{D}_f^{\mathbf{v}}}^T\cdot\mathbf{S}_f\right) \\ &= (\nabla p')_f\cdot\mathbf{S}'_f\end{aligned} \tag{15.93}$$

The expanded expression of $\mathbf{S}'_f$ is given by

$$\mathbf{S}'_f = \overline{\mathbf{D}_f^{\mathbf{v}}}^T\cdot\mathbf{S}_f = \begin{bmatrix}\overline{D_f^u} & 0 & 0\\ 0 & \overline{D_f^v} & 0\\ 0 & 0 & \overline{D_f^w}\end{bmatrix}\begin{bmatrix}S_f^x\\ S_f^y\\ S_f^z\end{bmatrix} = \begin{bmatrix}\overline{D_f^u}S_f^x\\ \overline{D_f^v}S_f^y\\ \overline{D_f^w}S_f^z\end{bmatrix} \tag{15.94}$$

Working with $\mathbf{S}'_f$, the discretization of the pressure correction gradient term proceeds as usual resulting in

$$\begin{aligned}(\nabla p')_f\cdot\mathbf{S}'_f &= (\nabla p')_f\cdot\mathbf{E}_f + (\nabla p')_f\cdot\mathbf{T}_f \\ &= \frac{E_f}{d_{CF}}\left(p'_F - p'_C\right) + \underline{(\nabla p')_f\cdot\mathbf{T}_f}\end{aligned} \tag{15.95}$$

where the following decomposition of $\mathbf{S}'_f$ was used:

$$\mathbf{S}'_f = \mathbf{E}_f + \mathbf{T}_f. \tag{15.96}$$

The type of decomposition could be any of those reviewed in Chap. 8, as will be detailed later. The underlined term, arising due to grid non-orthogonality, can either be neglected or retained. If neglected, it will not affect the final solution as it is a correction term. If retained, then it will be treated explicitly with an internal loop (non-orthogonal loop in OpenFOAM®). As the solution starts with a zero pressure correction field at every iteration, the term has to be updated iteratively while solving the equation.

Dropping the non-orthogonal contribution, the linearized term of the pressure correction equation becomes

$$
\begin{aligned}
(\nabla p')_f \cdot \mathbf{S}'_f &= \frac{E_f}{d_{CF}}\left(p'_F - p'_C\right) \\
&= \mathcal{D}_f\left(p'_F - p'_C\right)
\end{aligned}
\tag{15.97}
$$

Substituting back in Eq. (15.91) the algebraic form of the pressure correction equation is obtained as

$$
a_C^{p'} p'_C + \sum_{F \sim NB(C)} a_F^{p'} p'_F = b_C^{p'} \tag{15.98}
$$

with the coefficients given by

$$
\begin{aligned}
a_F^{p'} &= FluxF_f = -\rho_f \mathcal{D}_f \\
a_C^{p'} &= -\sum_{f \sim nb(C)} FluxF_f = -\sum_{F \sim NB(C)} a_F^{p'} \\
b_C^{p'} &= -\sum_{f \sim nb(C)} FluxV_f + \underline{\sum_{f \sim nb(C)} \left(\rho_f \overline{\mathbf{H}_f}[\mathbf{v}'] \cdot \mathbf{S}_f\right)} \\
&= -\sum_{f \sim nb(C)} \dot{m}_f^* + \underline{\sum_{f \sim nb(C)} \left(\rho_f \overline{\mathbf{H}_f}[\mathbf{v}'] \cdot \mathbf{S}_f\right)}
\end{aligned}
\tag{15.99}
$$

Note that different approximations to the underlined terms in Eq. (15.99) result in different algorithms. In the original SIMPLE algorithm these terms are simply neglected.

Finally the mass flow rate $\dot{m}_f^*$ in Eq. (15.99), is the one computed after solving the momentum equation using as usual the Rhie-Chow interpolation technique with the latest velocity field, i.e.,

$$
\dot{m}_f^* = \rho \mathbf{v}_f^* \cdot \mathbf{S}_f = \rho \overline{\mathbf{v}_f^*} \cdot \mathbf{S}_f - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f^{(n)} - \overline{\nabla p_f^{(n)}}\right) \cdot \mathbf{S}_f. \tag{15.100}
$$

Following the calculation of the pressure correction field, the pressure and velocity at the element centroids and the mass flow rate at the element faces are all corrected. As mentioned above, the underlined term in Eq. (15.99) is neglected in the SIMPLE algorithm resulting in large pressure correction values that may slow the rate of convergence or cause divergence. To increase robustness and improve the convergence behavior, pressure correction values obtained from Eq. (15.98) are explicitly under relaxed. No under relaxation is used when updating the velocity and mass flow rate fields since the pressure correction will ensure mass conservation for these fields. Denoting the under relaxation factor by λ^p, the following correction equations are used:

$$\begin{aligned}
\mathbf{v}_C^{**} &= \mathbf{v}_C^{*} + \mathbf{v}'_C \quad \mathbf{v}'_C = -\mathbf{D}_C^{\mathbf{v}}(\nabla p')_C \\
\dot{m}_f^{**} &= \dot{m}_f^{*} + \dot{m}'_f \quad \dot{m}'_f = -\rho_f \overline{\mathbf{D}_f^{\mathbf{v}}} \nabla p'_f \cdot \mathbf{S}_f \\
p_C^{*} &= p_C^{(n)} + \lambda^p p'_C
\end{aligned} \tag{15.101}$$

15.5.3 Calculation of the $\mathcal{D}_f$ Term

The type of decomposition suggested in Eq. (15.96) could be any of those reviewed in Chap. 8 with different approaches leading to different expressions for $\mathcal{D}_f$ as derived below.

15.5.3.1 Minimum Correction Approach

For this approach $\mathbf{E}_f$ is obtained by substituting $\mathbf{S}'_f$ for $\mathbf{S}_f$ in Eq. (8.68) leading to

$$\mathbf{E}_f = \left(\mathbf{e}_{CF} \cdot \mathbf{S}'_f\right)\mathbf{e}_{CF} \tag{15.102}$$

where $\mathbf{e}_{CF}$ is a unit vector in the CF direction. Combining Eqs. (15.94), (15.102), and (8.64), $\mathbf{E}_f$ becomes

$$\mathbf{E}_f = \frac{d_{CF}^x \overline{D_f^u} S_f^x + d_{CF}^y \overline{D_f^v} S_f^y + d_{CF}^z \overline{D_f^w} S_f^z}{d_{CF}^2} \mathbf{d}_{CF} \tag{15.103}$$

Using Eq. (15.103), the following expression for $\mathcal{D}_f$ is derived:

$$\mathcal{D}_f = \frac{E_f}{d_{CF}} = \frac{d_{CF}^x \overline{D_f^u} S_f^x + d_{CF}^y \overline{D_f^v} S_f^y + d_{CF}^z \overline{D_f^w} S_f^z}{\left(d_{CF}^x\right)^2 + \left(d_{CF}^y\right)^2 + \left(d_{CF}^z\right)^2} \tag{15.104}$$

15.5.3.2 Orthogonal Correction Approach

The definition of $\mathbf{E}_f$ in this case is obtained from Eq. (8.69) and written as

$$\mathbf{E}_f = S'_f \mathbf{e}_{CF} \tag{15.105}$$

Combining Eqs. (15.105), (15.94), and (8.64), $\mathcal{D}_f$ is found to be

$$\mathcal{D}_f = \frac{E_f}{d_{CF}} = \sqrt{\frac{\left(\overline{D_f^u} S_f^x\right)^2 + \left(\overline{D_f^v} S_f^y\right)^2 + \left(\overline{D_f^w} S_f^z\right)^2}{\left(d_{CF}^x\right)^2 + \left(d_{CF}^y\right)^2 + \left(d_{CF}^z\right)^2}} \tag{15.106}$$

15.5.3.3 Over-Relaxed Approach

In this method, $\mathbf{E}_f$ is computed from Eqs. (8.64) and (8.70) as

$$\mathbf{E}_f = \frac{\mathbf{S}'_f \cdot \mathbf{S}'_f}{\mathbf{d}_{CF} \cdot \mathbf{S}'_f} \mathbf{d}_{CF} \tag{15.107}$$

Combining Eq. (15.107) with Eq. (15.94), $\mathcal{D}_f$ is found to be

$$\mathcal{D}_f = \frac{\left(\overline{D_f^u} S_f^x\right)^2 + \left(\overline{D_f^v} S_f^y\right)^2 + \left(\overline{D_f^w} S_f^z\right)^2}{d_{CF}^x \overline{D_f^u} S_f^x + d_{CF}^y \overline{D_f^v} S_f^y + d_{CF}^z \overline{D_f^w} S_f^z} \tag{15.108}$$

Any of the above approaches can be adopted to calculate the value of $\mathcal{D}_f$.

15.5.4 The Collocated SIMPLE Algorithm

The sequence of events in the collocated SIMPLE algorithm is displayed in Fig. 15.16 and can be summarized as follows:

1. To compute the solution at iteration $n + 1$, start with the solution at time t for pressure, velocity, and mass flow rate fields, i.e., $p^{(n)}$, $u^{(n)}$, and $\dot{m}^{(n)}$, respectively, as the initial guess.
2. Solve the momentum equation given by Eq. (15.70) to obtain a new velocity field $\mathbf{v}^*$.
3. Update the mass flow rate at the element faces using the Rhie-Chow interpolation (Eq. 15.100) to compute a momentum satisfying mass flow rate field $\dot{m}^*$.
4. Using the new mass flow rates assemble the pressure correction equation (Eq. 15.98) and solve it to obtain a pressure correction field p'.
5. With the pressure correction field update the pressure and velocity fields at the element centroids and the mass flow rate at the element faces to obtain continuity-satisfying fields using Eq. (15.101). These resulting fields are denoted by u^{**}, $\dot{m}^{**}$, and p^*, respectively.
6. Treat the obtained solution as a new initial guess, return to step 2 and repeat until convergence.
7. Set the solution at time $n + 1$ (i.e., $t + \Delta t$) to be equal to the converged solution and set the current time $n + 1$ (i.e., $t + \Delta t$) to be n (i.e., t).
8. Advance to the next time step.
9. Return to step 1 and repeat until the last time step is reached.

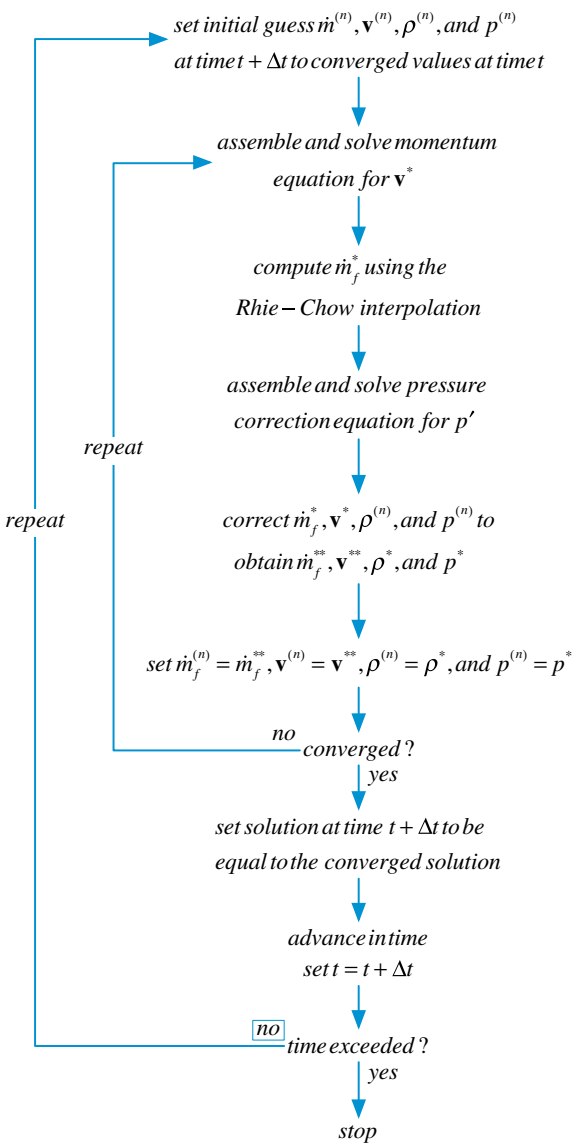


Fig. 15.16 A flow chart of the SIMPLE algorithm

Example 3

In the two dimensional problem shown below, the following quantities are given $p_W = 100$, $p_N = 20$ *and* $p_E = 50$ *and an inlet condition at face s with* $v_s = 20$ *and zero pressure gradient.*

The flow is steady state and the density is uniform of value 1. The momentum equations for u_e *and* v_n *are*

$$u_C = -d_x(p_e - p_w)$$

and

$$v_C = -d_y(p_n - p_s)$$

where the constants $d_x = 1$ *and* $d_y = 0.25$. *The element shown has* $\Delta x = \Delta y = 1$. *Use the collocated SIMPLE algorithm to derive the pressure correction equation and solve it to find the pressure for element C. Take* $p_C^{(n)} = 70$ *as an initial guess for pressure at C* (Fig. 15.17).

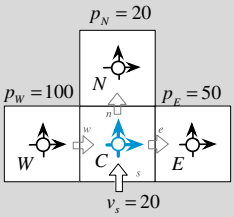


Fig. 15.17 The two dimensional domain used in Example 3

Solution

At the inlet the zero gradient condition can be used to compute

$$p_s = p_C = p_C^{(n)} = 70$$

now compute $u_C^{(n)}, v_C^{(n)}$ using

$$\begin{aligned} u_C^* &= -d_x\left(p_e^{(n)} - p_w^{(n)}\right) \\ &= -1(60 - 85) \\ &= 25 \end{aligned}$$

and

$$\begin{aligned} v_C^* &= -d_y\left(p_n^{(n)} - p_s^{(n)}\right) \\ &= -0.25(45 - 70) \\ &= 6.25 \end{aligned}$$

since $p_e^{(n)} = 0.5\left(p_E^{(n)} + p_C^{(n)}\right)$, $p_n^{(n)} = 0.5\left(p_E^{(n)} + p_N^{(n)}\right)$ and $p_w^{(n)} = 0.5\left(p_E^{(n)} + p_W^{(n)}\right)$

Now the pressure correction equation is constructed by substituting the mass flux and its correction into the mass conservation equation. Using the Rhie-Chow interpolation the face fluxes are computed as

$$\dot{m}_e^* = u_e^* \Delta y = \overline{u_e^*} - d_x \left[\underbrace{\left(p_E^{(n)} - p_C^{(n)}\right)}_{\substack{\text{pressure difference} \\ \text{across face}}} - \underbrace{0.5\left(\left(p_e^{(n)} - p_w^{(n)}\right) + \left(p_{ee}^{(n)} - p_e^{(n)}\right)\right)}_{\substack{\text{average pressure difference} \\ \text{across face}}} \right]$$

$$= \underbrace{0.5\left[d_x\left(p_e^{(n)} - p_w^{(n)}\right) + d_x\left(p_{ee}^{(n)} - p_{ww}^{(n)}\right)\right]}_{\overline{u_e^*}} - d_x \left[\underbrace{\left(p_E^{(n)} - p_C^{(n)}\right)}_{\substack{\text{pressure difference} \\ \text{across face}}} - \underbrace{0.5\left(\left(p_e^{(n)} - p_w^{(n)}\right) + \left(p_{ee}^{(n)} - p_e^{(n)}\right)\right)}_{\substack{\text{average pressure difference} \\ \text{across face}}} \right]$$

$$= -d_x\left(p_E^{(n)} - p_C^{(n)}\right)$$

$$= -1(50 - 70)$$

$$= 20$$

similarly for the n and w face

$$\dot{m}_n^* = v_n^* \Delta x = \overline{v_n^*} - d_y \left[\underbrace{\left(p_N^{(n)} - p_C^{(n)}\right)}_{\substack{\text{pressure difference} \\ \text{across face}}} - \underbrace{0.5\left(\left(p_n^{(n)} - p_s^{(n)}\right) + \left(p_{nn}^{(n)} - p_n^{(n)}\right)\right)}_{\substack{\text{average pressure difference} \\ \text{across face}}} \right]$$

$$= -d_y\left(p_N^{(n)} - p_C^{(n)}\right)$$

$$= -0.25(20 - 70)$$

$$= 12.5$$

$$\dot{m}_w^* = -u_w^* \Delta y = -\overline{u_w^*} + d_x \left[\underbrace{\left(p_C^{(n)} - p_W^{(n)}\right)}_{\substack{\text{pressure difference} \\ \text{across face}}} - \underbrace{0.5\left(\left(p_e^{(n)} - p_w^{(n)}\right) + \left(p_w^{(n)} - p_{ww}^{(n)}\right)\right)}_{\substack{\text{average pressure difference} \\ \text{across face}}} \right]$$

$$= d_x\left(p_C^{(n)} - p_W^{(n)}\right)$$

$$= 1(70 - 100)$$

$$= -30$$

with $\dot{m}_s = -20\Delta x = -20$ the pressure correction equation is constructed from

$$\left(\dot{m}_e^* + \dot{m}_e'\right) + \left(\dot{m}_n^* + \dot{m}_n'\right) + \left(\dot{m}_w^* + \dot{m}_w'\right) + \dot{m}_s = 0$$

and

$$\begin{aligned}\dot{m}'_e + \dot{m}'_n + \dot{m}'_w &= -\left(\dot{m}^*_e + \dot{m}^*_n + \dot{m}^*_w + \dot{m}_s\right)\\ &= -(20 + 12.5 - 30 - 20)\\ &= 17.5\end{aligned}$$

now

$$\begin{aligned}\dot{m}'_e &= -d_x\left(p'_E - p'_C\right)\\ \dot{m}'_n &= -d_y\left(p'_N - p'_C\right)\\ \dot{m}'_w &= d_x\left(p'_C - p'_W\right)\end{aligned}$$

thus we have

$$-\left(p'_E - p'_C\right) - 0.25\left(p'_N - p'_C\right) + \left(p'_C - p'_W\right) = 17.5$$

or

$$2.25p'_C - p'_E - 0.25p'_N - p'_W = 17.5$$

if the E, N and W values were exact then we would have

$$2.25p'_C = 17.5$$

or

$$p'_C = 7.78$$

then we would proceed with correcting the pressure and velocity components at C to yield

$$\begin{aligned}p^*_C = p^{(n)}_C + p'_C &= 77.78\\ u'_C &= 0 \Rightarrow u^{**}_C = 25\\ v'_C &= -d_y\left(p'_n - p'_s\right)\\ &= -0.25\left(0.5p'_C - p'_C\right)\\ &= -0.25(0.5 \times 7.78 - 7.78)\\ &= 0.9725\end{aligned}$$

so

$$\begin{aligned}v^{**}_C &= 6.25 + 0.9725\\ &= 7.2225\end{aligned}$$

15.6 Boundary Conditions

A **boundary element** has at least one face located at a boundary patch, which is denoted as a **boundary face** (Fig. 15.18). The treatment of boundary conditions at a boundary face is critical to the accuracy and robustness of any CFD code. Thus for any pressure-based code to succeed, it is imperative for the boundary conditions of both momentum and pressure-correction equations to be properly implemented. This section describes in detail the implementation of a variety of boundary conditions.

A first note of interest is the expression of the Rhie-Chow interpolation at a boundary face, which has to be modified since the averaging cannot be performed in a fashion similar to an interior face. The average at a boundary face is written in terms of the element value as

$$\overline{\square}_b = \square_C \tag{15.109}$$

where b refers to the boundary face centroid and C to the element centroid. Adopting this practice, the averages in the Rhie-Chow interpolation, the velocity, and the mass flow rate at a boundary face become

$$\begin{aligned} \overline{\mathbf{v}_b^*} &= \mathbf{v}_C^* \\ \overline{\nabla p_b^{(n)}} &= \nabla p_C^{(n)} \\ \overline{\mathbf{D}_b^{\mathbf{v}}} &= \mathbf{D}_C^{\mathbf{v}} \end{aligned} \tag{15.110}$$

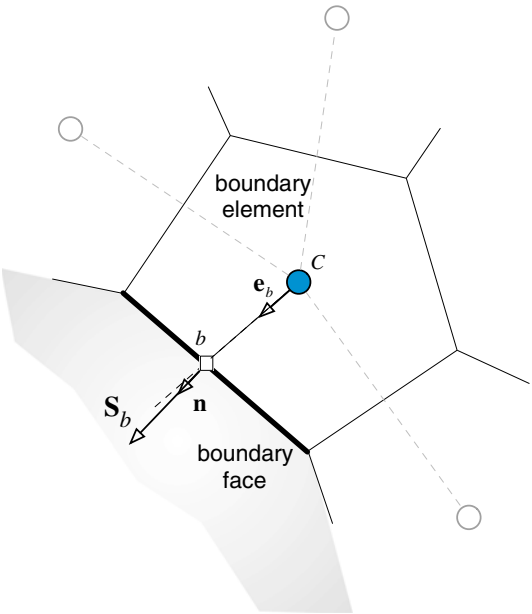


Fig. 15.18 An example of a boundary element

$$\begin{aligned}\underbrace{\mathbf{v}_b^*}_{\text{boundary face}} &= \underbrace{\overline{\mathbf{v}_b^*} - \mathbf{D}_C^{\mathbf{v}}\left(\nabla p_b^{(n)} - \overline{\nabla p_b^{(n)}}\right)}_{\text{standard Rhie-Chow}} \\ &= \underbrace{\mathbf{v}_C^* - \mathbf{D}_C^{\mathbf{v}}\left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right)}_{\text{boundary Rhie-Chow}}\end{aligned} \tag{15.111}$$

$$\begin{aligned}\dot{m}_b^* &= \rho_b \mathbf{v}_b^* \cdot \mathbf{S}_b \\ &= \rho_b \left[\mathbf{v}_C^* - \mathbf{D}_C^{\mathbf{v}}\left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right)\right] \cdot \mathbf{S}_b \\ &= \rho_b \mathbf{v}_C^* \cdot \mathbf{S}_b - \rho_b \mathcal{D}_C\left(p_b^{(n)} - p_C^{(n)}\right) - \rho_b\left(\mathbf{D}_C^{\mathbf{v}} \nabla p_b^{(n)} \cdot \mathbf{T}_b - \mathbf{D}_C^{\mathbf{v}} \nabla p_C^{(n)} \cdot \mathbf{S}_b\right)\end{aligned} \tag{15.112}$$

In what follows the implementation of boundary conditions is first presented for the momentum equation, followed by the implementation of the boundary conditions for the pressure (correction) equation. For the cases when the boundary conditions for the momentum and pressure correction equations are co-dependent, their full treatment is detailed in the pressure correction equation section.

15.6.1 Boundary Conditions for the Momentum Equation

The semi-discretized form of the momentum equation can be expressed as

$$\underbrace{\frac{(\rho\mathbf{v})_C - (\rho\mathbf{v})_C^{\circ}}{\Delta t} V_C}_{\text{element discretization}} + \underbrace{\sum_{f\sim nb(C)} \left(\dot{m}_f \mathbf{v}_f\right)}_{\text{face discretization}} = -\underbrace{\sum_{f\sim nb(C)} \left(p_f \mathbf{S}_f\right)}_{\text{face discretization}} + \underbrace{\sum_{f\sim nb(C)} \left(\tau_f \cdot \mathbf{S}_f\right)}_{\text{face discretization}} + \underbrace{\mathbf{B}}_{\substack{\text{element} \\ \text{discretization}}} \tag{15.113}$$

where the discretization type of the various terms is explicitly stated. Terms that are evaluated at the element faces should be modified along a boundary face in accordance with the type of boundary condition used. Therefore, for boundary elements, the terms evaluated at the element faces are written as

$$\underbrace{\sum_{f\sim nb(C)} \left(\dot{m}_f \mathbf{v}_f\right)}_{\text{face discretization}} = \sum_{f\sim \text{interior } nb(C)} \left(\dot{m}_f \mathbf{v}_f\right) + \underbrace{\dot{m}_b \mathbf{v}_b}_{\text{boundary face}} \tag{15.114}$$

$$\underbrace{\sum_{f\sim nb(C)}\left(\tau_f\cdot\mathbf{S}_f\right)}_{\text{face discretization}}=\sum_{f\sim \text{interior } nb(C)}\left(\tau_f\cdot\mathbf{S}_f\right)+\underbrace{\tau_b\cdot\mathbf{S}_b}_{\text{boundary face}}$$
$$=\sum_{f\sim \text{interior } nb(C)}\left(\tau_f\cdot\mathbf{S}_f\right)+\underbrace{\mathbf{F}_b}_{\substack{\text{boundary}\\ \text{face}}} \tag{15.115}$$

$$\underbrace{\sum_{f\sim nb(C)}\left(p_f\mathbf{S}_f\right)}_{\text{face discretization}}=\sum_{f\sim \text{interior } nb(C)}\left(p_f\mathbf{S}_f\right)+\underbrace{p_b\mathbf{S}_b}_{\text{boundary face}} \tag{15.116}$$

where subscript b refers to a value at the boundary. As previously stated, the pressure term may be discretized following either an element or a face discretization. In either case the expanded form is the same since $V_C(\nabla p)_C$ is calculated as $\sum_{f\sim nb(C)} p_f\mathbf{S}_f$ implying that the value at the boundary is always required. To show the way boundary pressure affects solution, the expanded form of the pressure gradient (i.e., face discretization) is adopted in the implementation of boundary conditions.

15.6.1.1 Wall Boundary Conditions

Generally a no-slip or a slip boundary condition may be applied to a moving or a stationary wall. The implementation involves computing and linearizing the shear stress at the wall. This is different from the Dirichlet condition, though for a cartesian grid the two conditions may be shown to be identical.

No-Slip Wall Boundary $(p_b = ?; \dot{m}_b = 0; \mathbf{v}_b = \mathbf{v}_{wall})$

A no slip boundary condition implies that the velocity of the fluid at the wall $\mathbf{v}_b$ is equal to the velocity of the wall $\mathbf{v}_{wall}$. For a stationary wall, the boundary velocity $\mathbf{v}_b$ is zero. The known velocity at the wall could be wrongly viewed as a Dirichlet boundary condition. However this is not really the case, since what is needed is to have a zero normal boundary flux while also accounting for the shear stress. This cannot be satisfied by simply setting $\mathbf{v}_b = \mathbf{v}_{wall}$. Figure 15.19 shows that this can be guaranteed by ensuring a shear stress that is tangential to the wall in addition to a boundary velocity equation that is equal to the wall velocity. The force $\mathbf{F}_b$ exerted by the wall on the fluid can be written as

$$\mathbf{F}_b = \mathbf{F}_\perp + \mathbf{F}_\parallel \tag{15.117}$$

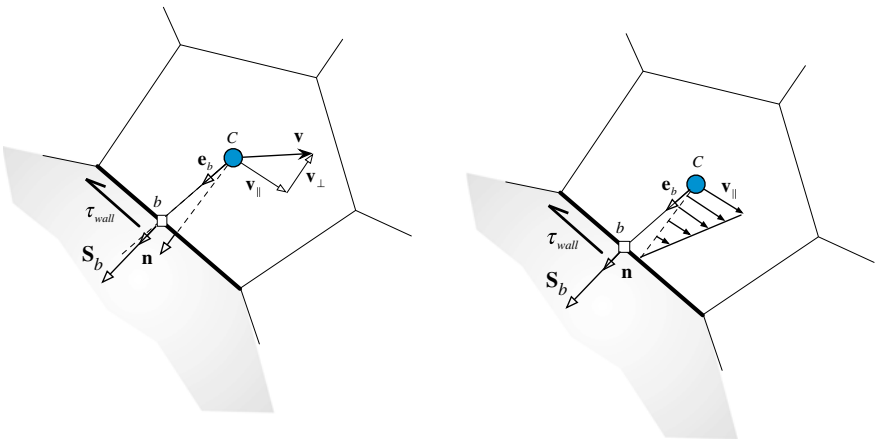


Fig. 15.19 A schematic of a no-slip wall boundary condition

where $\mathbf{F}_{\parallel}$ is in the tangential direction to the wall and $\mathbf{F}_{\perp}$ is in the normal direction, which is supposed to be zero. Thus

$$\mathbf{F}_b = \mathbf{F}_{\parallel} = \tau_{wall} S_b \tag{15.118}$$

where τ_{wall} is the shear stress exerted by the wall on the fluid given by

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{\parallel}}{\partial d_{\perp}}. \tag{15.119}$$

In Eq. (15.119) $\mathbf{v}_{\parallel}$ is the velocity vector in the direction parallel to the wall and $d_{\perp}$ is the normal distance from the centroid of the boundary element to the wall computed as

$$\begin{aligned} \mathbf{n} &= \frac{\mathbf{S}_b}{S_b} = n_x \mathbf{i} + n_y \mathbf{j} + n_z \mathbf{k} \\ d_{\perp} &= \mathbf{d}_{Cb} \cdot \mathbf{n} = \frac{\mathbf{d}_{Cb} \cdot \mathbf{S}_b}{S_b} \end{aligned} \tag{15.120}$$

and $\mathbf{n}$ the wall normal unit vector. The velocity vector $\mathbf{v}_{\parallel}$ can be written as the difference between $\mathbf{v}$ and its normal component as

$$\mathbf{v}_{\parallel} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n} = \left\{ \begin{array}{c} u - \left(u n_x + v n_y + w n_z\right) n_x \\ v - \left(u n_x + v n_y + w n_z\right) n_y \\ w - \left(u n_x + v n_y + w n_z\right) n_z \end{array} \right\} \tag{15.121}$$

From Eq. (15.119), the wall shear stress can be approximated using

$$\begin{aligned}\tau_{wall} &\approx -\mu_b \frac{(\mathbf{v}_C - \mathbf{v}_b)_{||}}{d_\perp} = -\mu_b \frac{(\mathbf{v}_C - \mathbf{v}_b) - [(\mathbf{v}_C - \mathbf{v}_b)\cdot\mathbf{n}]\mathbf{n}}{d_\perp} \\ &= -\frac{\mu_b}{d_\perp}\left\{\begin{array}{l}(u_C - u_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_x \\ (v_C - v_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_y \\ (w_C - w_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_z\end{array}\right\}\end{aligned} \tag{15.122}$$

from which the boundary force for a laminar flow can be obtained as

$$\mathbf{F}_b = -\frac{\mu_b S_b}{d_\perp}\left\{\begin{array}{l}(u_C - u_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_x \\ (v_C - v_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_y \\ (w_C - w_b) - \big[(u_C - u_b)n_x + (v_C - v_b)n_y + (w_C - w_b)n_z\big]n_z\end{array}\right\} \tag{15.123}$$

Using Eq. (15.123) the coefficients of the boundary elements for the momentum equation in the x, y and z directions are modified as follows:

u-component equation

$$\begin{aligned} a_C^u &\leftarrow \underbrace{a_C^u}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}(1 - n_x^2)}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^u \\ b_C^u &\leftarrow \underbrace{b_C^u}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}\big[u_b(1 - n_x^2) + (v_C - v_b)n_y n_x - (w_C - w_b)n_z n_x\big] - p_b S_b^x}_{\text{boundary face contribution}} \end{aligned} \tag{15.124}$$

v-component equation

$$\begin{aligned} a_C^v &\leftarrow \underbrace{a_C^v}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}\left(1 - n_y^2\right)}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^v \\ b_C^v &\leftarrow \underbrace{b_C^v}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}\left[(u_C - u_b)n_x n_y + v_b\left(1 - n_y^2\right) + (w_C - w_b)n_z n_y\right] - p_b S_b^y}_{\text{boundary face contribution}} \end{aligned} \tag{15.125}$$

w-component equation

$$
\begin{aligned}
a_C^w &\leftarrow \underbrace{a_C^w}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}\left(1-n_z^2\right)}_{\text{boundary face contribution}} \\
0 &\leftarrow a_{F=b}^w \\
b_C^w &\leftarrow \underbrace{b_C^w}_{\text{interior faces contribution}} + \underbrace{\frac{\mu_b S_b}{d_\perp}\left[(u_C-u_b)n_x n_z + (v_C-v_b)n_y n_z + w_b\left(1-n_z^2\right)\right] - p_b S_b^z}_{\text{boundary face contribution}}
\end{aligned}
\tag{15.126}
$$

The unknown boundary pressure p_b is extrapolated from the interior solution using either a truncated Taylor series expansion around point C such that pressure is found from

$$p_b = p_C + \nabla p_C^{(n)} \cdot \mathbf{d}_{Cb} \tag{15.127}$$

or the mass flow rate expressed via the Rhie-Chow interpolation as

$$\dot{m}_b^* = \rho_b \mathbf{v}_b^* \cdot \mathbf{S}_b - \rho_b \mathbf{D}_C^{\mathbf{v}} \left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right) \cdot \mathbf{S}_b \tag{15.128}$$

Since the mass flow rate and velocity at the wall boundary are zero, the above equation reduces to

$$0 = 0 - \rho_b \mathbf{D}_C^{\mathbf{v}} \left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right) \cdot \mathbf{S}_b \tag{15.129}$$

which can be modified into

$$
\begin{aligned}
\mathbf{D}_C^{\mathbf{v}} \nabla p_b^{(n)} \cdot \mathbf{S}_b &= \nabla p_b^{(n)} \cdot \mathbf{S}'_b \\
&= \nabla p_C^{(n)} \cdot \mathbf{S}'_b
\end{aligned}
\tag{15.130}
$$

Expressing $\mathbf{S}'_b$ as the sum of the two vector $\mathbf{E}_b$ and $\mathbf{T}_b$, Eq. (15.130) becomes

$$\nabla p_b^{(n)} \cdot (\mathbf{E}_b + \mathbf{T}_b) = \nabla p_C^{(n)} \cdot \mathbf{S}'_b \tag{15.131}$$

Since $\mathbf{E}_b$ is in the direction of $\mathbf{Cb}$, the above equation can be modified to

$$\mathcal{D}_C (p_b - p_C) = \left(\nabla p_C^{(n)} \cdot \mathbf{S}'_b - \nabla p_b^{(n)} \cdot \mathbf{T}_b\right) \tag{15.132}$$

from which the boundary pressure is obtained as

$$p_b = p_C + \frac{\left(\nabla p_C^{(n)} \cdot \mathbf{S}'_b - \nabla p_b^{(n)} \cdot \mathbf{T}_b\right)}{\mathcal{D}_C} \tag{15.133}$$

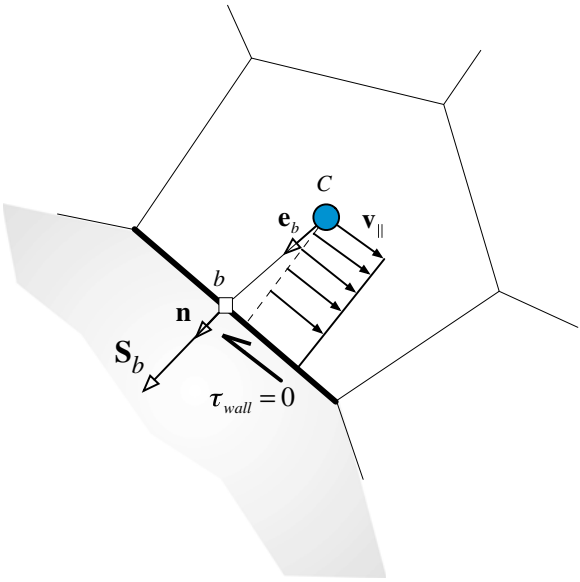


Fig. 15.20 A schematic of a slip wall boundary condition

Slip Wall Boundary $(p_b = ?; \dot{m}_b = 0; \mathbf{F}_b = 0)$

For this boundary condition, the wall shear stress is zero (Fig. 15.20). Therefore the boundary force is zero. The boundary pressure is computed as for the no-slip wall boundary case using Eq. (15.127) or Eq. (15.133). The coefficients of the momentum equation (in vector form) are modified as follows:

$$\begin{aligned} a_C^{\mathbf{v}} &\leftarrow \underbrace{a_C^{\mathbf{v}}}_{\text{interior faces contribution}} \\ 0 &\leftarrow a_{F=b}^{\mathbf{v}} \\ \mathbf{b}_C^{\mathbf{v}} &\leftarrow \underbrace{\mathbf{b}_C^{\mathbf{v}}}_{\text{interior faces contribution}} - \underbrace{p_b \mathbf{S}_b}_{\text{boundary face contribution}} \end{aligned} \tag{15.134}$$

15.6.1.2 Inlet Boundary Conditions

Three types of inlet boundary conditions are considered. (i) specified velocity; (ii) specified static pressure and velocity direction; and (iii) specified total pressure and velocity direction. All treatments of the pressure boundary conditions will be detailed in the *pressure correction boundary conditions* section.

Specified Velocity ($p_b = ?; \dot{m}_b$ *specified*; $\mathbf{v}_b$ *specified*)

For a specified velocity boundary condition at inlet (Fig. 15.21) the convection ($\dot{m}_b \mathbf{v}_b$) and diffusion ($\mathbf{F}_b = \tau_b \cdot \mathbf{S}_b$) terms at the boundary face are calculated using the known values of velocity $\mathbf{v}_b$ and mass flow rate $\dot{m}_b$. The pressure at the boundary is extrapolated from the boundary element centroid as

$$p_b = p_C + \nabla p_C^{(n)} \cdot \mathbf{d}_{Cb} \tag{15.135}$$

The terms involving the boundary velocity are treated explicitly by adding them to the source term and setting the coefficient $a^{\mathbf{v}}_{F=b}$ to zero while adding its value to the $a^{\mathbf{v}}_C$ coefficient.

The coefficients of the boundary element are modified according to

$$\begin{aligned} a^{\mathbf{v}}_C &\leftarrow a^{\mathbf{v}}_C \\ b^{\mathbf{v}}_C &\leftarrow b^{\mathbf{v}}_C - a^{\mathbf{v}}_{F=b}\mathbf{v}_b \\ 0 &\leftarrow a^{\mathbf{v}}_{F=b} \end{aligned} \tag{15.136}$$

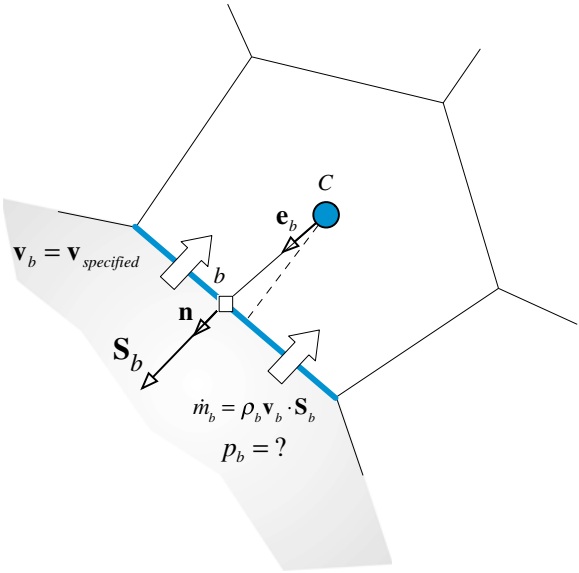


Fig. 15.21 A schematic of specified velocity boundary condition at inlet

Specified Pressure and Velocity Direction $(p_b = p_{specified};\ \dot{m}_b?;\ \mathbf{e_v}\ specified;\ \mathbf{v}_b?)$

In the case of a specified static pressure at inlet (Fig. 15.22), p_b is known. The velocity being unknown, has to be computed from the pressure gradient at the boundary. To this end, a velocity direction should be specified as part of the boundary condition.

As the boundary pressure p_b is known, its value is directly used in calculating the pressure gradient in the boundary element without any special treatment. Therefore p_b is used in calculating ∇p_C.

The mass flow rate at the boundary is computed from the continuity equation (see next section). Then, for a specified velocity direction given by the unit vector $\mathbf{e_v}$, the velocity for a specified pressure boundary condition at inlet is obtained as

$$\dot{m}_b^{**} = \rho_b \mathbf{v}_b^{**} \cdot \mathbf{S}_b = \rho_b \left\|\mathbf{v}_b^{**}\right\| \mathbf{e_v} \cdot \mathbf{S}_b \Rightarrow \left\|\mathbf{v}_b^{**}\right\| = \frac{\dot{m}_b^{**}}{\rho_b (\mathbf{e_v} \cdot \mathbf{S}_b)} \Rightarrow \mathbf{v}_b^{**} = \left\|\mathbf{v}_b^{**}\right\| \mathbf{e_v} \tag{15.137}$$

The velocity at the boundary is computed at every iteration and the problem is solved as in the case of a specified velocity with the coefficients in the momentum equation modified according to Eq. (15.136).

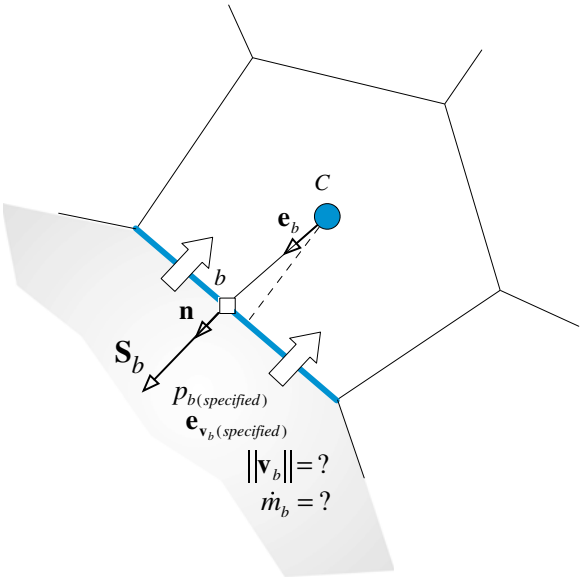


Fig. 15.22 A schematic of specified pressure and velocity direction boundary condition at inlet

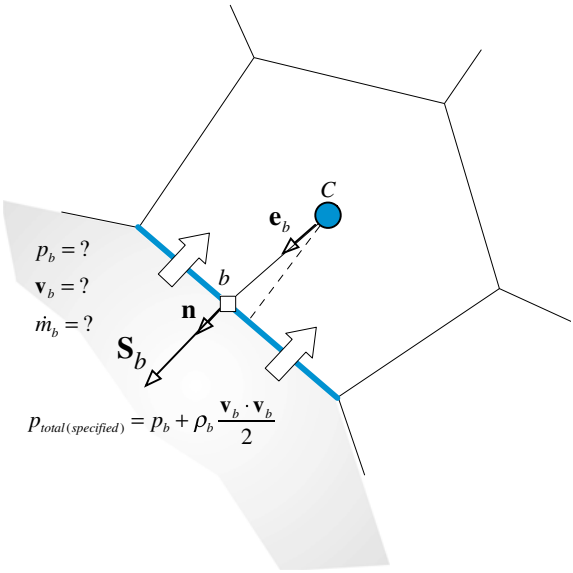


Fig. 15.23 A schematic of specified total pressure and velocity direction boundary condition at inlet

Specified Total Pressure and Velocity Direction $(p_{o,b} = p_{o,specified}; \dot{m}_b?; \mathbf{e}_\mathbf{v}\ specified; \mathbf{v}_b?)$

In the case of a specified total pressure at inlet (Fig. 15.23) the velocity direction should also be specified. However, the magnitude of the velocity and the pressure at the boundary are unknown though related using the total pressure definition given by

$$p_0 = \underbrace{p}_{\text{static pressure}} + \underbrace{\frac{1}{2}\rho \mathbf{v} \cdot \mathbf{v}}_{\text{dynamic pressure}} \tag{15.138}$$

where p_0 is the total pressure, p the static pressure, ρ the density, and $\mathbf{v}$ the velocity vector. The mass flow rate at the boundary is computed from the continuity equation (see next section). Knowing the mass flow rate, the velocity is computed in the same manner as for the specified pressure case using Eq. (15.137). The velocity is thus treated as a known velocity condition (i.e., a Dirichlet boundary condition) with the coefficients in the momentum equation modified according to Eq. (15.136).

15.6.1.3 Outlet Boundary Conditions

Three types of boundary conditions at an outlet are considered: (i) a specified static pressure, (ii) a specified mass flow rate, and (iii) a fully developed flow.

Specified Static Pressure $\left(p_b = p_{specified}; \dot{m}_b?; \mathbf{v}_b?\right)$

For the momentum equation, fully developed conditions are assumed at a specified pressure outlet (Fig. 15.24) implying a zero velocity gradient along the direction of the surface vector at outlet. This is also equivalent to assuming the velocity at the outlet to be equal to that of the boundary element. Thus it is similar to a zero flux boundary condition whose implementation is rather simple.

The modifications to the coefficients are given by

$$\begin{aligned} a_C^{\mathbf{v}} &\leftarrow \underbrace{a_C^{\mathbf{v}}}_{\text{interior faces contribution}} + \underbrace{\dot{m}_b}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^{\mathbf{v}} \\ \mathbf{b}_C^{\mathbf{v}} &\leftarrow \underbrace{\mathbf{b}_C^{\mathbf{v}}}_{\text{interior faces contribution}} - \underbrace{p_b \mathbf{S}_b}_{\text{boundary face contribution}} \end{aligned} \tag{15.139}$$

This sets the contribution of the outlet velocity to zero and uses the specified pressure boundary value in the computation of the pressure gradient.

However to ensure that the flux is zeroed in the outflow surface vector direction only, the velocity is usually extrapolated to the outlet using the boundary flux, which is computed from the boundary element flux as

$$\nabla \mathbf{v}_b = \nabla \mathbf{v}_C - (\nabla \mathbf{v}_C \cdot \mathbf{e}_b)\mathbf{e}_b \tag{15.140}$$

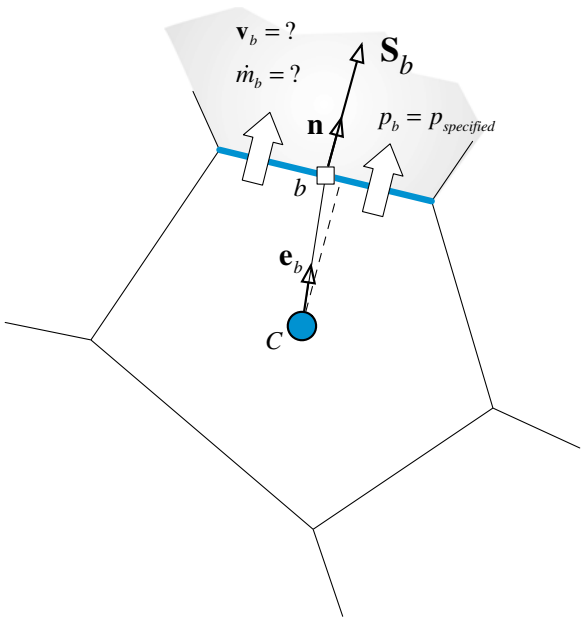


Fig. 15.24 A schematic of specified static pressure boundary condition at outlet

This ensures that the gradient along the boundary surface vector is zero. Then, using a Taylor series expansion, the velocity at the boundary is computed as

$$\mathbf{v}_b = \mathbf{v}_C + \nabla \mathbf{v}_b \cdot \mathbf{d}_{Cb} \tag{15.141}$$

where $\nabla \mathbf{v}_b$ is used instead of $\nabla \mathbf{v}_C$. Therefore an additional correction is now added to the source term and the modifications to the coefficients become

$$\begin{aligned} a_C^{\mathbf{v}} &\leftarrow \underbrace{a_C^{\mathbf{v}}}_{\text{interior faces contribution}} + \underbrace{\dot{m}_b}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^{\mathbf{v}} \\ \mathbf{b}_C^{\mathbf{v}} &\leftarrow \underbrace{\mathbf{b}_C^{\mathbf{v}}}_{\text{interior faces contribution}} \underbrace{-\dot{m}_b(\nabla \mathbf{v}_b \cdot \mathbf{d}_{Cb}) - p_b \mathbf{S}_b}_{\text{boundary face contribution}} \end{aligned} \tag{15.142}$$

Specified Mass Flow Rate $\left(\dot{m}_b = \dot{m}_{specified}; p_b ? \mathbf{v}_b ?\right)$

Since the flow is incompressible, a specified mass flow rate boundary condition (Fig. 15.25) is equivalent to specifying the normal component of velocity at the boundary. The velocity is calculated by assuming its direction to be the same as that at the main grid point, i.e., $(\mathbf{e}_\mathbf{v})_b = (\mathbf{e}_\mathbf{v})_C$. Thus, the velocity is expressed as

$$\mathbf{v}_b = |\mathbf{v}_b|(\mathbf{e}_\mathbf{v})_C \tag{15.143}$$

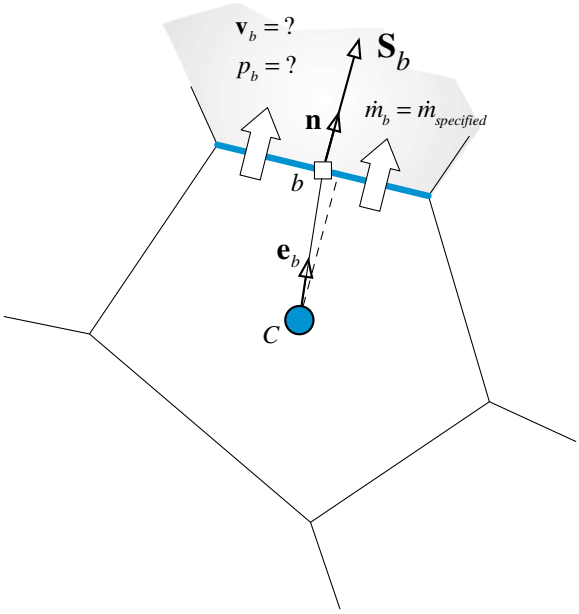


Fig. 15.25 A schematic of specified mass flow rate boundary condition at outlet

with $|\mathbf{v}_b|$ obtained from

$$\dot{m}_b = \rho_b \mathbf{v}_b \cdot \mathbf{S}_b = \rho_b |\mathbf{v}_b| (\mathbf{e}_\mathbf{v})_C \cdot \mathbf{S}_b \Rightarrow |\mathbf{v}_b| = \frac{\dot{m}_b}{\rho_b (\mathbf{e}_\mathbf{v})_C \cdot \mathbf{S}_b} \tag{15.144}$$

allowing $\mathbf{v}_b$ to be calculated. Thus for momentum, a specified velocity boundary condition is applied. The coefficients of the boundary elements are modified according to Eq. (15.136).

Fully Developed Outlet Flow

For a fully developed flow, the velocity gradient normal to the outlet surface is assumed to be zero. Hence the velocity at the outlet is assumed to be known and computed from the zero normal gradient using Eqs. (15.140) and (15.141). As for the pressure at the boundary, it can be extrapolated from the interior pressure field using

$$p_b = p_C + \nabla p_C \cdot \mathbf{d}_{Cb} \tag{15.145}$$

The velocity is treated as known and the coefficients of the momentum equation are modified according to Eq. (15.142).

15.6.1.4 Symmetry Boundary Condition

A scalar is reflected across a symmetry boundary. Thus, a symmetry boundary condition for a scalar variable is imposed by setting its normal gradient to zero, as with an insulated wall boundary condition. For the velocity vector, the symmetry condition shown in Fig. 15.26 also implies that it is reflected about the symmetry boundary with its parallel component (i.e., parallel to the symmetry boundary) retaining magnitude and direction, while its normal component becoming zero. This results in a zero shear stress but a non-zero normal stress along the symmetry boundary. Thus, the same boundary condition can be used to impose a slip wall boundary condition for viscous flows.

The unit vector in the direction normal to the boundary $\mathbf{n}$ and the normal distance to the boundary $d_\perp$ are given by Eq. (15.120). Therefore the velocity components normal and parallel to a symmetry boundary satisfy

$$\begin{aligned} \mathbf{v}_\perp &= 0 \\ \frac{\partial \mathbf{v}_{||}}{\partial n} &= 0 \end{aligned} \tag{15.146}$$

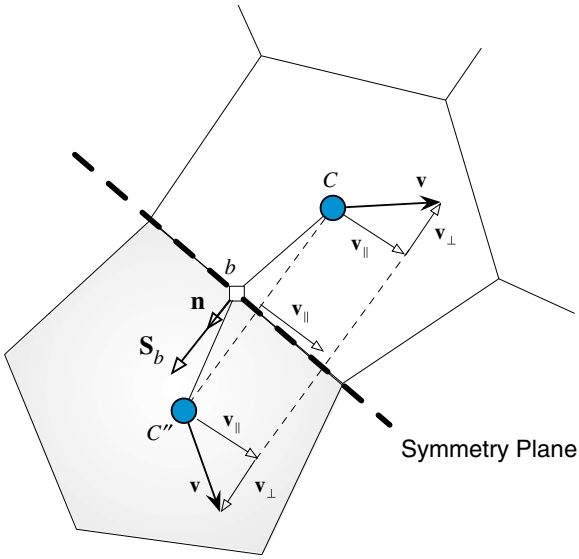


Fig. 15.26 A schematic of a symmetry boundary condition

The normal component of velocity can be written as

$$\mathbf{v}_\perp = (\mathbf{v}\cdot\mathbf{n})\mathbf{n} = \left\{ \begin{array}{l} \left(u_C n_x + v_C n_y + w_C n_z\right) n_x \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_y \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_z \end{array} \right\} \tag{15.147}$$

while the parallel component is given by Eq. (15.121). The boundary force $\mathbf{F}_b$ can be decomposed into a normal component $\mathbf{F}_\perp$ and a parallel component $\mathbf{F}_{||}$. As the shear stress along a symmetry boundary is zero, the parallel component of $\mathbf{F}_b$ is zero. Denoting the normal stress by $\sigma_\perp$, the force $\mathbf{F}_b$ is given by

$$\mathbf{F}_b = \sigma_\perp S_b \tag{15.148}$$

The normal stress component can be approximated as

$$\sigma_\perp \simeq -2\mu_b \frac{\mathbf{v}_\perp}{d_\perp} = -2\frac{\mu_b}{d_\perp} \left\{ \begin{array}{l} \left(u_C n_x + v_C n_y + w_C n_z\right) n_x \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_y \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_z \end{array} \right\} \tag{15.149}$$

from which the boundary force is found to be

$$\mathbf{F}_b = \mathbf{F}_n = -2\frac{\mu_b S_b}{d_\perp} \left\{ \begin{array}{l} \left(u_C n_x + v_C n_y + w_C n_z\right) n_x \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_y \\ \left(u_C n_x + v_C n_y + w_C n_z\right) n_z \end{array} \right\} \tag{15.150}$$

The pressure gradient in the direction normal to a symmetry boundary is zero. Mathematically this is written as

$$\nabla p_b \cdot \mathbf{n} = 0 \tag{15.151}$$

The pressure at a symmetry boundary should be extrapolated from the interior of the domain. Therefore to ensure a zero normal gradient, the pressure gradient at the symmetry boundary is computed as

$$\nabla p_b = \nabla p_C - (\nabla p_C \cdot \mathbf{n})\mathbf{n} \tag{15.152}$$

Thus, the pressure is obtained from

$$p_b = p_C + \nabla p_b \cdot \mathbf{d}_{Cb} \tag{15.153}$$

Using the above equations, the coefficients of the boundary elements for the momentum equation in the x, y and z directions are modified as follows:

u-component equation

$$\begin{aligned} a_C^u &\leftarrow \underbrace{a_C^u}_{\text{interior faces contribution}} + \underbrace{\frac{2\mu_b S_b}{d_\perp} n_x^2}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^u \\ b_C^u &\leftarrow \underbrace{b_C^u}_{\text{interior faces contribution}} \underbrace{-\frac{2\mu_b S_b}{d_\perp}\left[v_C n_y + w_C n_z\right] n_x - p_b S_b^x}_{\text{boundary face contribution}} \end{aligned} \tag{15.154}$$

v-component equation

$$\begin{aligned} a_C^v &\leftarrow \underbrace{a_C^v}_{\text{interior faces contribution}} + \underbrace{\frac{2\mu_b S_b}{d_\perp} n_y^2}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^v \\ b_C^v &\leftarrow \underbrace{b_C^v}_{\text{interior faces contribution}} \underbrace{-\frac{2\mu_b S_b}{d_\perp}\left[u_C n_x + w_C n_z\right] n_y - p_b S_b^y}_{\text{boundary face contribution}} \end{aligned} \tag{15.155}$$

w-component equation

$$\begin{aligned} a_C^w &\leftarrow \underbrace{a_C^w}_{\text{interior faces contribution}} + \underbrace{\frac{2\mu_b S_b}{d_\perp} n_z^2}_{\text{boundary face contribution}} \\ 0 &\leftarrow a_{F=b}^w \\ b_C^w &\leftarrow \underbrace{b_C^w}_{\text{interior faces contribution}} \underbrace{-\frac{2\mu_b S_b}{d_\perp}\left[u_C n_x + v_C n_y\right] n_z - p_b S_b^z}_{\text{boundary face contribution}} \end{aligned} \tag{15.156}$$

While this is not a comprehensive list of momentum boundary conditions, it does cover the most common types.

15.6.2 Boundary Conditions for the Pressure Correction Equation

The continuity equation for a boundary cell can be written as

$$\sum_{f\sim nb(C)} \dot{m}_f + \underbrace{\dot{m}_b}_{\text{boundary face}} = 0 \tag{15.157}$$

or

$$\sum_{f\sim nb(C)} (\dot{m}_f^* + \dot{m}_f') + \underbrace{(\dot{m}_b^* + \dot{m}_b')}_{\text{boundary face}} = 0 \tag{15.158}$$

where $\dot{m}_b^*$ is the boundary mass flux and $\dot{m}_b'$ is its correction. While for an internal face the mass flux and its correction are defined by Eqs. (15.100) and (15.101), for a boundary face the definition is slightly different. Since at a boundary face only the boundary cell contributes to the average quantities, the use of Eq. (15.109) in combination with Eqs. (15.100) and (15.101) gives

$$\begin{aligned} \dot{m}_b^* &= \rho_b \mathbf{v}_C^* \cdot \mathbf{S}_b - \rho_b \mathbf{D}_C^{\mathbf{v}} (\nabla p_b^{(n)} - \nabla p_C^{(n)}) \cdot \mathbf{S}_b \\ \dot{m}_b' &= -\rho_b \mathcal{D}_C (p_b' - p_C') \end{aligned} \tag{15.159}$$

In implementing boundary conditions the values of $\dot{m}_b^*$, $\dot{m}_b'$, p_b, and p_b' must be calculated. Based on the discussions related to the momentum equation, three types of boundary conditions can be inferred. The first type is designated by "specified mass flow rate" (e.g., walls or velocity specified at inlets). For this category $\dot{m}_b' = 0$, which is similar to a zero scalar flux boundary condition, and no modification to the pressure-correction equation is needed. The pressure however has to be computed at

the boundary from the interior field. The second type of boundary conditions is termed "pressure specified" where $p'_b = 0$ and for which a Dirichlet-like condition has to be enforced for the pressure-correction equation. For this condition, $\dot{m}^*_b$ is computed from the boundary and interior pressure field. In the third type, an implicit relation exists between the pressure and the mass flow rate, as in a specified total pressure boundary condition. In this case, an explicit equation is extracted from the implicit relation and substituted into the pressure-correction equation.

Details regarding the various types of boundary conditions and their implementation are now given.

15.6.2.1 Wall Boundary Condition

$$(p_b = ?; \dot{m}_b = 0; \mathbf{v}_b = \mathbf{v}_{wall}) \quad \text{or} \quad (p_b = ?; \dot{m}_b = 0; \mathbf{F}_b = 0)$$

Whether it is a slip (Fig. 15.20) or no-slip (Fig. 15.19) wall boundary condition the mass flow rate is zero. Therefore $\dot{m}'_b = 0$, which is equivalent to a specified zero flux and implying that no modification is needed for the pressure-correction equation. However the pressure at the wall is required and is computed using Eq. (15.127) or Eq. (15.133) or a low order extrapolation profile, as shown below.

$$p_b = \begin{cases} p_C + \nabla p_C^{(n)} \cdot \mathbf{d}_{Cb} & Eq.\ (15.127) \\ p_C + \dfrac{\left(\nabla p_C^{(n)} \cdot \mathbf{S}'_b - \nabla p_b^{(n)} \cdot \mathbf{T}_b\right)}{\mathcal{D}_C} & Eq.\ (15.133) \\ p_C & loworder\ extrapolation \end{cases} \tag{15.160}$$

15.6.2.2 Inlet Boundary Conditions

Specified Velocity $(p_b = ?; \dot{m}_b\ specified; \mathbf{v}_b\ specified)$

For a specified velocity at inlet (Fig. 15.21), the mass flux is known and its correction is set to zero, i.e., $\dot{m}'_b = 0$. Thus, similar to a wall boundary condition, the term is simply dropped from the pressure-correction equation. The pressure at the boundary is extrapolated from the internal pressure field using Eq. (15.127) or Eq. (15.133) or a low order extrapolation profile as summarized in Eq. (15.160).

Specified Pressure and Velocity Direction $(p_b = p_{specified}; \dot{m}_b?; \mathbf{e}_\mathbf{v}\ specified; \mathbf{v}_b?)$

In the case of a specified static pressure at inlet (Fig. 15.22), p_b is known and thus p'_b is set to zero but $\dot{m}'_b \neq 0$. The inlet is treated as a Dirichlet boundary condition for the pressure-correction equation. The coefficient of the p' equation becomes

$$a_C^{p'} = \underbrace{\sum_{f\sim nb(C)} \rho_f \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\rho_b \mathcal{D}_C}_{\text{boundary face contribution}} \tag{15.161}$$

Specified Total Pressure and Velocity Direction $\left(p_{o,b} = p_{o,specified};\ \dot{m}_b?;\ \mathbf{e_v}\ specified;\ \mathbf{v}_b?\right)$

As mentioned earlier, for a specified total pressure (Fig. 15.23), the velocity direction should also be specified. The total pressure relation given by Eq. (15.138) is first rewritten as a function of the mass flow rate and pressure by replacing the velocity magnitude by the mass flux. Thus,

$$\begin{aligned} \dot{m}_b = \rho \mathbf{v}_b \cdot \mathbf{S}_b = \rho |\mathbf{v}_b| \mathbf{e_v} \cdot \mathbf{S}_b \Rightarrow \rho |\mathbf{v}_b| = \frac{\dot{m}_b}{\mathbf{e_v} \cdot \mathbf{S}_b} \\ \Rightarrow p_{o,b} = p_b + \frac{1}{2\rho_b} \frac{\dot{m}_b^2}{\left(\mathbf{e_v} \cdot \mathbf{S}_b\right)^2} \end{aligned} \tag{15.162}$$

Using a Taylor expansion about p_b, p_b' is obtained as

$$p_b + p_b' = p_b + \frac{\partial p_b}{\partial \dot{m}_b}\left(\dot{m}_b'\right) \Rightarrow p_b' = \frac{\partial p_b}{\partial \dot{m}_b} \dot{m}_b' \tag{15.163}$$

Differentiating Eq. (15.162) with respect to $\dot{m}_b$ and substituting into Eq. (15.163), the final form of p_b' is found to be

$$p_b' = -\frac{\dot{m}_b^*}{\rho_b \left(\mathbf{e_v} \cdot \mathbf{S}_b\right)^2} \dot{m}_b' = -\frac{\rho_b \mathbf{v}_b^* \cdot \mathbf{v}_b^*}{\dot{m}_b^*} \dot{m}_b' \tag{15.164}$$

Substituting Eq. (15.164) in Eq. (15.159), the mass flux correction is expressed as

$$\dot{m}_b' = -\rho_b \mathcal{D}_C \left(p_b' - p_C'\right) \Rightarrow \dot{m}_b' = \frac{\dot{m}_b^* \rho_b \mathcal{D}_C}{\dot{m}_b^* - \mathcal{D}_C \left(\rho_b \mathbf{v}_b^* \cdot \rho_b \mathbf{v}_b^*\right)} p_C' \tag{15.165}$$

Replacing $\dot{m}_b'$ in the expanded continuity equation (Eq. 15.158) by its expression from Eq. (15.165), the modified $a_C^{p'}$ coefficient for the boundary cell becomes

$$a_C^{p'} = \underbrace{\sum_{f\sim nb(C)} \rho_f \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\frac{\rho_b \dot{m}_b^* \mathcal{D}_C}{\dot{m}_b^* - \mathcal{D}_C \left(\rho_b \mathbf{v}_b^* \cdot \rho_b \mathbf{v}_b^*\right)}}_{\text{boundary face contribution}} \tag{15.166}$$

15.6.2.3 Outlet Boundary Conditions

Specified Pressure $\left(p_b = p_{specified}; \dot{m}_b?;\ \mathbf{v}_b?\right)$

For a specified pressure at outlet (Fig. 15.24) p'_b is set to zero. On the other hand, $\dot{m}'_b$ is computed as

$$\dot{m}'_b = -\rho_b \mathcal{D}_C (p'_b - p'_C) \tag{15.167}$$

The velocity direction being needed, it is customary to take the direction of $\mathbf{v}_b$ to be that of the upwind velocity $\mathbf{v}_C$. The expression of the $a_C^{p'}$ coefficient in the pressure-correction equation becomes

$$a_C^{p'} = \underbrace{\sum_{f \sim nb(C)} \rho_f \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\rho_b \mathcal{D}_C}_{\text{boundary face contribution}} \tag{15.168}$$

Specified Mass Flow Rate $\left(\dot{m}_b = \dot{m}_{specified}; p_b? \mathbf{v}_b?\right)$

For a specified mass flow rate at outlet (Fig. 15.25), $\dot{m}'_b$ is zero and is simply dropped from the pressure correction equation with no modifications required for the coefficients of the boundary elements. By setting $\dot{m}'_b$ to zero in Eq. (15.159), the pressure correction (or pressure) at the boundary is set equal to the pressure correction (or pressure) at the boundary cell centroid.

Fully Developed Outlet Flow

For a fully developed flow, the velocity at the outlet is assumed to be known and computed from the zero normal gradient. This means that $\dot{m}_b$ at the outlet is known. Therefore no correction is needed and $\dot{m}'_b$ is set to zero. However, as the boundary pressure is unknown, it is extrapolated from the interior pressure field. Since the velocity is iteratively updated, the above treatment does not guarantee overall conservation except at convergence. It is customary with incompressible flows to overcome this issue and to enforce global mass conservation at any iteration by modifying $\dot{m}_b$ at the boundary to satisfy overall mass conservation. This is done by calculating the total mass flow rate entering the domain $\sum \dot{m}_{in}$. Then based on the calculated mass flow rates at outlet boundaries, the total mass flow rate leaving

the domain $\sum \dot{m}_{out}$ is computed. The mass flow rate at an outlet is adjusted according to

$$\dot{m}_{out} \leftarrow \dot{m}_{out} \frac{\sum \dot{m}_{in}}{\sum \dot{m}_{out}} \tag{15.169}$$

To be able to apply the above treatment, the outlet should be placed far away from any recirculation zone.

15.6.2.4 Symmetry Boundary Condition

The mass flow rate across a symmetry line (Fig. 15.26) is zero and as such its correction is set to zero, i.e., $\dot{m}'_b = 0$. Thus, similar to a wall boundary condition, the mass flow rate correction term is simply dropped from the pressure-correction equation. The pressure at the boundary is extrapolated from the internal pressure field using Eq. (15.127) or Eq. (15.133) or a low order extrapolation profile as summarized in Eq. (15.160).

15.6.2.5 The Relative Nature of Pressure

For incompressible flow problems in which the normal velocities are prescribed on all boundaries, a difficulty arises due to the relative nature of pressure. In such cases, since only the pressure gradient appears in the momentum equation, there is no means for determining the absolute level of pressure, and only pressure differences have physical meaning. This indeterminacy of the pressure level leads to a singular coefficient matrix $\mathbf{A}$ and the direct solution of the system $\mathbf{A}\phi = \mathbf{b}$ fails. The singularity is easily removed by simply setting the pressure at one point in the domain to a prescribed value. The remaining pressures are then calculated relative to this value.

15.7 The SIMPLE Family of Algorithms

In the SIMPLE algorithm [13], velocity and pressure are treated in a segregated (sequential) manner, with the pressure field computed by deriving a pressure correction equation that exploits the discrete momentum equation to replace the velocity field in the continuity equation with a pressure term. In the derivation, a velocity correction term, $\overline{\mathbf{H}_f}[\mathbf{v}']$, was neglected as retaining it renders the equation unmanageable.

Discarding this term does not affect the final solution since its value is zero at convergence. Rather, it affects the path to convergence because during the initial

iterations its value can be significant. This large value may either cause divergence or slow down the rate of convergence as a result of an overestimated pressure correction field. To counterbalance that, in SIMPLE the pressure is under relaxed by computing its value using $p = p^* + \lambda^p p'$, where λ^p is the pressure under relaxation factor. For optimum convergence, λ^p is usually set equal to $(1-\lambda^{\mathbf{v}})$, where $\lambda^{\mathbf{v}}$ is the momentum under relaxation factor, with more information on this provided later.

Despite the use of under relaxation, the rate of convergence of the SIMPLE algorithm remains problem dependent and researchers sought alternatives for further improvements. Their effort culminated in the development of a SIMPLE-like family of algorithms such as SIMPLEC [17], SIMPLER [3], PISO [18], SIMPLEX [5], PRIME [19], SIMPLEM [20], and SIMPLEST [21]. Moukalled and Darwish [22] unified the formulation of these algorithms for both incompressible and compressible flows while Darwish et al. [23] and Jang et al. [24] assessed their performance. It is not the intention here to give a full account of these algorithms, rather, attention will be focussed on the two most popular variants, which are the SIMPLEC (SIMPLE Consistent) algorithm of Van Doormal and Raithby and the PISO (Pressure-Implicit Split Operator) algorithm of Issa. These two algorithms present two different approaches for dealing with the $\overline{\mathbf{H}_f}[\mathbf{v}']$ term. In SIMPLEC the velocity correction at the main grid point is approximated as the weighted average of the velocity corrections at the neighboring locations altering the term $\overline{\mathbf{H}_f}[\mathbf{v}']$ into a modified one, $\tilde{\mathbf{H}}_f[\mathbf{v}']$, of smaller magnitude, which is then neglected. In the PISO algorithm, the $\overline{\mathbf{H}_f}[\mathbf{v}']$ term is accounted for as part of the split operator approach. In all other algorithms, the $\overline{\mathbf{H}_f}[\mathbf{v}']$ term is neglected as in SIMPLE and modifications are introduced either to the momentum equations or the $\mathbf{D}^{\mathbf{v}}$ operator. Because the PISO algorithm is equivalent to one step of the SIMPLE algorithm and one or more steps of the PRIME algorithm, the latter is also detailed.

In the PRIME [19] algorithm the momentum equation is solved explicitly. This explicit treatment of the momentum equation is justified by its small contribution to the convergence of the entire flow field. On the other hand, finding the correct solution for the pressure field represents the most important factor impacting the overall convergence.

In SIMPLEST [21], the coefficients in the momentum equation are separated into their convection and diffusion parts with the convection terms treated explicitly and the diffusion terms implicitly, thus affecting $\mathbf{D}^{\mathbf{v}}$ and $\mathbf{H}$. The justification for the explicit treatment of convection is based on the similarity between the propagation of disturbances at a finite speed without any change in magnitude in a pure convection situation, and the propagation of error, from a particular point to the neighboring grid points, in a single iteration of explicit iterative methods. While the implicit treatment of diffusion is argued based on the similarity between the propagation of disturbances in a pure diffusion situation instantaneously in all directions with rapid decay in their amplitude and the reduction of errors throughout the entire solution domain, in a single iteration, by implicit solution methods.

In SIMPLEM (SIMPLE-Modified) [20], the pressure-correction equation is solved before the momentum equation implying that the pressure field is computed using the old velocity field. This results in better pressure corrections than velocity corrections and interchange the disadvantages and advantages of the SIMPLE algorithm.

In SIMPLER (SIMPLE-Revised) [3], an additional equation is developed from which the pressure is directly calculated while the SIMPLE-like pressure-correction equation is used to update the velocity field. The reason for a separate pressure equation being that, once the velocity field is updated using the predicted pressure correction field, it no longer satisfies the momentum equations. Therefore, the pressure should be calculated from another equation to match the velocities, so that the momentum equations are also satisfied.

The SIMPLEX algorithm [5] was developed with the aim of ensuring that the rate of convergence will not degrade with grid refinement. It differs from SIMPLE in the way the $\mathbf{D}^{\mathbf{v}}$ field is computed. This is done by using an additional set of equations, which is developed and solved based on the assumption that the influence of the spatial distribution of pressure difference changes little with grid refinement. Therefore, if the pressure difference influence is restricted to a cell, it would be appropriate to assume that, by extrapolation, the pressure difference at the main grid point adequately represents the pressure differences at the element faces.

Though all of the above algorithms were originally derived for a segregated grid, they are applicable within a collocated grid framework.

15.7.1 The SIMPLEC Algorithm

The SIMPLEC (SIMPLE-Consistent) [17] algorithm is a modified version of the SIMPLE algorithm derived by simply assuming that the velocity correction at point C is the weighted average of the corrections at the neighboring grid points. Mathematically this is expressed by

$$\mathbf{v}'_C \approx \frac{\sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F}{\sum_{F\sim NB(C)} a_F^{\mathbf{v}}} \Rightarrow \sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F \approx \mathbf{v}'_C \sum_{F\sim NB(C)} a_F^{\mathbf{v}} \tag{15.170}$$

and using the $\mathbf{H}$ operator, Eq. (15.170) can be written as

$$\sum_{F\sim NB(C)} \frac{a_F^{\mathbf{v}} \mathbf{v}'_F}{a_C^{\mathbf{v}}} \approx \mathbf{v}'_C \sum_{F\sim NB(C)} \frac{a_F^{\mathbf{v}}}{a_C^{\mathbf{v}}} \Rightarrow \mathbf{H}_C[\mathbf{v}'] \approx \mathbf{v}'_C \mathbf{H}_C[1] \tag{15.171}$$

Therefore instead of neglecting the $\overline{\mathbf{H}_C}[\mathbf{v}']$ term as in SIMPLE, it is replaced by the approximate value given by the above equation. With this approximation the velocity correction given by Eq. (15.85) becomes

$$(1 + \mathbf{H}_C[1])\mathbf{v}'_C = -\mathbf{D}^{\mathbf{v}}_C(\nabla p')_C \Rightarrow \mathbf{v}'_C = -\tilde{\mathbf{D}}^{\mathbf{v}}_C(\nabla p')_C \tag{15.172}$$

Equation (15.172) can then be used to derive the pressure correction equation.

The same result can be achieved by adding and subtracting the term $\sum_{F \sim NB(C)} a^{\mathbf{v}}_F \mathbf{v}_C$ from the momentum equation obtained by combining Eqs. (15.76) and (15.77), leading to the following modified equation:

$$\left(a^{\mathbf{v}}_C + \sum_{F \sim NB(C)} a^{\mathbf{v}}_F\right)\mathbf{v}_C + \sum_{F \sim NB(C)} a^{\mathbf{v}}_F(\mathbf{v}_F - \mathbf{v}_C) = -V_C(\nabla p)_C + \hat{\mathbf{b}}^{\mathbf{v}}_C \tag{15.173}$$

which, in turn, can be written as

$$\mathbf{v}_C + \tilde{\mathbf{H}}_C[\mathbf{v} - \mathbf{v}_C] = -\tilde{\mathbf{D}}^{\mathbf{v}}_C(\nabla p)_C + \tilde{\mathbf{B}}^{\mathbf{v}}_C \tag{15.174}$$

By using Eq. (15.174), the velocity correction equation becomes

$$\mathbf{v}'_C = -\tilde{\mathbf{H}}_C[\mathbf{v}' - \mathbf{v}'_C] - \tilde{\mathbf{D}}^{\mathbf{v}}_C(\nabla p')_C \tag{15.175}$$

Then the term $\tilde{\mathbf{H}}_C[\mathbf{v}' - \mathbf{v}'_C]$ is dropped, which is equivalent to the approximation given by Eq. (15.171), and the modified velocity correction is used in deriving the pressure correction equation.

Due to a better estimate in SIMPLEC (i.e., a smaller term is dropped), the relaxation of pressure becomes unnecessary and as compared to SIMPLE, the resulting velocity corrections will satisfy better the momentum equations. Consequently, a higher rate of convergence is obtained. Thus, with the exceptions of dropping $\tilde{\mathbf{H}}_C[\mathbf{v}' - \mathbf{v}'_C]$ rather than $\mathbf{H}_C[\mathbf{v}']$ and replacing $\mathbf{D}^{\mathbf{v}}_C$ by $\tilde{\mathbf{D}}^{\mathbf{v}}_C$, the steps involved in the SIMPLEC algorithm are similar to those of the SIMPLE algorithm.

15.7.2 The PRIME Algorithm

In the PRIME (PRessure Implicit Momentum Explicit) [19] algorithm, the momentum equation is solved explicitly. This explicit treatment is justified by the small contribution to the convergence of the entire flow field by the iterative sweeps of the momentum equation. On the other hand, finding the correct solution for the pressure field represents the most important factor in the overall convergence. Based on this argument, the PRIME algorithm can be summarized as follows:

The momentum equation is solved explicitly to obtain a new velocity field $\mathbf{v}^*$ using

$$\mathbf{v}_C^* = -\mathbf{H}_C\left[\mathbf{v}^{(n)}\right] - \mathbf{D}_C^{\mathbf{v}}\left(\nabla p^{(n)}\right)_C + \mathbf{B}_C^{\mathbf{v}} \tag{15.176}$$

This velocity field is employed to derive the pressure correction equation. Thus defining the correction fields such that

$$\mathbf{v}_C^{**} = \mathbf{v}_C^* + \mathbf{v}'_C \quad p_C^* = p_C^{(n)} + p'_C \tag{15.177}$$

the corrected field would satisfy

$$\mathbf{v}_C^{**} = -\mathbf{H}_C[\mathbf{v}^{**}] - \mathbf{D}_C^{\mathbf{v}}(\nabla p^*)_C + \mathbf{B}_C^{\mathbf{v}} = -\mathbf{H}_C[\mathbf{v}^* + \mathbf{v}'] - \mathbf{D}_C^{\mathbf{v}}\left[\nabla\left(p^{(n)} + p'\right)\right]_C + \mathbf{B}_C^{\mathbf{v}} \tag{15.178}$$

leading to the following expression relating velocity and pressure correction:

$$\mathbf{v}'_C = \underline{-\left(\mathbf{H}_C\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right] + \mathbf{H}_C[\mathbf{v}']\right)} - \mathbf{D}_C^{\mathbf{v}}\nabla p'_C \tag{15.179}$$

Substituting Eq. (15.179) and its correction into the continuity equation yields

$$-\sum_{f\sim nb(C)} \rho_f \overline{\mathbf{D}_f}\nabla p'_f \cdot \mathbf{S}_f = -\sum_{f\sim nb(C)} \dot{m}_f^* + \underline{\sum_{f\sim nb(C)} \left[\rho_f\left(\overline{\mathbf{H}_f}\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right] + \overline{\mathbf{H}_f}[\mathbf{v}']\right)\cdot \mathbf{S}_f\right]} \tag{15.180}$$

where the underlined terms in Eqs. (15.179) and (15.180) are neglected.

The terms neglected in PRIME $\left(\mathbf{H}_C\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right] + \mathbf{H}_C[\mathbf{v}']\right)$ can become smaller than the term neglected in SIMPLE $(\mathbf{H}_C[\mathbf{v}'])$ if $\mathbf{H}_C[\mathbf{v}']$ and $\mathbf{H}_C\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right]$ are of opposite signs. It is worth noting that $\mathbf{H}_C[\mathbf{v}'] = \mathbf{H}_C[\mathbf{v}^{**} - \mathbf{v}^*]$ is a correction to satisfy continuity, while $\mathbf{H}_C\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right]$ is a correction to satisfy momentum. Usually the corrector added to satisfy momentum is opposite to that added to satisfy continuity and hence, the neglected term $\left(\mathbf{H}_C\left[\mathbf{v}^* - \mathbf{v}^{(n)}\right] + \mathbf{H}_C[\mathbf{v}']\right)$ is smaller. Moreover, since the momentum equations are explicitly solved, no under-relaxation is required. This has the advantage of increasing the stability of the algorithm.

15.7.3 The PISO Algorithm

In the PISO algorithm [18, 25], the $\mathbf{H}_C[\mathbf{v}']$ term is accounted for as part of a correction procedure composed of two or more steps. The first step is similar to the

SIMPLE algorithm where $\mathbf{v}'$ is computed from Eq. (15.83) while neglecting $\mathbf{H}_C[\mathbf{v}']$. The continuity satisfying velocity $\mathbf{v}^{**}$ and pressure p^* fields are used to recalculate the coefficients of the momentum equation and then to solve it explicitly. The new velocity field $\mathbf{v}^{***}$ is used to calculate the mass flow rate field $\dot{m}^{***}$ at the element faces using the Rhie-Chow interpolation. Then, $\mathbf{H}_C[\mathbf{v}']$ is partially recovered in a second corrector step where the velocity correction is written as

$$\begin{aligned}
\mathbf{v}_C^{****} &= \mathbf{v}_C^{***} + \mathbf{v}''_C \\
&= -\mathbf{H}_C^{**}[\mathbf{v}^{**}] - (\mathbf{D}_C^{\mathbf{v}})^{**}(\nabla p^*)_C + \mathbf{v}''_C \\
&= -\mathbf{H}_C^{**}[\mathbf{v}^* + \mathbf{v}'] - (\mathbf{D}_C^{\mathbf{v}})^{**}(\nabla p^*)_C + \mathbf{v}''_C \\
&= -\mathbf{H}_C^{**}[\mathbf{v}^*] - \mathbf{H}_C^{**}[\mathbf{v}'] - (\mathbf{D}_C^{\mathbf{v}})^{**}(\nabla p^*)_C + \mathbf{v}''_C \\
&= \underbrace{-\mathbf{H}_C^{**}[\mathbf{v}^*] - (\mathbf{D}_C^{\mathbf{v}})^{**}(\nabla p^*)_C}_{\approx \mathbf{v}_C^{**}} - \mathbf{H}_C^{**}\left[-\mathbf{D}_C^{\mathbf{v}}(\nabla p')_C\right] + \mathbf{v}''_C \\
&\approx \mathbf{v}_C^{**} + \underline{\mathbf{v}''_C - \mathbf{H}_C^{**}\left[\mathbf{D}_C^{\mathbf{v}}(\nabla p')_C\right]}
\end{aligned} \tag{15.181}$$

In Eq. (15.181) the underlined term represents the portion of the $\mathbf{H}_C[\mathbf{v}']$ that is recovered by the second corrector step. The second velocity correction satisfies

$$\mathbf{v}''_C = -\underline{\mathbf{H}_C^{**}[\mathbf{v}'']} - (\mathbf{D}_C^{\mathbf{v}})^{**}(\nabla p'')_C \tag{15.182}$$

Using Eq. (15.181) with the Rhie-Chow interpolation between points C and F, a new pressure-correction field is obtained as

$$-\sum_{f\sim nb(C)} \rho_f \overline{\mathbf{D}_f} \nabla p''_f \cdot \mathbf{S}_f = -\sum_{f\sim nb(C)} \dot{m}_f^* + \underline{\sum_{f\sim nb(C)} \left(\rho_f \overline{\mathbf{H}_f}[\mathbf{v}''] \cdot \mathbf{S}_f\right)} \tag{15.183}$$

where the underlined terms in Eqs. (15.182) and (15.183) are again neglected. This corrector step may be repeated as many times as desired, each time recovering a new additional portion of $\mathbf{H}_C[\mathbf{v}']$.

By following the sequence of events, it can be easily seen that PISO may be considered as a combination of one SIMPLE step and one or more PRIME steps, hence combining the implicitness of the SIMPLE algorithm with the stability of the PRIME algorithm. The sequence of events in the collocated PISO algorithm can be summarized as follows:

1. To compute the solution at time $t + \Delta t$, use as an initial guess the solution at time t for pressure, velocity, and mass flow rate fields $p^{(n)}$, $u^{(n)}$, and $\dot{m}^{(n)}$, respectively.

SIMPLE Step

2. Solve implicitly the momentum equation given by Eq. (15.70) to obtain a new velocity field $\mathbf{v}^*$.
3. Update the mass flow rate at the cell faces using the Rhie-Chow interpolation technique (Eq. 15.100) to obtain a momentum satisfying mass flow rate field $\dot{m}_f^*$.

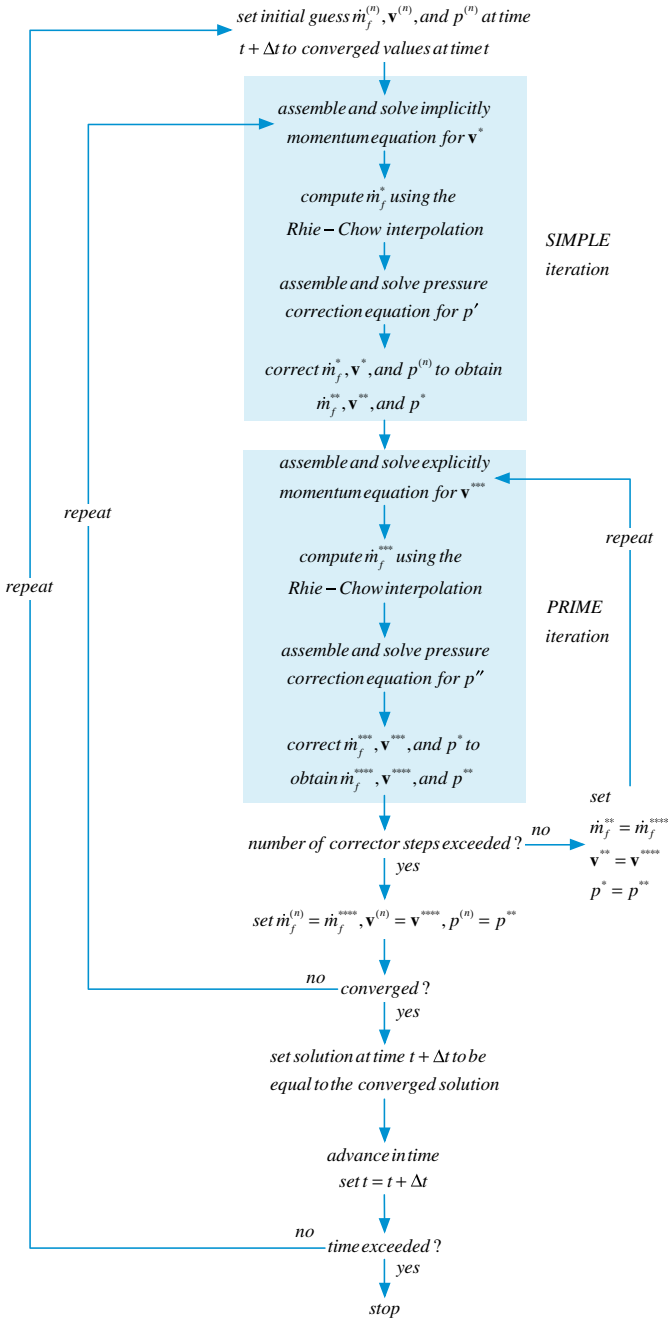


Fig. 15.27 A flow chart of the PISO algorithm

4. Using the new mass flow rates, assemble the pressure correction equation (Eq. 15.98) and solve it to obtain a pressure correction field p'.
5. Update the pressure and velocity fields at the cell centroids and the mass flow rate at the cell faces to obtain continuity-satisfying fields using Eq. (15.101).

PRIME Step(s)

6. Using the latest available velocity and pressure fields, calculate the coefficients of the momentum equation and solve it explicitly.
7. Update the mass flow rate at the cell faces using the Rhie-Chow interpolation technique.
8. Using the new mass flow rates, assemble the pressure correction equation (Eq. 15.183) and solve it to obtain a pressure correction field.
9. Update pressure, velocity, and mass flow rate fields using expressions similar to the ones given in Eq. (15.101).
10. Go to step 6 and repeat based on the desired number of corrector steps.
11. Set the initial guess for velocity, mass flow rate, and pressure as u^{**}, $\dot{m}^{**}$, and p^{*}.
12. Go back to step 2 and repeat until convergence.
13. Set the solution at time $t + \Delta t$ to be equal to the converged solution and set the current time $t + \Delta t$ to be t.
14. Advance to the next time step.
15. Go back to step 1 and repeat until the last time step is reached.

A flowchart of the PISO algorithm is presented in Fig. 15.27.

15.8 Optimum Under-Relaxation Factor Values for v and p'

To promote convergence in the SIMPLE algorithm the momentum and continuity equations are under relaxed using the under relaxation factors $\lambda^{\mathbf{v}}$ and λ^{p}, respectively. An important task is to find the under relaxation values that will result in the optimum convergence rate. Recalling that the velocity correction is obtained without any under relaxation from

$$\mathbf{v}'_C = -\mathbf{D}_C(\nabla p')_C \tag{15.184}$$

Moreover, in calculating the pressure field, the pressure correction is under relaxed in order for the velocity correction field given by Eq. (15.184) to satisfy the exact velocity correction equation given by

$$\mathbf{v}'_C = -\mathbf{H}_C[\mathbf{v}'] - \lambda^{p}\mathbf{D}_C(\nabla p')_C \tag{15.185}$$

Equating Eqs. (15.184) and (15.185), an expression for λ^p is obtained as

$$-\mathbf{D}_C(\nabla p')_C = -\mathbf{H}_C[\mathbf{v}'] - \lambda^p \mathbf{D}_C(\nabla p')_C \Rightarrow \lambda^p = 1 + \frac{\mathbf{H}_C[\mathbf{v}']}{\mathbf{v}'_C} = 1 + \frac{\sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F}{a_C^{\mathbf{v}} \mathbf{v}'_C} \tag{15.186}$$

The SIMPLEC algorithm eliminated the need to under relax pressure correction and resulted in the optimum acceleration rate. Therefore, using the approximation introduced in SIMPLEC, the velocity correction at C can be written as the weighted average of the velocity corrections at the neighboring grid points such that

$$\mathbf{v}'_C \approx \frac{\sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F}{\sum_{F\sim NB(C)} a_F^{\mathbf{v}}} \tag{15.187}$$

From Eqs. (15.70)–(15.73) the coefficient $a_C^{\mathbf{v}}$ can be expressed as

$$a_C^{\mathbf{v}} = \frac{1}{\lambda^{\mathbf{v}}} \left(a_C^{\mathbf{v}} - \sum_{F\sim NB(C)} a_F^{\mathbf{v}} + \sum_{f\sim nb(C)} \dot{m}_f \right) \tag{15.188}$$

which in the limit of a steady state solution (the case for which under relaxation is used, since for an unsteady situation the time step plays the role of the under relaxation factor) reduces to

$$a_C^{\mathbf{v}} = -\frac{1}{\lambda^{\mathbf{v}}} \sum_{F\sim NB(C)} a_F^{\mathbf{v}} \tag{15.189}$$

Substituting Eq. (15.189) in Eq. (15.187), the velocity correction is approximated as

$$\mathbf{v}'_C \approx -\frac{\sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F}{\lambda^{\mathbf{v}} a_C^{\mathbf{v}}} \Rightarrow a_C^{\mathbf{v}} \mathbf{v}'_C \approx -\frac{\sum_{F\sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}'_F}{\lambda^{\mathbf{v}}} \tag{15.190}$$

substituting Eq. (15.190) in Eq. (15.186), an expression relating $\lambda^{\mathbf{v}}$ and λ^p is obtained as

$$\lambda^p \approx 1 - \lambda^{\mathbf{v}} \tag{15.191}$$

Experience has shown that the performance of the SIMPLE algorithm with its under relaxation factors satisfying Eq. (15.191) is similar to that of the SIMPLEC algorithm.

15.9 Treatment of Various Terms with the Rhie-Chow Interpolation

15.9.1 Treatment of the Under-Relaxation Term

The use of the collocated grid method with the Rhie-Chow interpolation resulted in solutions that are dependent on the value of under relaxation factor in the momentum equation. To eliminate this dependence, a modification to the Rhie-Chow interpolation is required. The under relaxed momentum equation is written as

$$\frac{1}{\lambda^{\mathbf{v}}} a_C^{\mathbf{v}} \mathbf{v}_C = -\sum_{F \sim NB(C)} a_F^{\mathbf{v}} \mathbf{v}_F + \mathbf{b}_C^{\mathbf{v}} - V_C \nabla p_C + \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}}\right) a_C^{\mathbf{v}} \mathbf{v}_C^{(n)} \tag{15.192}$$

where $\mathbf{b}_C^{\mathbf{v}}$ is the source term of the momentum equation from which the pressure and under relaxation source terms are extracted and $\mathbf{v}_C^{(n)}$ is the previous iteration value of velocity at cell centroid C. The corresponding under relaxed momentum equation using a staggered grid formulation can be expressed as

$$\frac{1}{\lambda^{\mathbf{v}}} a_f^{\mathbf{v}} \mathbf{v}_f = -\sum_{nb \sim NB(f)} a_{nb}^{\mathbf{v}} \mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}} - V_f \nabla p_f + \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}}\right) a_f^{\mathbf{v}} \mathbf{v}_f^{(n)} \tag{15.193}$$

The Rhie-Chow interpolation method mimics the staggered grid formulation by forming a pseudo-momentum equation at the cell face. It is because of this behavioral imitation that the Rhie-Chow interpolation is successful. Therefore as a guiding principle, the yardstick to any modification to the Rhie-Chow interpolation should be whether the modified formulation is similar to the staggered grid formulation. Therefore, the form of the under relaxed equation using the Rhie-Chow interpolation should be given by

$$\frac{1}{\lambda^{\mathbf{v}}} \overline{a_f^{\mathbf{v}}} \mathbf{v}_f = -\overline{\sum_{nb \sim NB(f)} a_{nb}^{\mathbf{v}} \mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} - \overline{V_f} \nabla p_f + \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}}\right) \overline{a_f^{\mathbf{v}}} \mathbf{v}_f^{(n)} \tag{15.194}$$

The average of the first term on the right hand side is obtained as

$$\begin{aligned}
-\overline{\sum_{nb \sim NB(f)} a_{nb}^{\mathbf{v}} \mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} &= -g_C \left(\sum_{F \sim NB(C)} \left(a_F^{\mathbf{v}} \mathbf{v}_F \right) + \mathbf{b}_C^{\mathbf{v}} \right) - g_F \left(\sum_{N \sim NB(F)} \left(a_N^{\mathbf{v}} \mathbf{v}_N \right) + \mathbf{b}_F^{\mathbf{v}} \right) \\
&= g_C \left[\frac{1}{\lambda^{\mathbf{v}}} a_C^{\mathbf{v}} \mathbf{v}_C + V_C \nabla p_C - \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}} \right) a_C^{\mathbf{v}} \mathbf{v}_C^{(n)} \right] \\
&\quad + g_F \left[\frac{1}{\lambda^{\mathbf{v}}} a_F^{\mathbf{v}} \mathbf{v}_F + V_F \nabla p_F - \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}} \right) a_F^{\mathbf{v}} \mathbf{v}_F^{(n)} \right] \\
&= \frac{1}{\lambda^{\mathbf{v}}} \overline{a_f^{\mathbf{v}} \mathbf{v}_f} + \overline{V_f \nabla p_f} - \left(\frac{1-\lambda^{\mathbf{v}}}{\lambda^{\mathbf{v}}} \right) \overline{a_f^{\mathbf{v}} \mathbf{v}_f^{(n)}}
\end{aligned} \tag{15.195}$$

Substituting Eq. (15.195) into Eq. (15.194), the extended Rhie-Chow interpolated cell face velocity $\mathbf{v}_f$ is obtained as

$$\mathbf{v}_f = \overline{\mathbf{v}_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right) + (1-\lambda^{\mathbf{v}})\left(\mathbf{v}_f^{(n)} - \overline{\mathbf{v}_f^{(n)}}\right) \tag{15.196}$$

Not accounting for the effect of under-relaxation on the face velocity results in solutions that depend on the under relaxation factor.

15.9.2 Treatment of the Transient Term

When solving a transient problem with a backward Euler transient scheme the discretized momentum equation can be written as

$$a_C^{\mathbf{v}}\mathbf{v}_C = -\sum_{F\sim NB(C)}\left(a_F^{\mathbf{v}}\mathbf{v}_F\right) + \mathbf{b}_C^{\mathbf{v}} - V_C\nabla p_C + a_C^{\circ}\mathbf{v}_C^{\circ} \tag{15.197}$$

where $\mathbf{b}_C^{\mathbf{v}}$ is the source term of the momentum equation from which the pressure and transient source terms are extracted. The equivalent equation for the staggered grid variable arrangement has a similar form given by

$$a_f^{\mathbf{v}}\mathbf{v}_f = -\sum_{nb\sim NB(f)}\left(a_{nb}^{\mathbf{v}}\mathbf{v}_{nb}\right) + \mathbf{b}_f^{\mathbf{v}} - V_f\nabla p_f + a_f^{\circ}\mathbf{v}_f^{\circ} \tag{15.198}$$

Using the Rhie-Chow interpolation method, a pseudo cell-face equation will be constructed as

$$\overline{a_f^{\mathbf{v}}}\mathbf{v}_f = -\overline{\sum_{nb\sim NB(f)} a_{nb}^{\mathbf{v}}\mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} - \overline{V_f}\nabla p_f + \overline{a_f^{\circ}}\mathbf{v}_f^{\circ} \tag{15.199}$$

The average of the first term on the right hand side is obtained as

$$\begin{aligned}
-\overline{\sum_{nb\sim NB(f)} a_{nb}^{\mathbf{v}}\mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} &= -g_C\left(\sum_{F\sim NB(C)}\left(a_F^{\mathbf{v}}\mathbf{v}_F\right) + \mathbf{b}_C^{\mathbf{v}}\right) \\
&\quad - g_F\left(\sum_{N\sim NB(F)}\left(a_N^{\mathbf{v}}\mathbf{v}_N\right) + \mathbf{b}_F^{\mathbf{v}}\right) \\
&= g_C\left[a_C^{\mathbf{v}}\mathbf{v}_C + V_C\nabla p_C - a_C^{\circ}\mathbf{v}_C^{\circ}\right] \\
&\quad + g_F\left[a_F^{\mathbf{v}}\mathbf{v}_F + V_F\nabla p_F - a_F^{\circ}\mathbf{v}_F^{\circ}\right] \\
&= \overline{a_f^{\mathbf{v}}\mathbf{v}_f} + \overline{V_f\nabla p_f} - \overline{a_f^{\circ}\mathbf{v}_f^{\circ}}
\end{aligned} \tag{15.200}$$

Substituting into Eq. (15.84), the extended Rhie-Chow interpolated cell face velocity $\mathbf{v}_f$ is obtained

$$\mathbf{v}_f = \overline{\mathbf{v}_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right) + \frac{\overline{a_f^{\circ}\mathbf{D}_f^{\mathbf{v}}}}{V_f}\left(\mathbf{v}_f^{\circ} - \overline{\mathbf{v}_f^{\circ}}\right) \tag{15.201}$$

Not accounting for the effect of the unsteady term on the face velocity results in solutions that are time step dependent and have an oscillatory behavior for small time step. This correction is valid only for the first order Euler discretization. In case of more accurate time discretization schemes similar corrections can be performed following the same principles.

15.9.3 Treatment of the Body Force Term

When treating body forces in the staggered grid arrangement, the stencil of the body force term is exactly that of the pressure gradient term. In the case of a collocated grid arrangement, the body force, velocity, and momentum variables are calculated at the same location. Thus, in order to have a discretization of the body force that retains a similar stencil as the pressure, a redistribution of the body force term is needed. The discretized momentum equation is written as

$$a_C^{\mathbf{v}}\mathbf{v}_C = -\sum_{F\sim NB(C)} a_F^{\mathbf{v}}\mathbf{v}_F + \mathbf{b}_C^{\mathbf{v}} - V_C(\nabla p)_C + V_C\overline{\overline{\mathbf{B}_C^{\mathbf{v}}}} \tag{15.202}$$

where the double bar indicates two averaging steps. The first step is to compute $\overline{\mathbf{B}_C^{\mathbf{v}}}$ (Fig. 15.28) at the cell face as

$$\overline{\mathbf{B}_f^{\mathbf{v}}} = g_C\mathbf{B}_C^{\mathbf{v}} + (1 - g_C)\mathbf{B}_F^{\mathbf{v}} \tag{15.203}$$

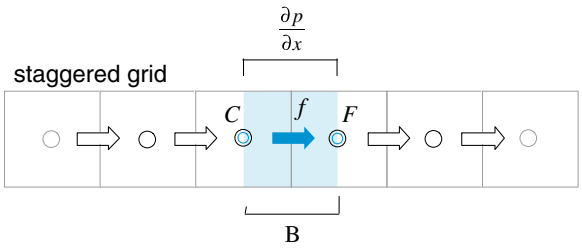


Fig. 15.28 Treatment of body force and pressure gradient on a staggered grid

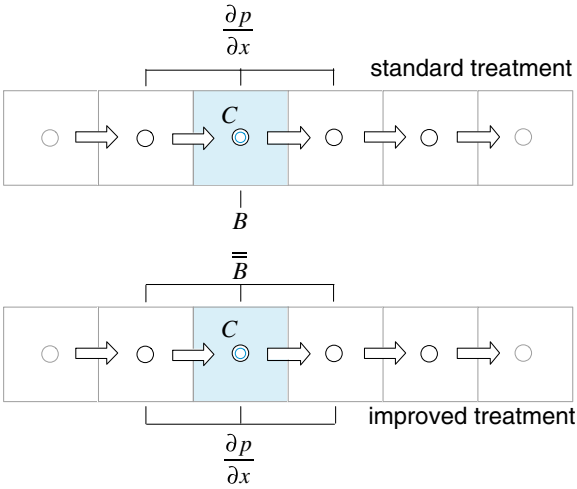


Fig. 15.29 Standard and improved Rhie-Chow treatment of body forces

while the second (Fig. 15.29) is to get an average of these face values at the cell centre.

The average values at cell center can best be derived [26] by considering the one dimensional situation depicted in Fig. 15.30.

For the case of a stationary fluid, the pressure gradient should be in equilibrium with the body forces leading to

$$0 = -\nabla p_f + \mathbf{B}_f^{\mathbf{v}} \tag{15.204}$$

Expanding the above equation, a relation between the pressures at C and F can be written as

$$p_C = p_F + B_f \delta y \tag{15.205}$$

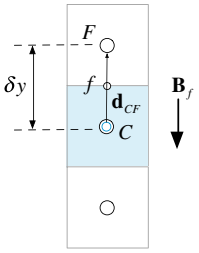


Fig. 15.30 One dimensional stationary fluid

or more generally as

$$p_C = p_F + \mathbf{B}_f \cdot \mathbf{d}_{CF} \tag{15.206}$$

where B_f is the magnitude of $\mathbf{B}_f$ given by

$$B_f = \rho_f g \tag{15.207}$$

For incompressible flows, the variation with temperature of the density appearing in the body force term is modeled using the Boussinesq approximation as given by Eq. (3.101).

Again for cell C the pressure gradient should be in equilibrium with the body forces, resulting in

$$0 = -\nabla p_C + \mathbf{B}_C^{\mathbf{v}} \Rightarrow \nabla p_C = \mathbf{B}_C^{\mathbf{v}} \tag{15.208}$$

However the pressure gradient for cell C is computed as

$$\begin{aligned} \nabla p_C &= \frac{\sum_f p_f \mathbf{S}_f}{V_C} \\ &= \frac{\sum_f (g_C p_C + (1 - g_C) p_F) \mathbf{S}_f}{V_C} \end{aligned} \tag{15.209}$$

substituting from Eq. (15.206) gives

$$\begin{aligned} \nabla p_C &= \frac{\sum_f \left(g_C \left(p_F + \overline{\mathbf{B}_f^{\mathbf{v}}} \cdot \mathbf{d}_{CF} \right) + (1 - g_C) p_F \right) \mathbf{S}_f}{V_C} \\ &= \frac{\sum_f p_F \mathbf{S}_f}{V_C} + \frac{\sum_f g_C \left(\overline{\mathbf{B}_f^{\mathbf{v}}} \cdot \mathbf{d}_{CF} \right) \mathbf{S}_f}{V_C} \\ &= p_F \underbrace{\frac{\sum_f \mathbf{S}_f}{V_C}}_{0} + \frac{\sum_f g_C \left(\overline{\mathbf{B}_f^{\mathbf{v}}} \cdot \mathbf{d}_f \right) \mathbf{S}_f}{V_C} \\ &= \frac{\sum_f g_C \left(\overline{\mathbf{B}_f^{\mathbf{v}}} \cdot \mathbf{d}_f \right) \mathbf{S}_f}{V_C} \\ &= \overline{\overline{\mathbf{B}_C^{\mathbf{v}}}} \end{aligned} \tag{15.210}$$

which implies that

$$\overline{\overline{\mathbf{B}_C^{\mathbf{v}}}} = \frac{\sum_f g_C \left(\overline{\mathbf{B}_f^{\mathbf{v}}} \cdot \mathbf{d}_f \right) \mathbf{S}_f}{V_C} \tag{15.211}$$

The second requirement is that the cell-face velocity be similar to that of the staggered arrangement equation:

$$\overline{a_f^{\mathbf{v}}}\mathbf{v}_f = \overline{-\sum_{nb\sim NB(f)} a_{nb}^{\mathbf{v}}\mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} - \overline{V_f}\nabla p_f + \overline{V_f\mathbf{B}_f^{\mathbf{v}}} \tag{15.212}$$

where $\mathbf{b}_C^{\mathbf{v}}$ is the source term given in Eq. (15.71) from which the pressure and body force terms are extracted. The averaging of the coefficients yields

$$\begin{aligned}
-\overline{\sum_{nb\sim NB(f)} a_{nb}^{\mathbf{v}}\mathbf{v}_{nb} + \mathbf{b}_f^{\mathbf{v}}} &= -g_C\left(\sum_{F\sim NB(C)}\left(a_F^{\mathbf{v}}\mathbf{v}_F\right) + \mathbf{b}_C^{\mathbf{v}}\right) \\
&\quad - g_F\left(\sum_{N\sim NB(F)}\left(a_N^{\mathbf{v}}\mathbf{v}_N\right) + \mathbf{b}_F^{\mathbf{v}}\right) \\
&= g_C\left[a_C^{\mathbf{v}}\mathbf{v}_C + V_C\nabla p_C - V_C\overline{\overline{\mathbf{B}_C^{\mathbf{v}}}}\right] \\
&\quad + g_F\left[a_F^{\mathbf{v}}\mathbf{v}_F + V_F\nabla p_F - V_F\overline{\overline{\mathbf{B}_F^{\mathbf{v}}}}\right] \\
&= \overline{a_f^{\mathbf{v}}\mathbf{v}_f} + \overline{V_f\nabla p_f} - \overline{V_f\overline{\overline{\mathbf{B}_f^{\mathbf{v}}}}}
\end{aligned} \tag{15.213}$$

and substituting into Eq. (15.179), the extended Rhie-Chow interpolated cell face velocity $\mathbf{v}_f$ is obtained as

$$\mathbf{v}_f = \overline{\mathbf{v}_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right) + \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\overline{\mathbf{B}_f^{\mathbf{v}}} - \overline{\overline{\mathbf{B}_f^{\mathbf{v}}}}\right) \tag{15.214}$$

where $\overline{\overline{\mathbf{B}_f^{\mathbf{v}}}}$ is calculated as

$$\overline{\overline{\mathbf{B}_f^{\mathbf{v}}}} = g_C\overline{\mathbf{B}_C^{\mathbf{v}}} + (1 - g_C)\overline{\mathbf{B}_F^{\mathbf{v}}} \tag{15.215}$$

The above additional treatment of the cell face velocity increases the overall robustness of the solution procedure for situations where variations in body forces are important (e.g., free-surface flows).

15.9.4 Combined Treatment of Under-Relaxation, Transient, and Body Force Terms

In general all three terms described above should be dealt with together. This necessitates modifying the Rhie-Chow interpolation to account for all three effects. Fortunately this can easily be derived by using the principle of superposition leading to the following interface velocity:

$$\begin{aligned}\mathbf{v}_f = \overline{\mathbf{v}_f} - \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right) + \overline{\mathbf{D}_f^{\mathbf{v}}}\left(\overline{\mathbf{B}_f^{\mathbf{v}}} - \overline{\overline{\mathbf{B}_f^{\mathbf{v}}}}\right) \\ + \frac{\overline{a_f^{\circ}\mathbf{D}_f^{\mathbf{v}}}}{V_f}\left(\mathbf{v}_f^{\circ} - \overline{\mathbf{v}_f^{\circ}}\right) + (1-\lambda^{\mathbf{v}})\left(\mathbf{v}_f^{(n)} - \overline{\mathbf{v}_f^{(n)}}\right)\end{aligned} \tag{15.216}$$

where in calculating $\overline{\mathbf{D}_f^{\mathbf{v}}}$ the under relaxed value of the $a_C^{\mathbf{v}}$ coefficient is used.

15.10 Computational Pointers

15.10.1 uFVM

In uFVM, the pressure correction equation is implemented in one script file denoted by cfdAssembleMdotTerm. Listing 15.1 shows the core of the algorithm whereby the coefficients of the pressure correction equation are assembled by linearizing the fluxes at each of the interior faces. In addition, the mdot field (i.e., the mass flow rate at the faces) is calculated based on Eq. (15.216), which is subdivided into 9 terms (i.e., terms I trough IX) and assembled step by step to produce the cell face velocity. The terms into which the velocity at the face is decomposed are as follows:

term I: the interpolated velocity field $\overline{\mathbf{v}_f}$,
terms II and III: the face and average pressure gradients $-\overline{\mathbf{D}_f^{\mathbf{v}}}\left(\nabla p_f - \overline{\nabla p_f}\right)$,
terms IV and V: the average and redistributed body forces $\overline{\mathbf{D}_f^{\mathbf{v}}}\left(\overline{\mathbf{B}_f^{\mathbf{v}}} - \overline{\overline{\mathbf{B}_f^{\mathbf{v}}}}\right)$,
terms VI and VII: the transient fluxes $\frac{\overline{a_f^{\circ}\mathbf{D}_f^{\mathbf{v}}}}{V_f}\left(\mathbf{v}_f^{\circ} - \overline{\mathbf{v}_f^{\circ}}\right)$, and
terms VIII and IX: the relaxation correction term $(1-\lambda^{V})(\mathbf{v}_f^{(n)} - \overline{\mathbf{v}_f^{(n)}})$.

```
    %
    % assemble term I
    %     density_f [v]_f.Sf
    %
    U_bar_f = (dot(vel_bar_f(:,:)',Sf(:,:)'))';
    local_FLUXVf =local_FLUXVf+ density_f.*U_bar_f;
    %
    % Assemble term II and linearize it
    %      - density_f ([DPVOL]_f.P_grad_f).Sf
    %
    DUSf = [DU1_f.*Sf(:,1),DU2_f.*Sf(:,2),DU3_f.*Sf(:,3)];
    geoDiff =( dot(Sf(:,:)',DUSf') ./ dot(CN(:,:)',Sf(:,:)'))';
    local_FLUXCf1 = local_FLUXCf1 + density_f.*geoDiff;
    local_FLUXCf2 = local_FLUXCf2 - density_f.*geoDiff;
    local_FLUXVf  = local_FLUXVf  - density_f.*dot(p_grad_f',T')';
    %
    %  assemble term III
    %    density_f ([P_grad]_f.([DPVOL]_f.Sf))
    %
    local_FLUXVf = local_FLUXVf +
density_f.*dot(p_grad_bar_f(iFaces,:)',DUSf(iFaces,:)')';
    %
    % assemble terms IV and V
    %    density_f [DBVOL]_f.([B]_f -[[B]]_f).S_f
    %
    local_FLUXVf = local_FLUXVf +
density_f[iFace]*FVVectorDotProduct(FVTensorVectorDotProduct(DB_f,FVVe
ctorSubstract(FVMakeVector(bf1_bar_f[iFace],bf2_bar_f[iFace]),FVMakeVe
ctor(bf1_redistributed_f[iFace],bf2_redistributed_f[iFace]))),S_f) ;
    %
    % assemble terms VI and VII
    %    [Dt]_f (U_Old_f -[v_old]_f.S_f)
    %
    U_bar_old_f = [velx_old_bar_f[iFace] vely_old_bar_f[iFace])] *
S_f'
    local_FLUXVf = local_FLUXVf + DT_f*(mdot_old_f[iFace] -
density_old_f[iFace]*U_bar_old_f);
    %
    % assemble terms VIII and IX
    %     (1-URF)(U_f -[v]_f.S_f)
    %
    local_FLUXVf = local_FLUXVf + (1.0-Mdot_URF)*(mdot_previous_f -
density_f.*U_bar_f);
    %
    % assemble the flow term dot for the face
    %
    local_mdot_f = local_FLUXCf_1*(pressure[iElement1]+ Pref) +
local_FLUXCf_2*(pressure[iElement2]+ Pref) + local_FLUXVf;
    %


    %
    % Assemble in Global Fluxes
    %
```

Listing 15.1 Script used for the calculation of the mass flow rates and coefficients of the pressure correction equation in uFVM

```
    theFluxes.FLUXC1f(iFaces,1) = local_FLUXCf1;
    theFluxes.FLUXC2f(iFaces,1) = local_FLUXCf2;
    theFluxes.FLUXVf(iFaces,1)  = local_FLUXVf;
    %
    theFluxes.FLUXTf(iFaces,1)  = theFluxes.FLUXC1f.*pressureC(iFaces)
+ theFluxes.FLUXC2f.*pressureN(iFaces) + theFluxes.FLUXVf(iFaces);
    %
    mdot_f = theFluxes.FLUXTf(iFaces);
```

Listing 15.1 (continued)

15.10.2 OpenFOAM®

The numerical techniques introduced so far are used in what follows to develop OpenFOAM® [27] applications for solving the incompressible Navier-Stokes equations.

15.10.2.1 Pressure Correction SIMPLE Solvers

Based on the SIMPLE algorithm, a number of solvers will be constructed. The base solver, simpleFoam, will be presented first. This is followed by a number of versions, with each one adding more capabilities to the base code. These solvers can be summarized as follows:

1. simpleFoam (not the OpenFOAM® built-in solver) is the base code that incorporates the SIMPLE Algorithm in its most basic form.
2. simpleFoamImproved extends the base code to allow for improved treatment of relaxation.
3. simpleFoamTransient adds transient capabilities to the steady-state simpleFoam.
4. simpleFoamBuoyancy adds to the code the body force treatment.

More versions will be covered in the chapters to follow, each one with extended capabilities, added by modifying the base code described in this chapter. A list of the versions that will be covered in the next chapters is given below.

5. simpleFoamCompressible is the compressible version of simpleFoam (Chap. 16)
6. simpleFoamTurbulent includes capabilities for treating turbulent flows (Chap. 17).

simpleFoam

Before reviewing the simpleFoam code, some basic notational issues are addressed. The first step is to define, as shown in Listing 15.2, the geometric fields and parameters that will be initialized and used in the code.

```
#include "fvCFD.H"
#include "orthogonalSnGrad.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
```

Listing 15.2 The #include macro derivatives used to define the types of objects needed

In Listing 15.2, the #include macro directives outside the *main* function are needed to define the types of objects that are then declared and used in the application. The #include "*fvCFD.H*" contains a list of definitions for classes that are in general necessary to build any application in OpenFOAM®. In the developed application an additional header, not present in the *fvCFD.H* header, necessary for the SIMPLE solver implementation will be added.

The use of the #include statements inside the *main* function is a compacting procedure, with each declared statement representing a piece of the code moved to the corresponding file name. For example, the statement #include "createMesh.H" just represents the code shown in Listing 15.3, which is necessary to instantiate the mesh class.

```
createMesh.H
~~~~~~~~~~~~
Foam::Info
    << "Create mesh for time = "
    << runTime.timeName() << Foam::nl << Foam::endl;


Foam::fvMesh mesh
(
    Foam::IOobject
    (
        Foam::fvMesh::defaultRegion,
        runTime.timeName(),
        runTime,
        Foam::IOobject::MUST_READ
    )
);
```

Listing 15.3 The code representing the #include createMesh.H file necessary to instantiate the mesh class

Once the necessary initialization has been performed, the next step is the definition of the proper fields or variables needed by the solver. These are defined in file "*createFields.H*". The first defined field, shown in Listing 15.4, is the pressure field (p).

```
Info << "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Listing 15.4 Script used to define the pressure field

Since the solution is obtained by solving a pressure correction equation instead of a pressure equation, a pressure correction field (pp) is also defined (Listing 15.5).

```
const volScalarField::GeometricBoundaryField& pbf=p.boundaryField();
wordList pbt = pbf.types();
volScalarField pp
(
    IOobject
    (
        "pp",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", p.dimensions(), 0.0),
    pbt
);
```

Listing 15.5 Script used to define the pressure correction field

It is worth noting that in Listing 15.5 a different constructor is used to define the pressure correction field. Since the pressure corrector represents the pressure itself, the same boundary conditions defined for the real pressure can be used for the pressure correction field without the need to define the same quantity twice.

The list of pressure boundary types displayed in Listing 15.6 are now copied under the pbt variable in Listing 15.5 and directly used in the constructor of pp.

```
// Set pp boundary values
forAll(pp.boundaryField(), patchi)
{
      if (isType<fixedValueFvPatchScalarField>(pp.boundaryField()
[patchi]))
    {
       fixedValueFvPatchScalarField& ppbound =

refCast<fixedValueFvPatchScalarField>(pp.boundaryField()[patchi]);

       ppbound == scalarField(ppbound.size(),0.0);
    }
}
```

Listing 15.6 Script showing the declaration of the different pressure boundary types

The corrector should reset to zero the correction field at every iteration and should also apply a zero value at all boundaries for which a Dirichlet boundary condition is used for the pressure.

The velocity and corresponding mass flux fields must also to be defined. As depicted in Listing 15.7, the velocity field is defined through an input file while the mass flux field can be defined as a derived quantity.

```
Info << "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
surfaceScalarField mDot
(
    IOobject
    (
        "mDot",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(U) & mesh.Sf()
);
```

Listing 15.7 Script used to define the velocity and mass flux fields

Finally the fluid thermo-physical properties should also be defined. For incompressible laminar flows this involves simply assigning a value to the kinematic viscosity nu, as shown in Listing 15.8.

```
Info<< "Reading transportProperties\n" << endl;
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);
dimensionedScalar nu
(
    transportProperties.lookup("nu")
);
```

Listing 15.8 Code used to define the fluid thermo-physical properties

After defining all variables the implementation of the SIMPLE algorithm can proceed. A *while* loop can be used for the cases when the stopping criterion is the number of SIMPLE iterations. For each single loop, the momentum and pressure correction equations are solved and updates of the variables are performed. Starting with the momentum equation written as (the form solved in OpenFOAM®),

$$\nabla \cdot \{\mathbf{vv}\} = \nu \nabla^2 \mathbf{v} - \nabla p \tag{15.217}$$

where the kinematic viscosity ν is defined as

$$\nu = \frac{\mu}{\rho} \tag{15.218}$$

and p represents the pressure divided by the density, i.e.,

$$p = \frac{static\,pressure}{\rho} \tag{15.219}$$

its solution is translated into the script shown in Listing 15.9.

```
// Solve the Momentum equation
fvVectorMatrix UEqn
(
        fvm::div(mDot, U)
      - fvm::laplacian(nu, U)
);
UEqn.relax();
solve
(
    UEqn == -fvc::grad(p)
);
```

Listing 15.9 Script to solve the momentum equation

The first instruction in Listing 15.9 defines the finite volume discretization of the momentum equation in vector form with its storage matrix (the three components of the velocity vector are solved in a segregated manner despite its vectorial implementation). The system is then implicitly relaxed and solved via an iterative solver.

Once the momentum equation is solved, a new guess for the velocity field is obtained. This velocity does not satisfy the continuity equation in general and the assembly of the continuity equation in the form of a pressure correction equation is now required to correct the flow field. The pressure correction equation is written as

$$-\nabla \cdot (\bar{\mathbf{D}}_f \nabla p') = -\nabla \cdot (\mathbf{v}) \tag{15.220}$$

where a component of $\bar{\mathbf{D}}$ at an element centroid is computed as

$$D = \frac{V}{a_c^{\mathbf{v}}} \tag{15.221}$$

with values at the faces obtained by interpolation. Its implementation is translated into the following syntax (Listing 15.10):

```
pp = scalar(0.0)*pp;
pp.correctBoundaryConditions();

fvScalarMatrix ppEqn
(
    - fvm::laplacian(DUf, pp, "laplacian(pDiff,pp)")
    + fvc::div(mDot)
);
```

Listing 15.10 Script to implement the pressure correction equation

where div(mDot) is basically $\sum \dot{m}_f$, while pp represents p' and is reset to zero at each iteration. Moreover, the DUf variable is the value of D at the cell face obtained, as shown in Listing 15.11, by linear interpolation using the values at the nodes straddling the face. Further, .A() represents the diagonal terms in the momentum matrix divided by the volume.

```
volScalarField DU = 1.0/UEqn.A();
surfaceScalarField DUf("DUf",linearInterpolate(DU));
```

Listing 15.11 Script to calculate the values of DUf

Listing 15.12 computes the mass flow rate at cell faces (*mDot*) using the Rhie-Chow interpolation where the calculation of ∇p_f and $\overline{\nabla p_f}$ is clearly shown.

```
const surfaceVectorField ed = mesh.delta()()/mag(mesh.delta()());
Foam::fv::orthogonalSnGrad<scalar> faceGradient(mesh);
surfaceVectorField gradp_avg_f = linearInterpolate(fvc::grad(p));
surfaceVectorField gradp_f = gradp_avg_f - (gradp_avg_f & ed)*ed +
(faceGradient.snGrad(p))*ed;

surfaceVectorField U_avg_f = linearInterpolate(U);

// Rhie-Chow interplation
mDot = (U_avg_f & mesh.Sf()) - ( (DUf*( gradp_f - gradp_avg_f)) &
mesh.Sf() );
```

Listing 15.12 Script to compute the mass flow rate at cell faces using the Rhie-Chow interpolation

The pressure correction equation is now fully set and is solved by executing the statement in Listing 15.13.

```
ppEqn.solve();
```

Listing 15.13 Statement used to solve the pressure correction equation

Once the pressure correction equation is solved, the velocity, pressure, and mass flow rate fields are updated using the obtained pressure correction field. Starting with the mass flow rate field (i.e., the *mDot* flux), it is updated by executing the below statement (Listing 15.14).

```
mDot += ppEqn.flux();
```

Listing 15.14 Statement used to update the mass flow rate field to satisfy continuity

The *flux*() function in Listing 15.14 provides the matrix multiplication, the extra diagonal matrix coefficients, and the corresponding solution. Recalling the finite volume discretization of the diffusion term, each extra diagonal of coefficients represents a face of the mesh. Thus the update of the fluxes can be performed in a more consistent way using directly the matrix coefficients and cell values. A simplified version of the *flux*() function is shown in Listing 15.15.

```
for (label face=0; face<lowerAddr.size(); face++)
{
    mDotPrime[face] =
        upperCoeffs[face]*pp[upperAddr[face]]
      - lowerCoeffs[face]*pp[lowerAddr[face]];
}

return mDotPrime;
```

Listing 15.15 A simplified version of the *flux()* function where the flux correction *mDotPrime* is computed

In Listing 15.15 the correction flux *mDotPrime* (Eq. 15.101) is basically evaluated by performing a loop over the faces using the upper and lower coefficients of the matrix and multiplying these coefficients with the corresponding cell values.

Finally the velocity and pressure at cell centroids are updated using the script shown in Listing 15.16,

```
scalar URF = mesh.equationRelaxationFactor("pp");
p += URF*pp;
p.correctBoundaryConditions();
U -= fvc::grad(pp)*DU;
U.correctBoundaryConditions();
```

Listing 15.16 Update of the velocity and pressure fields at cell centroids

where the variable URF is the explicit relaxation factor for pressure update λ^p, necessary for a stable SIMPLE solver.

simpleFoamImproved

In simpleFoamImproved the Rhie-Chow interpolation is extended to account for the relaxation of the velocity field. This is translated into the syntax presented below in Listing 15.17 that expands the generic Rhie-Chow interpolation to include the additional term in Eq. (15.196).

```
// Rhie-Chow interplation
mdotf =   (U_avg_f & mesh.Sf()) - ( (DUf*( gradp_f - gradp_avg_f)) &
mesh.Sf() )
        +(scalar(1) - URFU)*(mdotf.prevIter() - (U_avg_prevIter_f &
mesh.Sf()));
```

Listing 15.17 Improved Rhie-Chow interpolation accounting for under relaxation

Thus the fields mdotf.prevIter() and U.prevIter() need to be defined.

simpleFoamTransient

In simpleFoamTransient the Rhie-Chow interpolation is extended to account for the transient term. Thus the expression for the mass flow rate becomes (Listing 15.18).

```
// Rhie-Chow interplation
mdotf =   (U_avg_f & mesh.Sf()) - ( (DUf*( gradp_f - gradp_avg_f)) &
mesh.Sf() )
        +(scalar(1) - URFU)*(mdotf.prevIter() - (U_avg_prevIter_f &
mesh.Sf()))
        + DTf*( mdotf_old - (U_old_f& mesh.Sf()));
```

Listing 15.18 Rhie-Chow interpolation accounting for the effects of under-relaxation and the unsteady term

Furthermore the main loop is modified to add a transient loop, with the main code becoming as shown in Listing 15.19.

```
        pimpleControl pimple(mesh);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

    Info<< "\nStarting time loop\n" << endl;
    while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "setDeltaT.H"
        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        scalar iter=0;
        while (pimple.loop())
        {
            iter++;
            Info<< "Iteration = " << iter << nl << endl;
            //
            U.storePrevIter();
            mdotf.storePrevIter();

            // Pressure-velocity SIMPLE corrector

            #include "UEqn.H"
            #include "ppEqn.H"


        }
        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
    }
```

Listing 15.19 The main loop used for solving unsteady flow problems

simpleFoamBuoyancy

The simpleFoamBuoyancy solver adds to simpleFoamTransient the following capabilities: (i) the solution of the energy equation, (ii) the inclusion of a body source term in the momentum equation, and (iii) an account of the effects of the body force term redistribution in the Rhie-Chow interpolation.

The codes used to introduce these modifications are shown in Listings (15.20), (15.21), and (15.22).

The code needed to solve the energy equation is given in Listing (15.20).

```
// Solve the Energy equation

  fvScalarMatrix TEqn
  (
          fvm::ddt(T)
        + fvm::div(phi, T)
        - fvm::laplacian(K, T)
  );

   TEqn.relax();
   TEqn.solve();
```

Listing 15.20 The script used to solve the energy equation

The source term in the momentum equation is implemented as (Listing 15.21),

```
surfaceVectorField B_f = linearInterpolate(- beta*(T-To)*g);
volVectorField B_reconstructed = fvc::average(B_f);

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

Listing 15.21 Accounting for the body force term source in the momentum equation

while the calculation of the mass fluxes using the Rhie-Chow interpolation are as displayed in Listing (15.22).

```
surfaceVectorField B_reconstructed_f =
linearInterpolate(B_reconstructed);
// Rhie-Chow interplation
phi = (U_avg_f & mesh.Sf())
     - DUf*( (gradp_f*mesh.magSf())-(gradp_avg_f&mesh.Sf()))
        + (scalar(1) - URFU)*(phi.prevIter() - (U_avg_prevIter_f &
mesh.Sf()))
+ DTf*(phi_old - (U_old_f& mesh.Sf()))
+ DUf* ( (B_f& mesh.Sf()) - (B_reconstructed_f& mesh.Sf()) )  ;
```

Listing 15.22 The modified Rhie-Chow interpolation accounting for body forces

15.11 Closure

This chapter presented the segregated pressure-based approach for solving incompressible flow problems on collocated grids. It also demonstrated that the success of the Rhie-Chow interpolation on collocated grids is due to its formation of a pseudo-momentum equation at the cell face that has a tight pressure gradient stencil similar to the one resulting from a staggered grid formulation. In addition, the details of implementing the most commonly encountered boundary conditions in the momentum and pressure-correction equations were discussed. The next chapter will extend the pressure based method to predict compressible fluid flow at all speeds.

15.12 Exercises

Exercise 1
A portion of a water-supply system is shown in Fig. 15.31. The flow rate $\dot{m}$ in a pipe section is given by

$$\dot{m} = C\Delta p$$

where Δp is the pressure drop over the length of the pipe section, and C is the hydraulic conductance. The following data is known:

$$\begin{aligned} p_1 &= 400,\ p_2 = 350 \\ \dot{m}_F &= 25 \\ C_A &= 0.4, C_B = 0.2, C_C = 0.1, C_D = 0.3, C_E = 0.2 \end{aligned}$$

Find p_3, p_4, p_5, $\dot{m}_A, \dot{m}_B, \dot{m}_C, \dot{m}_D$ and $\dot{m}_E$ using the following procedure

- Start with a guess for p_3, p_4, and p_5.
- Compute $\dot{m}^*$ values based on the guessed pressures.
- Construct the pressure-correction equations and solve for p'_3, p'_4 and p'_5.
- Update the pressures and the $\dot{m}^*$ values

Do you need to iterate? Why?

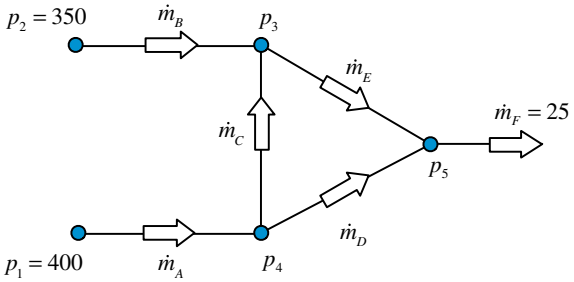


Fig. 15.31 A portion of a water-supply system

Exercise 2

A one dimensional flow through a porous material is governed by

$$c|u|u + \frac{dp}{dx} = 0$$

where c is a constant. The continuity equation is

$$\frac{du}{dx} = 0$$

$$x_2 - x_1 = 1 \quad x_3 - x_2 = 2$$
$$S_A = 3 \qquad S_B = 2$$

Use the SIMPLE procedure for the grid shown in Fig. 15.32 to compute p_2, u_A, and u_B from the following data:

$$c_A = 0.3 \quad c_B = 0.15$$
$$p_1 = 150 \quad p_3 = 18$$

with the size and area at the center of each control volume given by

$$\Delta x_A = 1; A_A = 3$$
$$\Delta x_B = 3; A_B = 2$$

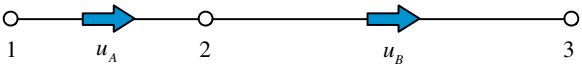


Fig. 15.32 One dimensional flow in a porous material

Exercise 3

In the Steady, one dimensional, constant density situation shown in Fig. 15.33, the velocity u is calculated at locations B and C, while the pressure is calculated at locations 1 and 2. The velocity correction formulae are written as

$$u_B = D_B(p_1 - p_2) \quad \text{and} \quad u_C = D_C(p_2 - p_3)$$

where the values of D_B and D_C are 3 and 4, respectively. The boundary conditions are $u_A = 5$ and $p_3 = 70$.

(a) If at a given stage in the iteration process, the momentum equations give $u_B^* = 4$ and $u_C^* = 6$, calculate the values of p_1 and p_2.
(b) Explain how you could obtain the values of p_1 and p_3 if the right hand boundary condition is given as $u_C = 5$ instead of $p_3 = 70$.

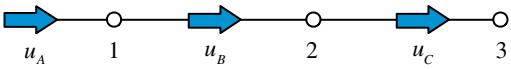


Fig. 15.33 Incompressible flow in a one dimensional domain

Exercise 4

Consider the main control volume shown in Fig. 15.34. A staggered mesh is used with the u and v velocity components stored as shown. The following quantities are given: $u_w = 7$, $v_s = 3$, $p_N = 0$ and $p_E = 50$. The flow is steady and the density is constant. The momentum equations for u_e and v_n are given by:

$$u_e = -D_e(p_E - p_C)$$
$$v_n = -D_n(p_N - p_C)$$

Also given $D_e = 2$, $D_n = 1.6$, and the control volume has $\Delta x = \Delta y = 1$.

(a) Starting with a guessed value of $p_C^{(n)} = 50$, use the SIMPLE algorithm to find u_e and v_n.
(b) Is an iteration loop needed for this problem? Explain.

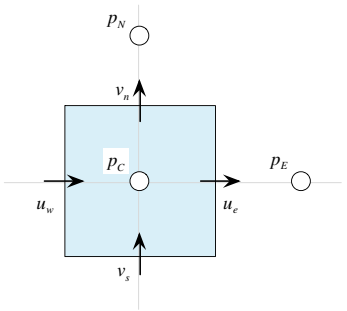


Fig. 15.34 A main control volume in a two-dimensional staggered grid arrangement

Exercise 5

Consider the simplified one-dimensional Forchheimer model for flow in porous media given by

$$bu^2 = -k\frac{dp}{dx}$$

with the continuity equation given by

$$\frac{d(\varepsilon u)}{dx} = 0$$

In the above equations b is a constant and ε is the porosity coefficient that accounts for the effective porous area.

Devise a SIMPLE-like procedure to compute p_C, u_e, and u_w for the following data:

$$\begin{aligned} \Delta x &= 0.1; \quad \Delta y = 1 \\ b_W &= 5; \quad b_C = 4; b_E = 3 \\ \varepsilon_e &= 0.9; \quad \varepsilon_w = 0.6 \\ p_W &= 40; \quad p_E = -200 \end{aligned}$$

Start with the following initial values for velocity and pressure (Fig. 15.35):

$$u_e^* = u_w^* = 3 \text{ and } p_C = -100.$$

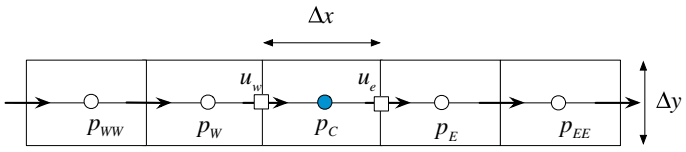


Fig. 15.35 One-dimensional Forchheimer model for flow in porous media

Exercise 6

Compute the interface velocities u_e and u_w using the Rhie-Chow interpolation and compare it to the averaged values $\bar{u}_e$ and $\bar{u}_w$ knowing the following data (Fig. 15.36):

$$\begin{aligned} &p_{WW} = 10; p_W = 12; p_C = 16; p_E = 24; p_{EE} = 40, \text{and} \\ &u_W = 5; u_C = 10; u_E = 40. \end{aligned}$$

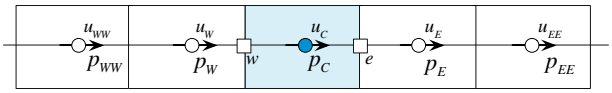


Fig. 15.36 A one dimensional collocated grid

Exercise 7
In OpenFOAM® develop a SIMPLEC pressure correction algorithm by modifying the SIMPLE algorithm described in this chapter. Hint: in order to find the summation of the extra diagonal coefficients use the H1() function of the fvMatrix.

Exercise 8
Check the pisoFoam solver located in $FOAM_SRC/../applications/solvers/incompressible/pisoFoam/pisoFoam.C and compare it with the algorithm described in this chapter, i.e., "The Collocated PISO Algorithm". Find out the inconsistency with the standard OpenFOAM® implementation.

Exercise 9
Develop a pressure correction PISO algorithm for OpenFOAM®.

References

1. Patankar SV (1981) A calculation procedure for two dimensional elliptic situations. Numer Heat Transfer 4(4):409–425
2. Patankar SV (1980) Numerical heat transfer and fluid flow. Hemisphere, NY
3. Patankar SV, Spalding DB (1972) A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. Int J Heat Mass Transf 15(10):1787–1806
4. Harlow FH, Welch JE (1965) Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. Phys Fluids 8(12):2182–2189
5. Van Doormaal JP, Raithby GD (1985) An evaluation of the segregated approach for predicting incompressible fluid flows. ASME Paper 85-HT-9, Presented at the national heat transfer conference, Denver, Colorado
6. Raithby GD, Schneider GE (1979) Numerical solution of problems in incompressible fluid flow: treatment of the velocity-pressure coupling. Numer Heat Transfer, Part A 2(4):417–440
7. Patankar SV (1975) Numerical prediction of three-dimensional flow. In Launder BE (ed) studies in convection: theory, measurement, and application, vol 1. Academic, New York, pp 1–9
8. Rhie CM, Chow WL (1983) Numerical study of the turbulent flow past an airfoil with trailing edge separation. AIAA J 21:1525–1532
9. Rhie CM (1988) A three-dimensional passage flow analysis method aimed at centrifugal impellers. Comput Fluids 13:443–460
10. Majumdar S (1988) Role of under relaxation in momentum interpolation for calculation of flow with nonstaggered grids. Numer Heat Transfer 13:125–132
11. Miller TF, Schmidt FW (1988) Use of a pressure-weighted interpolation method for the solution of incompressible Navier-Stokes equations on a nonstaggered grid system. Numer Heat Transfer 14:213–233
12. Karki KC, Patankar SV (1988) Calculation procedure for viscous incompressible flows in complex geometries. Numer Heat Transfer 14:295–307
13. Choi SK, Nam HY, Cho M (1993) Use of the momentum interpolation method for numerical solution of incompressible flows in complex geometries: choosing cell face velocities. Numer Heat Transfer, Part B 23:21–41
14. Choi SK, Nam HY, Lee YB, Cho M (1993) An efficient three-dimensional calculation procedure for incompressible flows in complex geometries. Numer Heat Transfer, Part B 23:387–400

15. Choi SK, Nam HY, Cho M (1994) Use of staggered and nonstaggered grid arrangements for incompressible flow calculations on nonorthogonal grids. Numer Heat Transfer, Part B 25 (2):193–204
16. Choi SK, Nam HY, Cho M (1994) Systematic comparison of finite-volume calculation methods with staggered and nonstaggered grid arrangements. Numer Heat Transfer, Part B 25 (2):205–221
17. Van Doormaal JP, Raithby GD (1984) Enhancement of the SIMPLE method for predicting incompressible fluid flows. Numer Heat Transfer 7:147–163
18. Issa RI (1982) Solution of the implicit discretized fluid flow equations by operator splitting. Mechanical Engineering Report, FS/82/15, Imperial College, London
19. Maliska CR, Raithby GD (1983) Calculating 3-D fluid flows using non-orthogonal grid. In: Proceedings of the third international conference on numerical methods in laminar and turbulent flows, Seattle, pp 656–666
20. Acharya S, Moukalled F (1989) Improvements to incompressible flow calculation on a non-staggered curvilinear grid. Numer Heat Transfer, Part B 15:131–152
21. Spalding DB (1980) Mathematical modelling of fluid mechanics, heat transfer and mass transfer processes. Mechanical Engineering Department Report HTS/80/1, Imperial College of Science, Technology and Medicine, London
22. Moukalled F, Darwish M (2000) A unified formulation of the segregated class of algorithms for fluid flow at all speeds. Numer Heat Transfer, Part B 37:103–139
23. Darwish M, Asmar D, Moukalled F (2004) A comparative assessment within a multigrid environment of segregated pressure-based algorithms for fluid flow at all speeds. Numer Heat Transfer, Part B 45(1):49–74
24. Jang DS, Jetli R, Acharya S (1986) Comparison of the PISO, SIMPLER and SIMPLEC algorithms for the treatment of the pressure-velocity coupling in steady flow problems. Numer Heat Transfer 10:209–228
25. Yen RH, Liu CH (1993) Enhancement of the SIMPLE algorithm by an additional explicit corrector step. Numer Heat Transfer, Part B 24:127–141
26. Mecinger J (2012) An alternative finite volume discretization of body force field on collocated grids. In: Petrova R (ed) Finite volume method-powerful means of engineering design. ISBN:978-953-51-0445-2
27. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org

Chapter 16
Fluid Flow Computation: Compressible Flows

Abstract The previous chapter presented the methodology for solving incompressible flow problem using pressure based algorithms. In this chapter these algorithms are extended to allow for the simulation of compressible flows in the various Mach number regimes, i.e., over the entire spectrum from subsonic to hypersonic speeds. While incompressible flow solutions do not generally require solving the energy equation, compressibility effects couple hydrodynamics and thermodynamics necessitating the simultaneous solution of the continuity, momentum, and energy equations. The dependence of density on pressure and temperature, a relation expressed via an equation of state, further complicates the velocity-pressure coupling present in incompressible flows. The derivation of the pressure correction equation now involves a density correction that introduces to the equation a convection-like term, in addition to the diffusion-like term introduced by the velocity correction. Another difficulty is introduced by the complex boundary conditions that arise in compressible flow problems. Details on resolving all these issues are presented throughout this chapter.

16.1 Historical

Computational Fluid Dynamics methods have been traditionally classified into two families denoted by density-based and pressure-based. Specifically density-based methods have historically dominated the simulation of transonic and supersonic flows usually encountered in the aeronautics industry, and were well-established when the SIMPLE algorithm was first instigated. SIMPLE, a pressure-based method, was initially developed to address this shortcoming and was quite efficient in resolving incompressible and low Mach number flows.

Early on, efforts were directed towards extending the operation of each of these approaches to flow regimes customarily dominated by the other. Harlow and Amsden [1, 2] were amongst the earliest to simulate fluid flow at all speeds. In their work, the use of pressure as a main variable in preference to density was presented an

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_16

advantage since its variations remained finite irrespective of the Mach number value. Nonetheless it was the work of Patankar [3] that provided a clear resolution to this problem, and allowed for SIMPLE-based methods to genuinely develop into methods capable of resolving fluid flow at all speeds [4–13]. The critical development was the reformulation of the pressure equation to include density and velocity correction such that the type of the equation changed from purely elliptic for incompressible flows to hyperbolic in transonic and supersonic compressible flows [14, 15]. This allowed the SIMPLE-family of methods to seamlessly solve flow problems across the entire Mach number spectrum, with pressure playing the dual role of affecting density in the limit of high Mach compressible flow and velocity in the limit of incompressible flow [16], in order to enforce mass conservation.

16.2 Introduction

An important advantage of the pressure-based approach is its ability to resolve fluid flows in the various Mach number regimes without any artificial treatment to promote convergence and stabilize computations. This ability of the pressure based method is due to the dual role the pressure plays in compressible flows [17], which can best be described by considering the following two extreme cases:

1. At very low Mach numbers, the pressure gradient needed to establish the flow field through momentum conservation is so small that it does not significantly influence the density, and the flow can be considered to be incompressible. Hence, density and pressure in addition to density and velocity are very weakly related indicating that variations in density are not sensitive to variations in velocity. In this case the continuity equation can no longer be considered as an equation for density, rather, it acts as a constraint on the velocity field.
2. At hypersonic speeds, changes in velocity become relatively small as compared to the magnitude of the velocity indicating that variations in pressure do significantly affect density. Consequently, the pressure now acts on density alone through the equation of state to satisfy mass conservation [16, 18] and the continuity equation can be viewed as the equation for density.

The above limiting cases highlight the dual role played by pressure in compressible flow situations. It clearly shows that pressure acts on both the density field through the equation of state and the velocity field via the gradient in the momentum equation to enforce mass conservation. This dual role explains the success of the pressure-based approach to predict fluid flow at all speeds. This fact however did not deter workers in the density based track from using the artificial compressibility technique [19] to develop methods capable of solving fluid flow at all speeds. To overcome degradation in performance due to the stiff matrices encountered in these methods, preconditioning of the resulting stiff matrices was introduced and several methods [20, 21] using this technique have recently appeared in the literature.

Similarly several pressure-based methods [22] for predicting fluid flow at all speeds following either a staggered grid approach [23] or a collocated variable formulation have been developed. While in some methods primitive variables were used, others employed the momentum components as dependent variables. Some workers adopted the stream-wise directed density-retardation concept, which is controlled by Mach-number-dependent monitor functions [24, 25], to account for the hyperbolic character of the conservation laws in the transonic and supersonic regimes. Other techniques used the first order upwind scheme for evaluating the density at the control volume faces at high Mach number values and the central difference scheme at low values [26].

This chapter extends the collocated pressure-based technique developed in the previous chapter allowing the simulation of fluid flows at all Mach number values. The adopted method is easy to implement, highly accurate, and does not require any explicit addition of damping terms to improve robustness or to properly resolve shock waves.

16.3 The Conservation Equations

The conservation equations for solving compressible flow problems include the continuity, momentum, and energy equations. For a Newtonian fluid behaving as an ideal gas, these equations can be expressed as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{16.1}$$

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = \nabla \cdot \{\mu \nabla \mathbf{v}\} - \nabla p + \nabla \cdot \left\{\mu (\nabla \mathbf{v})^{\mathrm{T}}\right\} - \frac{2}{3}\nabla(\mu \nabla \cdot \mathbf{v}) + \mathbf{f}_b \tag{16.2}$$

$$\frac{\partial}{\partial t}\left(\rho c_p T\right) + \nabla \cdot \left[\rho c_p \mathbf{v} T\right] = \nabla \cdot [k \nabla T] + \rho T \frac{D c_p}{D t} + \frac{D p}{D t} - \frac{2}{3}\mu \Psi + \mu \Phi + \dot{q}_V \tag{16.3}$$

where the form of the energy equation adopted here is the one expressed in terms of temperature. The above set of equations should be appended by an equation of state relating density to pressure and temperature, i.e., $\rho = \rho(p, T)$, which for an ideal gas is given by

$$\rho = \frac{p}{RT} \tag{16.4}$$

where R is the gas constant.

In the derivations to follow, superscript n refers to values used at the beginning of an iteration, superscript * indicates values updated once during an iteration, and superscript ** refers to values updated twice during the same iteration.

16.4 Discretization of the Momentum Equation

The discretized momentum equation, Eq. (16.2), over the control volume C shown in Fig. 16.1 is similar to its incompressible form given in Chap. 15. The only two differences are related to the interpolation of density to the interface and the additional term $-(2/3)\nabla(\mu\nabla\cdot\mathbf{v})$ involving the bulk viscosity. Starting with the first difference, the density in compressible flows is no longer constant and since it is stored at the control volume centroids it has to be interpolated to find its value at the control volume faces where it is needed for computing the mass flow rate. The use of a linear interpolation profile (central difference) causes oscillation at high speeds. Thus a bounded upwind biased scheme should be used. Any of the bounded convective schemes presented in Chaps. 11 and 12 can be adopted for that purpose.

The second difference is the additional term involving $\nabla(\mu\nabla\cdot\mathbf{v})$. This term has not been discretized so far and its discretized form is obtained by making use of Eq. (2.85) based on which the volume integral of the gradient of a scalar quantity is

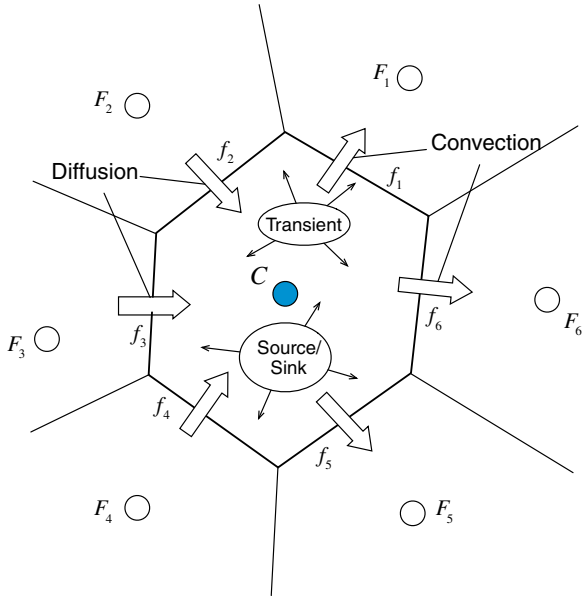


Fig. 16.1 A schematic of a control volume C with its neighbors

transformed into a surface integral and then into a summation of fluxes over the faces of the control volume according to

$$\int_{V_C} [\nabla(\mu\nabla\cdot\mathbf{v})]dV = \int_{\partial V_C} (\mu\nabla\cdot\mathbf{v})d\mathbf{S} = \sum_{f\sim nb(C)} (\mu\nabla\cdot\mathbf{v})_f\mathbf{S}_f \tag{16.5}$$

The divergence of the velocity vector at the faces is computed as

$$(\mu\nabla\cdot\mathbf{v})_f = \mu_f^{(n)}\left[\left(\frac{\partial u}{\partial x}\right)_f^{(n)} + \left(\frac{\partial v}{\partial y}\right)_f^{(n)} + \left(\frac{\partial w}{\partial z}\right)_f^{(n)}\right] \tag{16.6}$$

where the gradient of $\phi = u, v$ or w is interpolated linearly to the face

$$\left(\frac{\partial\phi}{\partial x}\right)_f^{(n)} = g_C\left(\frac{\partial\phi}{\partial x}\right)_C^{(n)} + g_F\left(\frac{\partial\phi}{\partial x}\right)_F^{(n)} \tag{16.7}$$

The final discretized form of the momentum equation is given by Eq. (15.70) with its coefficients given by Eq. (15.71) with the term $-(2/3)\sum_{f\sim nb(C)}(\mu\nabla\cdot\mathbf{v})_f\mathbf{S}_f$ added to the source term in that equation.

As for incompressible flow problems, the algebraic equations are under relaxed and written in the form of Eq. (15.78), which is suitable for the derivation of the pressure correction equation.

16.5 The Pressure Correction Equation

The pressure correction equation for compressible flows is obtained by a simple extension of that for incompressible flows. The difference is related to variations in density which are accounted for by defining a density correction field ρ' and relating it to the pressure-correction field through a pressure-density relation. This, however, yields substantial differences in the treatment of boundary conditions, as will be explained later in the chapter.

For an ideal gas the relation between pressure and density is written as

$$\rho RT = p \tag{16.8}$$

Using this relation, an equation relating density correction to pressure correction can be derived by expanding Eq. (16.8) via a Taylor series to yield

$$\rho|_{(p^{(n)}+p')} = \rho|_{(p^{(n)})} + \frac{\partial\rho}{\partial p}\,p' = \rho^* + \rho' \Rightarrow \rho' = \frac{\partial\rho}{\partial p}p' = \frac{1}{RT}p' = C_\rho p' \tag{16.9}$$

The corrected pressure, density, velocity, and mass flow rate fields are defined as

$$\begin{aligned} p &= p^{(n)} + p' \\ \rho &= \rho^* + \rho' \\ \mathbf{v} &= \mathbf{v}^* + \mathbf{v}' \\ \dot{m} &= \dot{m}^* + \dot{m}' \end{aligned} \tag{16.10}$$

and the semi-discretized continuity equation can be written in terms of the correction fields as

$$\frac{\left(\rho_C^* + \rho_C' - \overset{o}{\rho_C}\right)}{\Delta t} V_C + \sum_{f \sim nb(C)} \left(\dot{m}_f^* + \dot{m}_f'\right) = 0 \tag{16.11}$$

where

$$\begin{aligned} \dot{m}_f &= \left(\rho_f^* + \rho_f'\right)\left(\mathbf{v}_f^* + \mathbf{v}_f'\right) \cdot \mathbf{S}_f \\ &= \underbrace{\rho_f^* \mathbf{v}_f^* \cdot \mathbf{S}_f}_{\dot{m}_f^*} + \underbrace{\rho_f^* \mathbf{v}_f' \cdot \mathbf{S}_f + \rho_f' \mathbf{v}_f^* \cdot \mathbf{S}_f + \rho_f' \mathbf{v}_f' \cdot \mathbf{S}_f}_{\dot{m}_f'} \end{aligned} \tag{16.12}$$

The second order correction term $\rho_f' \mathbf{v}_f' \cdot \mathbf{S}_f$ is usually neglected because it is considerably smaller than other terms. This approximation does not influence the convergence rate except during the first few iterations of the solution process. In addition, the final solution is not affected, since at the state of convergence, the correction fields vanish.

Using the Rhie-Chow interpolation for $\mathbf{v}_f^*$ and $\mathbf{v}_f', \dot{m}_f^*$ and $\dot{m}_f'$ are respectively expressed as

$$\dot{m}_f^* = \underbrace{\rho_f^* \overline{\mathbf{v}_f^*} \cdot \mathbf{S}_f - \rho_f^* \overline{\mathbf{D}_f^{\mathbf{v}}} \left(\nabla p_f^{(n)} - \overline{\nabla p_f^{(n)}}\right) \cdot \mathbf{S}_f}_{\rho_f^* \mathbf{v}_f^* \cdot \mathbf{S}_f} \tag{16.13}$$

and

$$\begin{aligned} \dot{m}_f' &= \underbrace{\rho_f^* \overline{\mathbf{v}_f'} \cdot \mathbf{S}_f - \rho_f^* \overline{\mathbf{D}_f^{\mathbf{v}}} \left(\nabla p_f' - \overline{\nabla p_f'}\right) \cdot \mathbf{S}_f}_{\rho_f^* \mathbf{v}_f' \cdot \mathbf{S}_f} + \underbrace{\left(\frac{\dot{m}_f^*}{\rho_f^*} \cdot \mathbf{S}_f\right) C_{\rho,f} p_f'}_{\rho_f' \mathbf{v}_f \cdot \mathbf{S}_f} \\ &= -\rho_f^* \overline{\mathbf{D}_f^{\mathbf{v}}} \nabla p_f' \cdot \mathbf{S}_f + \left(\frac{\dot{m}_f^*}{\rho_f^*} \cdot \mathbf{S}_f\right) C_{\rho,f} p_f' + \left(\rho_f^* \overline{\mathbf{v}_f'} \cdot \mathbf{S}_f + \rho_f^* \overline{\mathbf{D}_f^{\mathbf{v}}}\, \overline{\nabla p_f'} \cdot \mathbf{S}_f\right) \\ &= -\rho_f^* \overline{\mathbf{D}_f^{\mathbf{v}}} \nabla p_f' \cdot \mathbf{S}_f + \left(\frac{\dot{m}_f^*}{\rho_f^*} \cdot \mathbf{S}_f\right) C_{\rho,f} p_f' - \underline{\rho_f^* \overline{\mathbf{H}_f}[\mathbf{v}'] \cdot \mathbf{S}_f} \end{aligned} \tag{16.14}$$

where the second order term is neglected. Note the substitution of ρ_f' with $C_{\rho,f} p_f'$.

The underlined term in Eq. (16.14) presents the same difficulties as for the incompressible algorithm and is usually dropped from the equation. Neglecting this term, the correction to the mass flow rate becomes

$$\dot{m}'_f = -\rho^*_f \overline{\mathbf{D}^{\mathbf{v}}_f} \nabla p'_f \cdot \mathbf{S}_f + \left(\frac{\dot{m}^*_f}{\rho^*_f} \cdot \mathbf{S}_f \right) C_{p,f} p'_f \tag{16.15}$$

where the first term on the right hand side of Eq. (16.15) is similar to that arising in incompressible flow while the second term is the new density correction contribution. This second term is important as it transforms the pressure correction equation from an elliptic equation to a hyperbolic one capable of resolving shock waves that may arise at supersonic and hypersonic speeds. This allows the compressible SIMPLE algorithm to be used for predicting fluid flow at all speeds without the need for any special preconditioning.

More insight can be gained through a simple normalization procedure whereby Eq. (16.15) is divided by $\left(\dot{m}^*_f \cdot \mathbf{S}_f\right) C_{p,f} \Big/ \rho^*_f$ yielding a weighting factor of 1 for the p'_f term, and a weighting factor proportional to $1/(M^2)$ (where M is the Mach number of the flow) for the $\nabla p'_f$ term, i.e.,

$$\dot{m}'_f = -\frac{RT\left(\rho^*_f\right)^2 \overline{\mathbf{D}^{\mathbf{v}}_f}}{\left(\dot{m}^*_f \cdot \mathbf{S}_f\right)} \nabla p'_f \cdot \mathbf{S}_f + p'_f \tag{16.16}$$

For flows at low Mach number values, the $\nabla p'$ correction term dominates returning the equation to an elliptic form as in the incompressible case. On the other hand, for flows at very high Mach number values the p'_f correction term can no longer be neglected giving a hyperbolic character to the correction equation. This combined behavior allows the prediction of fluid flow at all speeds.

Substitution of Eq. (16.14) in the continuity equation, Eq. (16.11), yields the compressible form of the pressure correction equation and is written as

$$\begin{aligned} \frac{V_C}{\Delta t} C_\rho p'_C + \sum_{f \sim nb(C)} \left\{ -\rho^*_f \overline{\mathbf{D}^{\mathbf{v}}_f} \nabla p'_f \cdot \mathbf{S}_f + \left(\frac{\dot{m}^*_f}{\rho^*_f} \right) C_\rho p'_f \right\} \\ = -\left(\frac{\rho^*_C - \overset{\circ}{\rho}_C}{\Delta t} V_C + \sum_{f \sim nb(C)} \dot{m}^*_f \right) + \underline{\sum_{f \sim nb(C)} \overline{\mathbf{H}_f}[\mathbf{v}'] \cdot \mathbf{S}_f} \end{aligned} \tag{16.17}$$

Again the treatment of the underlined term yields the different variants of the SIMPLE family of algorithms. Dropping the underlined term, the pressure correction equation for the SIMPLE algorithm is obtained as

$$\underbrace{\frac{V_C C_\rho}{\Delta t} p'_C}_{transient-like\ term} + \underbrace{\sum_{f\sim nb(C)} C_\rho \left(\frac{\dot{m}^*_f}{\rho^*_f}\right) p'_f}_{convection-like\ term} - \underbrace{\sum_{f\sim nb(C)} \rho^*_f \mathbf{D}^{\mathbf{v}}_f (\nabla p')_f \cdot \mathbf{S}_f}_{diffusion-like\ term}$$
$$= \underbrace{-\frac{\left(\rho^*_C - \overset{\circ}{\rho}_C\right)}{\Delta t} V_C - \sum_{f\sim nb(C)} \dot{m}^*_f}_{source-like\ term} \tag{16.18}$$

Equation (16.18) can be obtained directly by substituting Eq. (16.15) in Eq. (16.11).

It is worth stressing that the convection-like term came about naturally during the derivation of the pressure correction equation and its presence is critical for the ability of the algorithm to resolve flows at all speeds. Moreover, for flows at high Mach number values, density correction is convected (i.e., exhibiting a hyperbolic behavior) and the mathematical operator describing this phenomenon is the first order divergence operator. Thus, contrary to incompressible flows where only the diffusion-like term is present implying that the pressure correction equation exhibits an elliptic behavior, pressure correction solutions of the form $p' + C$ can no longer satisfy the equation. This indicates that while for incompressible flows any pressure value can be set as a boundary condition without affecting the solution, for compressible flows it is important to define the exact value of pressure at the boundaries because the chosen value will affect the final solution.

It should also be noted that because at convergence the correction field is zero, the order of the scheme used to discretize the convection-like term is of no consequence on the accuracy of the final results. However, this is not the case for $\dot{m}^*_f$ where the use of high order schemes in its evaluation does improve the shock capturing properties of the algorithm. Thus, to enhance robustness, it is helpful to use an upwind scheme for the discretization of the convection like term. Further, neglecting the non-orthogonal contribution of the diffusion-like term as explained in Chap. 15, the pressure correction equation and its coefficients become

$$\begin{aligned}
&a^{p'}_C p'_C + \sum_{F\sim NB(C)} a^{p'}_F p'_F = b^{p'}_C \\
&a^{p'}_F = -\rho^*_F \mathcal{D}_f - \left\| -\dot{m}^*_f, 0 \right\| \frac{C_{\rho,f}}{\rho^*_f} \\
&a^{p'}_C = \frac{V_C C_\rho}{\Delta t} + \sum_{f\sim nb(C)} \left(\frac{C_{\rho,f}}{\rho^*_f} \left\| \dot{m}^*_f, 0 \right\| \right) + \sum_{f\sim nb(C)} \rho^*_f \mathcal{D}_f \\
&b^{p'}_C = -\left(\frac{\left(\rho^*_C - \overset{\circ}{\rho}_C\right)}{\Delta t} V_C + \sum_{f\sim nb(C)} \dot{m}^*_f \right) + \underbrace{\sum_{f\sim nb(C)} \rho^*_f \left(\mathbf{D}^{\mathbf{v}}_f \nabla p'_f \right) \cdot \mathbf{T}_f}_{non-orthogonal\ term\ usually\ neglected}
\end{aligned} \tag{16.19}$$

Following the calculation of the pressure-correction field p' by solving Eq. (16.19) the velocity, pressure, density, and mass flow rate fields are corrected using the following equations:

$$\mathbf{v}_C^{**} = \mathbf{v}_C^{*} + \mathbf{v}_C' \quad \mathbf{v}_C' = -\mathbf{D}_C^{\mathbf{v}}(\nabla p')_C \tag{16.20}$$

$$p_C^{*} = p_C^{(n)} + \lambda^p p_C' \tag{16.21}$$

$$\rho_C^{**} = \rho_C^{*} + \lambda^\rho C_\rho p_C' \tag{16.22}$$

$$\dot{m}_f^{**} = \dot{m}_f^{*} + \dot{m}_f' \quad \dot{m}_f' = -\rho_f^{**}\overline{\mathbf{D}_f^{\mathbf{v}}}\nabla p_f' \cdot \mathbf{S}_f + \left(\frac{\dot{m}_f^{*}}{\rho_f^{**}} \cdot \mathbf{S}_f\right) C_{\rho,f} p_f' \tag{16.23}$$

where λ^ρ is the under relaxation factor for density.

16.6 Discretization of The Energy Equation

The discretization of the unsteady, convection, and diffusion terms of the energy equation, Eq. (16.3), follows the general procedures described in previous chapters and will not be repeated.

16.6.1 Discretization of the Extra Terms

The focus here will be on the discretization over the control volume C shown in Fig. (16.1), of the new terms appearing on the right hand side of Eq. (16.3). These are specific to the energy equation and have not been handled during the discretization of the general scalar equation. Many of the terms are treated as source terms and evaluated at the centroid of the element during their integration to ensure a second order accurate discretization.

16.6.1.1 The Specific Heat Term

The discretization of the term involving the specific heat proceeds as follows:

$$\int_{V_C} \rho T \frac{Dc_p}{Dt} dV = \rho_C^{**} T_C^{(n)} \left(\frac{Dc_p}{Dt}\right)_C V_C$$
$$= \rho_C^{**} T_C^{(n)} \left[\frac{c_p^{(n)} - c_p^{\circ}}{\Delta t} + u_C^{**} \left(\frac{\partial c_p}{\partial x}\right)_C^{(n)} + v_C^{**} \left(\frac{\partial c_p}{\partial y}\right)_C^{(n)} + w_C^{**} \left(\frac{\partial c_p}{\partial z}\right)_C^{(n)}\right] V_C \tag{16.24}$$

16.6.1.2 The Substantial Derivative Term

The discretization of the substantial derivative of the pressure term is performed as

$$\int_{V_c} \frac{Dp}{Dt} dV = \left(\frac{Dp}{Dt}\right)_C^{*} V_C = \left[\frac{p_C^{*} - p_C^{\circ}}{\Delta t} + u_C^{**} \left(\frac{\partial p}{\partial x}\right)_C^{*} + v_C^{**} \left(\frac{\partial p}{\partial y}\right)_C^{*} + w_C^{**} \left(\frac{\partial p}{\partial z}\right)_C^{*}\right] V_C \tag{16.25}$$

16.6.1.3 The Dissipation Term

The discretized form of the dissipation term involving the bulk viscosity is obtained as

$$\int_{V_C} \mu \Psi dV = \mu_C^{(n)} \Psi_C^{**} V_C = \mu_C^{(n)} \left[\left(\frac{\partial u}{\partial x}\right)_C^{**} + \left(\frac{\partial v}{\partial y}\right)_C^{**} + \left(\frac{\partial w}{\partial z}\right)_C^{**}\right]^2 V_C \tag{16.26}$$

16.6.1.4 The Viscous Dissipation Term

The discretization of the viscous dissipation term is performed in a way similar to the term involving the bulk viscosity and is given by

$$\int_{V_C} \mu \Phi dV = \mu_C^{(n)} \Phi_C^{**} V_C$$
$$= \mu_C^{(n)} \left\{ \begin{array}{l} 2\left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial z}\right)^2\right]_C^{**} + \left[\left(\frac{\partial u}{\partial y}\right)_C^{**} + \left(\frac{\partial v}{\partial x}\right)_C^{**}\right]^2 + \\ \left[\left(\frac{\partial u}{\partial z}\right)_C^{**} + \left(\frac{\partial w}{\partial x}\right)_C^{**}\right]^2 + \left[\left(\frac{\partial v}{\partial z}\right)_C^{**} + \left(\frac{\partial w}{\partial y}\right)_C^{**}\right]^2 \end{array} \right\} V_C \tag{16.27}$$

16.6.1.5 The Source/Sink Term

The term involving the heat source/sink per unit volume is discretized as

$$\int_{V_C} \dot{q}_V \, dV = (\dot{q}_V)_C V_C \tag{16.28}$$

The discrete forms of the above terms are substituted into the energy equation to yield the algebraic form of the energy equation as described next.

16.6.2 The Algebraic Form of the Energy Equation

Assuming a first order Euler scheme for the discretization of the unsteady term and a high resolution scheme for the discretization of the convection term applied in the context of a deferred correction approach, the final algebraic form of the energy equation can be written as

$$a_C^T T_C + \sum_{F \sim NB(C)} a_F^T T_F = b_C^T \tag{16.29}$$

where the coefficients are given by

$$\begin{aligned}
a_F^T &= -k_f \frac{E_f}{d_{CF}} - \left\| -\dot{m}_f, 0 \right\| (c_p)_f \\
a_C^T &= \dot{a_C} - \sum_{F \sim NB(C)} a_F^T + \sum_{f \sim nb(C)} \dot{m}_f (c_p)_f + \left\| -a_C^{c_p}, 0 \right\| \\
\dot{a_C} &= \frac{\rho_C (c_p)_C V_C}{\Delta t} \quad \overset{\circ}{a_C} = \frac{\overset{\circ}{\rho_C} \left(\overset{\circ}{c_p} \right)_C V_C}{\Delta t} \quad a_C^{c_p} = \rho_C V_C \left(\frac{Dc_p}{Dt} \right)_C \\
b_C^T &= \sum_{f \sim nb(C)} \left(k_f (\nabla T)_f \cdot \mathbf{T}_f \right) - \sum_{f \sim nb(C)} \dot{m}_f (c_p)_f \left(T_f^{HR} - T_f^U \right) + \overset{\circ}{a_C} \overset{\circ}{T_C} \\
&\quad + T_C \left\| a_C^{c_p}, 0 \right\| + \left[\left(\frac{Dp}{Dt} \right)_C + \mu_C \left(-\frac{2}{3} \Psi_C + \Phi_C \right) + (\dot{q}_V)_C \right] V_C
\end{aligned} \tag{16.30}$$

Similar to other variables, under relaxation of the energy equation is usually required.

16.7 The Compressible SIMPLE Algorithm

The various elements of the collocated compressible SIMPLE algorithm are displayed in Fig. 16.2 and can be summarized as follows:

1. To compute the solution at time $t + \Delta t$, start with the solution at time t for pressure, velocity, density, temperature, and mass flow rate fields $p^{(n)}$, $\mathbf{v}^{(n)}$, $\rho^{(n)}$, $T^{(n)}$, and $\dot{m}^{(n)}$, respectively, as the initial guess.

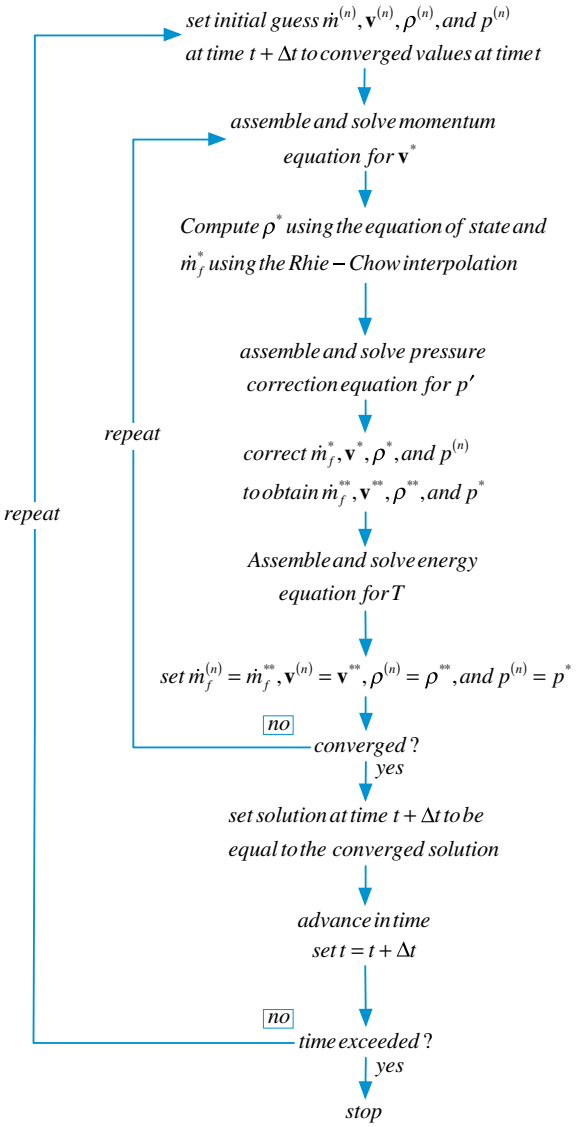


Fig. 16.2 A flow chart of the SIMPLE algorithm for compressible fluid flow

2. Solve the momentum equation given by Eq. (16.2) to obtain a new velocity field $\mathbf{v}^*$.
3. Use the equation of state to calculate a new density field ρ^*.
4. Update the mass flow rate at the control volume faces using the Rhie-Chow interpolation technique (Eq. 16.13) to obtain a momentum satisfying mass flow rate field $\dot{m}^*$.
5. Using the new mass flow rates calculate the coefficients of the pressure correction equation and solve it (Eq. 16.19) to obtain a pressure correction field p'.
6. Update the pressure, density, and velocity fields at the control volume centroids and the mass flow rate at the control volume faces to obtain continuity-satisfying fields using Eqs. (16.20)–(16.23).
7. Solve the energy equation to obtain a new temperature field T^*.
8. Set $\mathbf{v}^{**}, \dot{m}^{**}, \rho^{**}, T^*$, and p^* as the initial guess for velocity, mass flow rate, density, temperature, and pressure.
9. Go back to step 2 and repeat until convergence.
10. Set the solution at time $t + \Delta t$ to be equal to the converged solution.
11. Advance to the next time step by setting the current time t to $t + \Delta t$.
12. Go to step 1 and repeat until the last time step is reached.

16.8 Boundary Conditions

Generally, there is no difference in the implementation of boundary conditions for the momentum equation between incompressible and compressible flow. Therefore the required modifications in the momentum equation are those discussed in Chap. 15 and will not be repeated here. However substantial differences do arise with the pressure correction equation and will form the main subject of this section. The boundary conditions for the energy equation follow the ones described for a general scalar variable ϕ and also will not be repeated here (inlet, outlet, Dirichlet, Van-Neumann, and symmetry conditions).

For a boundary cell, such as the one shown in Fig. 16.3, the continuity equation is written as

$$\frac{\left(\rho_C^* + \rho_C' - \overset{\circ}{\rho}_C\right)}{\Delta t} V_C + \sum_{f \sim nb(C)} \left(\dot{m}_f^* + \dot{m}_f'\right) + \underbrace{\left(\dot{m}_b^* + \dot{m}_b'\right)}_{boundary\,face} = 0 \tag{16.31}$$

where the contribution of the boundary face is separately displayed with $\dot{m}_b^*$ representing the boundary mass flux and $\dot{m}_b'$ its correction. Moreover, expressing the Rhie-Chow interpolation at the boundary faces by adopting the same approach that was used for incompressible flow, the velocity, mass flow rate, and mass flow rate correction at a boundary face are expressed as

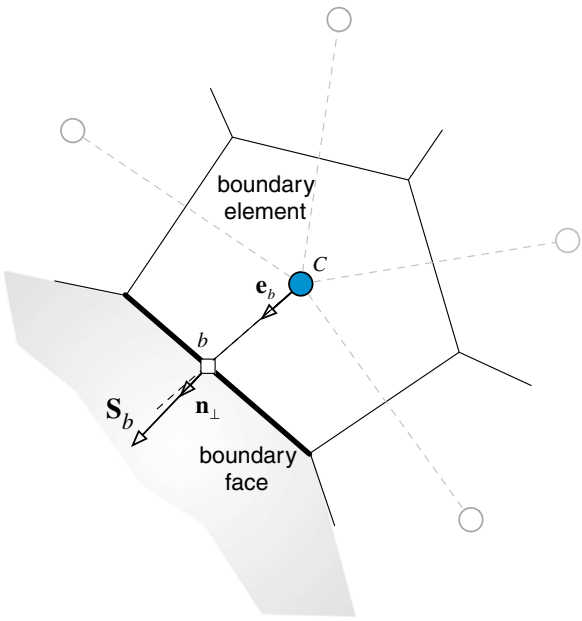


Fig. 16.3 A boundary control volume

$$\underbrace{\mathbf{v}_b^*}_{boundary\,face} = \underbrace{\mathbf{v}_C^* - \mathbf{D}_C\left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right)}_{boundary\,Rhie-Chow} \tag{16.32}$$

$$\dot{m}_b^* = \rho_b^{(n)}\mathbf{v}_C^* \cdot \mathbf{S}_b - \rho_b^* \mathbf{D}_C^{\mathbf{v}}\left(\nabla p_b^{(n)} - \nabla p_C^{(n)}\right) \cdot \mathbf{S}_b \tag{16.33}$$

$$\dot{m}_b' = -\rho_b^* \mathcal{D}_C\left(p_b' - p_C'\right) + \left(\frac{\dot{m}_b^*}{\rho_b^*}\right) C_{\rho,b} p_b' \tag{16.34}$$

The only difference between these expressions and those presented in Chap. 15 is in the correction equation of the mass flow rate, which has an additional term related to density correction. Moreover, there is no difference in the implementation of boundary conditions at a wall and a symmetry plane between incompressible and compressible flows. Therefore the modifications to the boundary elements presented in the previous chapter for wall and symmetry boundary conditions in the pressure correction equation are applicable here with no need to be repeated.

The boundary conditions left to be discussed here are those applicable at the inlet and outlet. For compressible flow, the conditions to be imposed are dictated by the Mach number values. For an inviscid flow, the mathematical type of the equations changes from elliptic to hyperbolic as the flow changes from subsonic to supersonic. Details regarding their implementation are given next.

16.8.1 Inlet Boundary Conditions

At the inlet to a domain the flow may be subsonic or supersonic necessitating different treatment since the flow equations may be either of the elliptic or the hyperbolic type.

16.8.1.1 Subsonic Flow at Inlet

At subsonic speeds several conditions at inlet to a domain can be imposed. This include specified velocity, specified static pressure and velocity direction, or specified stagnation pressure and velocity direction. The last type should be used when transition to supersonic speed occurs within the domain.

$$\text{Specified Velocity}(p_b = ?;\ \dot{m}_b = ?;\ \mathbf{v}_b \textit{ specified})$$

Unlike incompressible flow, since for compressible flow the density depends on pressure, the mass flux remains unknown even with a specified velocity at inlet $\left(\text{i.e, } \dot{m}'_b = \rho'_b \mathbf{v}^*_b \cdot \mathbf{S}_b \neq 0\right)$. At an inlet boundary the coefficient multiplying the pressure correction p'_b is given by

$$a_b^{p'} = C_{\rho,b} \frac{\dot{m}^*_b}{\rho^*_b} \tag{16.35}$$

For implementation in the pressure correction equation, p'_b is expressed in terms of internal nodes and the coefficients at these nodes modified accordingly. For the constant profile case (i.e., $p_b = p_C$), the a_C coefficient is obtained as

$$a_C^{p'} = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f\sim nb(C)} \left(\frac{C_{\rho,f}}{\rho^*_f} \left\| \dot{m}^*_f, 0 \right\| \right) + \sum_{f\sim nb(C)} \rho^*_f \mathcal{D}_f}_{\textit{interior faces contribution}} + \underbrace{C_{\rho,b} \frac{\dot{m}^*_b}{\rho^*_b}}_{\textit{boundary face contribution}} \tag{16.36}$$

The value of the pressure p_b is again obtained by extrapolation from the interior as explained in the incompressible flow chapter.

$$\text{Specified Static Pressure and Velocity Direction } \left(p_b = p_{specified};\ \mathbf{e}_\mathbf{v} \textit{ specified};\ \dot{m}_b = ?;\ \mathbf{v}_b = ?\right)$$

In the case of a specified static pressure at inlet, p_b is known and thus p'_b is set to zero and consequently ρ'_b is also zero. Therefore the implementation is similar to the incompressible case with the inlet treated as a Dirichlet boundary condition. Knowing the velocity direction, its magnitude is computed as in the incompressible

case using Eq. (16.33) leading to an equation similar to Eq. (15.137). The coefficient of the pressure-correction equation becomes

$$a_C^{p'} = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f\sim nb(C)} \left(\frac{C_{\rho,f}}{\rho_f^*} \left\| \dot{m}_f^*, 0 \right\| \right) + \sum_{f\sim nb(C)} \rho_f^* \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\rho_b^* \mathcal{D}_C}_{\text{boundary face contribution}} \tag{16.37}$$

Specified Total Pressure and Velocity Direction ($p_{o,b} = p_{o,\,specified}$; $\mathbf{e_v}$ *specified*; $\dot{m}_b = ?$; $\mathbf{v}_b = ?$)

For this case, the magnitude of the velocity and the pressure at the boundary are unknown while related through the total pressure equation given by

$$p_{o,b} = p_b \left(1 + \frac{\gamma - 1}{2} M_b^2 \right)^{\gamma/(\gamma-1)} \tag{16.38}$$

where b refers to the boundary, $p_{o,b}$ is the total pressure, p_b the static pressure, γ the ratio of specific heats, and M_b the Mach number which is equivalent to

$$M_b = \sqrt{\frac{\mathbf{v}_b \cdot \mathbf{v}_b}{\gamma R T_b}} \tag{16.39}$$

Equation (16.37) can be rearranged to give the static pressure in terms of the total pressure as

$$p_b = p_{o,b} \left(1 + \frac{\gamma - 1}{2} \frac{\left(\dot{m}_b^*\right)^2}{\left(\rho_b^*\right)^2 (\mathbf{e_v} \cdot \mathbf{n} S_b)^2 \gamma R T_b} \right)^{-\gamma/(\gamma-1)} \tag{16.40}$$

where $\mathbf{e_v}$ is the unit vector in the direction of the velocity vector. Differentiating Eq. (16.40) with respect to $\dot{m}_b^*$ gives

$$\frac{dp_b}{d\dot{m}_b^*} = -\frac{\gamma \dot{m}_b^* p_{o,b}}{\left(\rho_b^*\right)^2 (\mathbf{e_v} \cdot \mathbf{n} S_b)^2 \gamma R T_b} \left(1 + \frac{\gamma - 1}{2} \frac{\left(\dot{m}_b^*\right)^2}{\left(\rho_b^*\right)^2 (\mathbf{e_v} \cdot \mathbf{n} S_b)^2 \gamma R T_b} \right)^{-\frac{(2\gamma-1)}{(\gamma-1)}} \tag{16.41}$$

Substituting Eq. (16.41) into Eq. (15.163) an equation for pressure correction function of the mass flux correction is obtained as

$$\begin{aligned} p'_b &= -\frac{\gamma \dot{m}^*_b p_{o,b}}{\left(\rho^*_b\right)^2 (\mathbf{e_v} \cdot \mathbf{n} S_b)^2 \gamma R T_b} \left(1 + \frac{\gamma - 1}{2} \frac{\left(\dot{m}^*_b\right)^2}{\left(\rho^*_b\right)^2 (\mathbf{e_v} \cdot \mathbf{n} S_b)^2 \gamma R T_b}\right)^{-\frac{(2\gamma-1)}{(\gamma-1)}} \dot{m}'_b \\ &= c_b \dot{m}'_b \end{aligned} \tag{16.42}$$

Replacing p'_b in Eq. (16.34) by its equivalent expression given by Eq. (16.42), the mass flux correction becomes

$$\dot{m}'_b = \frac{\rho^*_b \mathcal{D}_b}{1 + \rho^*_b \mathcal{D}_b c_b - \left(\frac{\dot{m}^*_b}{\rho^*_b}\right) C_{\rho,b} c_b} p'_c \tag{16.43}$$

The modified boundary cell coefficient is obtained by substituting $\dot{m}'_b$ from Eq. (16.43) in the expanded continuity equation and is given by

$$a_C^{p'} = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f \sim nb(C)} \left(\frac{C_{\rho,f}}{\rho^*_f} \left\| \dot{m}^*_f, 0 \right\|\right) + \sum_{f \sim nb(C)} \rho^*_f \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\frac{\rho^*_b \mathcal{D}_C}{1 + \rho^*_b \mathcal{D}_C c_b - \left(\frac{\dot{m}^*_b}{\rho^*_b}\right) C_{\rho,b} c_b}}_{\text{boundary face contribution}} \tag{16.44}$$

It should be mentioned that the boundary condition for the energy equation at a subsonic inlet is usually either a specified static temperature T_b or a specified stagnation temperature $T_{o,b}$. If the static temperature is specified, then this is similar to a Dirichlet condition. If the stagnation temperature is specified then at each iteration the value of the static temperature is extracted from the stagnation temperature equation using

$$T_{o,b} = T_b + \frac{\mathbf{v}_b \cdot \mathbf{v}_b}{2 c_p} \tag{16.45}$$

and the obtained value treated as known. Thus a Dirichlet-type boundary condition is also applied.

16.8.1.2 Supersonic Flow at Inlet

Specified static pressure, velocity, and temperature

$$\left(p_b = p_{specified};\ \mathbf{v}_b = \mathbf{v}_{specified};\ T = T_{specified}\right)$$

At a supersonic inlet, values for all variables must be specified (pressure, velocity, and temperature). This is equivalent to a Dirichlet-type condition implying that $\dot{m}'_b = p'_b = 0$. Therefore the $a_C^{p'}$ coefficient of the boundary cell is found to be

$$a_C^{p'} = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f\sim nb(C)} \left(\frac{C_{\rho,f}}{\rho_f^*} \left\| \dot{m}_f^*, 0 \right\| \right) + \sum_{f\sim nb(C)} \rho_f^* \mathcal{D}_f}_{\text{interior faces contribution}} \tag{16.46}$$

16.8.2 Outlet Boundary Conditions

16.8.2.1 Subsonic Flow at Outlet

$$\text{Specified Pressure} \left(p_b = p_{specified};\ \dot{m}_b\,?;\ \mathbf{v}_b\ =\ ? \right)$$

At a subsonic outlet, the pressure is usually prescribed. Therefore the pressure correction p'_b is set to zero while the mass flow rate correction $\dot{m}'_b$ is computed as

$$\dot{m}'_b = -\rho_b^* \mathcal{D}_C \left(p'_b - p'_C \right) + \left(\frac{\dot{m}_b^*}{\rho_b^*} \right) C_{\rho,b} p'_b = \rho_b^* \mathcal{D}_C p'_C \tag{16.47}$$

Since the velocity $\mathbf{v}_b^*$ is not known, it is customary to assume its direction to be that of the upwind velocity $\mathbf{v}_C^*$. The expression of the a_c coefficient in the pressure-correction equation may be written as

$$a_C^{p'} = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f\sim nb(C)} \left(\frac{C_{\rho,f}}{\rho_f^*} \left\| \dot{m}_f^*, 0 \right\| \right) + \sum_{f\sim nb(C)} \rho_f^* \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\rho_b^* \mathcal{D}_c}_{\text{boundary face contribution}} \tag{16.48}$$

For the energy equation a zero flux Neumann-type boundary condition is applied.

$$\text{Specified Mass Flow Rate } \left(\dot{m}_b = \dot{m}_{specified};\ p_b ?\ \mathbf{v}_b ? \right)$$

For a specified mass flow rate at outlet, $\dot{m}'_b$ is zero and is simply dropped from the pressure correction equation with no modifications required for the coefficients of the boundary elements. By setting $\dot{m}'_b$ to zero in Eq. (16.34), an expression for the pressure correction at the boundary as a function of the pressure correction at the boundary cell centroid is obtained as

$$p'_b = \frac{\rho^*_b \mathcal{D}_C}{\rho^*_b \mathcal{D}_C - \left(\frac{\dot{m}^*_b}{\rho^*_b}\right) C_{\rho,b}} p'_C \tag{16.49}$$

allowing the boundary pressure and density to be computed. For the energy equation a zero flux Neumann-type boundary condition is applied.

16.8.2.2 Supersonic Flow at Outlet

At a supersonic outlet none of the variables should be specified and the values of pressure, velocity, density, and temperature are extrapolated from the interior of the domain. Thus both $\dot{m}_b$ and p_b are extrapolated from interior cells. This is equivalent to applying a Neumann boundary condition on pressure-correction, leading to the following modified a_C coefficient

$$a^{p'}_C = \frac{V_C C_\rho}{\Delta t} + \underbrace{\sum_{f \sim nb(C)} \left(\frac{C_{\rho,f}}{\rho^*_f} \left\| \dot{m}^*_f, 0 \right\| \right) + \sum_{f \sim nb(C)} \rho^*_f \mathcal{D}_f}_{\text{interior faces contribution}} + \underbrace{\left(\frac{\dot{m}^*_b}{\rho^*_b}\right) C_{\rho,b}}_{\text{boundary face contribution}} \tag{16.50}$$

16.9 Computational Pointers

16.9.1 uFVM

For compressible flows a major modification to the algorithm arises in the assembly of the pressure equation through the inclusion of the convection like term. This is added to the cfdAssembleMdotTerm shown in Listing 16.1.

```
    % assemble terms X (for compressible flow)
    %     (mdot_f/density_f)*(∂rho/∂p) P'
    % where mdot_f is the newly computed mdot_f
    %
    local_mdot_f = local_FLUXCf_1*(pressure[iElement1]+ Pref) +
local_FLUXCf_2*(pressure[iElement2]+ Pref) + local_FLUXVf;
    %
    local_FLUXCf1 = local_FLUXCf1 +  max(local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace);
    local_FLUXCf2 = local_FLUXCf2 - max(-local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace);
    local_FLUXVf  = local_FLUXVf -(max(local_mdot_f/density_f(iFace),
0.0)*drhodp_f(iFace)*(pressure(iElement1)+ Pref)- max(-local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace)*(pressure(iElement2)+ Pref));
    local_FLUXVf   += -local_mdot_f;
```

Listing 16.1 Script used to assemble the additional terms for the compressible pressure correction equation

The other important change is in the treatment of boundary conditions that now necessitates accounting for a variable density field. For example the supersonic inlet condition is implemented for the pressure correction equation as shown in Listing 16.2.

```
    % assemble terms X (for compressible flow)
    %     (mdot_f/density_f)*(?rho/?p) P'
    % where mdot_f is the newly computed mdot_f
    %
    local_mdot_f = local_FLUXCf_1*(pressure[iElement1]+ Pref) +
local_FLUXCf_2*(pressure[iElement2]+ Pref) + local_FLUXVf;
    %
    local_FLUXCf1 = local_FLUXCf1 +  max(local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace);
    local_FLUXCf2 = local_FLUXCf2 - max(-local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace);
    local_FLUXVf  = local_FLUXVf -(max(local_mdot_f/density_f(iFace),
0.0)*drhodp_f(iFace)*(pressure(iElement1)+ Pref)- max(-local_mdot_f/
density_f(iFace),0.0)*drhodp_f(iFace)*(pressure(iElement2)+ Pref));
    local_FLUXVf   += -local_mdot_f;
```

Listing 16.2 Implementation of a supersonic inlet condition

16.9.2 OpenFOAM®

In this section simpleFoam is extended to handle compressible fluid flow at all speeds. This entails the following modifications to simpleFoam: (i) the addition of the energy equation to be solved simultaneously with the continuity and momentum equations, (ii) the use of an equation of state relating density to temperature and pressure, (iii) and the introduction of the necessary modifications to the pressure correction equation and to a number of boundary conditions. The resulting code is denoted simpleFoamCompressible with many of the extensions added in the form of supplemental include files as shown in Listing 16.3.

The *upwind.H* and *gaussConvectionScheme.H* are used to force an upwind discretization on the convective term of the pressure correction equation. The *bound.H* class is used to bound variables within certain limits.

```
#include "fvCFD.H"
#include "psiThermo.H"
#include "RASModel.H"
#include "upwind.H"
#include "gaussConvectionScheme.H"
#include "bound.H"
#include "simpleControl.H"
#include "totalPressureCompFvPatchScalarField.H"
#include "totalPressureCorrectorCompFvPatchScalarField.H"
#include "totalVelocityFvPatchVectorField.H"
#include "orthogonalSnGrad.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
    simpleControl simple(mesh);

#   include "createFields.H"
```

Listing 16.3 The include files used in simpleFoamCompressible

The *createFields.H* now includes the definition of the density field and other variables and constants related to compressible flow physics. The *psiTermo* class, depicted in Listing 16.4, provides access to the thermophysical relations that are part of the OpenFOAM® library [27], such as the perfect gas law described in Eq. (16.4).

```
Info<< "Reading thermophysical properties\n" << endl;

autoPtr<psiThermo> thermo
(
    psiThermo::New(mesh)
);

Info<< "Calculating field rho\n" << endl;

volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    thermo->rho()
);

volScalarField& p = thermo->p();
volScalarField& h = thermo->he();

thermo->correct();

volScalarField gammaGas
(
    IOobject
    (
        "gammaGas",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    thermo->Cp() /  thermo->Cv()
);

volScalarField RGas
(
    IOobject
    (
        "RGas",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    thermo->Cp() -  thermo->Cv()
);
```

Listing 16.4 An excerpt of the *psiTermo* class

The pressure and enthalpy fields are defined in the *thermo* class, displayed in Listing 16.5, and accessed as references in *createFields.H*.

```
volScalarField& p = thermo->p();
volScalarField& h = thermo->he();

const volScalarField::GeometricBoundaryField& pbf=p.boundaryField();
wordList pbt = pbf.types();
volScalarField pp
(
    IOobject
    (
        "pp",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", p.dimensions(), 0.0),
    pbt
);
```

Listing 16.5 Defining the pressure and enthalpy field in *thermo* class

The velocity is defined as in the incompressible version but the mass flux now includes density in its definition (Listing 16.6).

```
Info << "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
surfaceScalarField mDot
(
    IOobject
    (
        "mDot",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(rho*U) & mesh.Sf()
);
```

Listing 16.6 Defining the velocity and mass flux fields

In compressible flows, setting physical limits on some variables such as density and pressure can enhance robustness, especially during the first few iterations. This prevents variables from assuming non-physical values (like negative densities or pressures). Therefore bounds can be set as part of the case definition, as shown in Listing 16.7, and read in *createFields.H*.

```
dimensionedScalar rhoMax(simple.dict().lookup("rhoMax"));
dimensionedScalar rhoMin(simple.dict().lookup("rhoMin"));

dimensionedScalar pMax(simple.dict().lookup("pMax"));
dimensionedScalar pMin(simple.dict().lookup("pMin"));
```

Listing 16.7 Setting lower and upper bounds for the density and pressure fields

The momentum equation is defined with a slightly modified syntax that accounts for density and thermophysical property relations. The syntax of the linearized formula is given in Listing 16.8.

```
// Solve the Momentum equation
fvVectorMatrix UEqn
(
        fvm::ddt(rho,U)
      + fvm::div(mDot, U)
      - fvm::laplacian(thermo->mu, U)
);

UEqn.relax();

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

Listing 16.8 Syntax used to solve the momentum equation

The first instruction defines the finite volume discretization of the momentum equation in a vector form (the three components of the velocity are solved in a segregated manner despite the vectorial implementation). The system is implicitly relaxed and then solved with an iterative solver.

Once the momentum equation is solved, a new guess of the velocity field is obtained. This velocity field does not necessarily satisfy the continuity equation.

To enforce mass conservation, assembly of the pressure correction equation is now required to correct the velocity. Following Eq. (16.19) the syntax used for that purpose is shown in Listing 16.9.

```
pp = scalar(0.0)*pp;
pp.correctBoundaryConditions();


surfaceScalarField phid
(
      IOobject
      (
       "phid",
       runTime.timeName(),
       mesh,
       IOobject::NO_READ,
       IOobject::NO_WRITE
      ),
    mDot*drhodp/(rhofd)
);

fvMatrix<scalar> ppCompEqn
(
    fvm::ddt(thermo->psi(),pp) +
    fv::gaussConvectionScheme<scalar>
     (
         mesh,
         phid,
         tmp<surfaceInterpolationScheme<scalar> >
         (
             new upwind<scalar>(mesh,phid)
         )
     ).fvmDiv(phid, pp)
  - fvm::laplacian(pDiff, pp)
  + fvc::div(mDot) + fvc::ddt(rho)
)
```

Listing 16.9 Syntax used to assemble the pressure correction equation

As for the incompressible case, in order to avoid checker boarding, the *mDot mass flux field is evaluated* using the Rhie-Chow interpolation but now taking into account also the density field, evaluated based on the *thermo* model as shown in Listing 16.10.

```
…
rho = thermo->rho();

Foam::fv::orthogonalSnGrad<scalar> faceGradient(mesh);

surfaceVectorField gradp_avg_f = linearInterpolate(fvc::grad(p));
surfaceVectorField gradp_f = gradp_avg_f - (gradp_avg_f & ed)*ed +
(faceGradient.snGrad(p))*ed;

surfaceVectorField U_avg_prevIter_f = linearInterpolate(U.prevIter());
surfaceVectorField U_avg_f = linearInterpolate(U);

surfaceScalarField rhofd = upwind<scalar>(mesh,mDot).interpolate(rho);
surfaceScalarField rhof("srho",fvc::interpolate(rho));
surfaceScalarField DUf("srUA",fvc::interpolate(DU,"interpolate((1|
A(U)))"));

volScalarField dt
(
      IOobject
      (
       "dt",
       runTime.timeName(),
       mesh,
       IOobject::NO_READ,
       IOobject::NO_WRITE
      ),
      mesh ,
      dimensionedScalar("dt" ,runTime.deltaT().dimensions(),
runTime.deltaT().value()),
      zeroGradientFvPatchScalarField::typeName
);
surfaceScalarField dt_f = linearInterpolate(dt);
surfaceScalarField drhodp = linearInterpolate(thermo->psi());

scalar UURF = mesh.equationRelaxationFactor("U");

// Rhie-Chow interplation
mDot = rhof*(
              (U_avg_f & mesh.Sf()) - ( (DUf*( gradp_f - gradp_avg_f))
& mesh.Sf() )
            );
           + (scalar(1) - UURF)*(mDot.prevIter() -
( (rhof*U_avg_prevIter_f) & mesh.Sf() ) )
           +    rhof*(DUf/dt_f)*(mDot.prevIter() -
( (rhof*U_avg_prevIter_f) & mesh.Sf() ) );
```

Listing 16.10 Calculation of mass fluxes at cell faces using the Rhie-Chow interpolation

It is worth mentioning that density is interpolated to faces using an upwind scheme in order to mimic the hyperbolic behavior of compressible flows.

The pressure correction equation is fully set and is solved using the syntax displayed in Listing 16.11.

```
ppEqn.solve();
```

Listing 16.11 Syntax for solving the pressure correction equation

After solving the pressure correction equation, variables that depend on pressure correction are updated. For the mass flux field this is performed using the syntax in Listing 16.12, this is similar to the incompressible flux correction.

```
mDot += ppEqn.flux();
```

Listing 16.12 Syntax to update the mass flux field

Where again the flux() function in Listing 16.12 updates the fluxes using directly the matrix coefficients and cell values. A simplified version of the flux() function is shown in Listing 16.13.

```
for (label face=0; face<lowerAddr.size(); face++)
{
    mDotPrime[face] =
        upperCoeffs[face]*pp[upperAddr[face]]
      - lowerCoeffs[face]*pp[lowerAddr[face]];
}

return mDotPrime;
```

Listing 16.13 A simplified version of the *flux( )* function where the flux correction *mDotPrime* is computed

In Listing 16.13 the correction flux *mDotPrime* is basically evaluated by performing a loop over the faces using the upper and lower coefficients of the matrix and multiplying these coefficients with the corresponding cell values.

Finally the velocity, density and pressure at cell centroids are updated using Eqs. (16.20), (16.21), and (16.22), as shown in Listing 16.14, where the variable alphaP is the explicit relaxation factor for pressure and density updates λ^p, necessary for a stable SIMPLE solver.

```
scalar alphaP = mesh.equationRelaxationFactor("pp");

mDot += ppCompEqn.flux();

p += alphaP*pp;
p.correctBoundaryConditions();

rho += alphaP*pp*thermo->psi();

boundMinMax(p, pMin, pMax);
boundMinMax(rho, rhoMin, rhoMax);

U -= fvc::grad(pp)*DU;
U.correctBoundaryConditions();
```

Listing 16.14 Update of the velocity and pressure fields at cell centroids

In order to account for compressibility effects, the energy equation is introduced and the related temperature is calculated. In OpenFOAM®, the energy equation expressed in terms of specific static enthalpy $\left(h = C_p T\right)$, given by Eq. (3.61), is solved as depicted in Listing 16.15.

```
fvScalarMatrix hEqn
(
    fvm::ddt(rho,h)
  + fvm::div(mDot, h)
  - fvm::laplacian(turbulence->alphaEff(), h)
  + fvm::SuSp(-fvc::div(mDot),h)
 ==

    fvc::div(mDot/fvc::interpolate(rho)*fvc::interpolate(p))
  - p*fvc::div(mDot/fvc::interpolate(rho))
);


hEqn.relax();
hEqn.solve();

h.correctBoundaryConditions();

thermo->correct();

gammaGas =  thermo->Cp()/ thermo->Cv();
gammaGas.correctBoundaryConditions();

RGas =  thermo->Cp() -  thermo->Cv();
RGas.correctBoundaryConditions();
```

Listing 16.15 Solving the energy equation

Once the energy equation is solved, the new enthalpy is used to update the temperature and gas properties (e.g., specific heats).

In addition to the main solver, new *total pressure* and *total temperature* boundary conditions are implemented for *subsonic inlet* patches, these are often used boundary conditions for the simulation of compressible flows. The boundary conditions are defined in the directory "derivedFvPatchFields" and are next presented. For a better understanding, it may be beneficial to read Chap. 18 prior to going over the implementation process presented below.

- **totalPressureComp**: This implements the total pressure condition at subsonic inlet. For that purpose the updateCoeffs() function is modified as shown in Listing 16.16, which indicates that after gathering the necessary data from the solver, the boundary static pressure is computed using Eq. (16.40) and the obtained values are stored in the defined *newp* variable.

```
    const fvPatchScalarField& TB =
        patch().lookupPatchField<volScalarField, scalar>("T");

    const fvPatchField<scalar>& RB =
        patch().lookupPatchField<volScalarField, scalar>("RGas");

    const fvPatchField<scalar>& gammaB =
        patch().lookupPatchField<volScalarField, scalar>("gammaGas");

      const fvsPatchField<scalar>&  sphi =
        patch().lookupPatchField<surfaceScalarField, scalar>("mDot");

      const fvPatchField<scalar>&  srho =
    patch().lookupPatchField<volScalarField, scalar>("rho");

      const fvPatchField<vector>& UF =
patch().lookupPatchField<volVectorField, vector>("U");

    scalarField newp = max(min(p0_/pow((scalar(1.0) + (gammaB -
scalar(1.0) )*(sqr(sphi)
                                /(sqr(srho) * sqr(patch().magSf()))))/
(scalar(2.0) * gammaB * RB * TB  ))
               , gammaB/(gammaB - scalar(1.0))), p0_),SMALL);

    operator==
    (
        newp
    );
```

Listing 16.16 Modified updateCoeffs() function for implementing the total pressure boundary condition at inlet

- **totalPressureCorrectorComp**: This is also needed with a total inlet pressure boundary condition. Its role is to deal with the mass flow rate correction at inlet affecting the diagonal coefficient of the boundary element, as expressed by

Eqs. (16.43) and (16.44). In the implementation, the contributions to the coefficients are reset to zero, as can be inferred from Listing 16.17, except for the gradientInternalCoeffs() that is required to return the correct values. This ensures that for both the divergence and laplacian operator one boundary condition is applied.

```
tmp<Field<scalar> >
totalPressureCorrectorCompFvPatchScalarField::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<scalar> >
    (
        new Field<scalar>(this->size(), pTraits<scalar>::zero)
    );
}

tmp<Field<scalar> >
totalPressureCorrectorCompFvPatchScalarField::gradientBoundaryCoeffs()
const
{
    return tmp<Field<scalar> >
    (
        new Field<scalar>(this->size(), pTraits<scalar>::zero)
    );
}

tmp<Field<scalar> >
totalPressureCorrectorCompFvPatchScalarField::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<scalar> >
    (
        new Field<scalar>(this->size(), pTraits<scalar>::zero)
    );
}

tmp<Field<scalar> >
totalPressureCorrectorCompFvPatchScalarField::gradientInternalCoeffs()
const
{
      const fvsPatchField<scalar>&  srUA =
    patch().lookupPatchField<surfaceScalarField, scalar>("srUA");

      const fvPatchField<scalar>&  srho =
    patch().lookupPatchField<volScalarField, scalar>("rho");

    return( deltaM()/(-srUA*patch().magSf()*srho) ); //to remove the
laplacian operator (see gaussLaplacianScheme)
}
```

Listing 16.17 Script used to reset the diagonal coefficients and to return only the diffusion contribution or a total pressure boundary condition at inlet

The modified diagonal coefficient of the pressure correction equation for a boundary element is computed in Listing 16.18. Here the function deltaM() implements Eq. (16.44) as is, while the diffusion term "(-srUA*patch().magSf() *srho)" is removed from the coefficient of the pressure correction equation, as defined by the laplacian operator (see Chap. 8, computational pointers).

```
Field<scalar> totalPressureCorrectorCompFvPatchScalarField::deltaM()
const
{

      const fvPatchField<scalar>&  T =
    patch().lookupPatchField<volScalarField, scalar>("T");

      const fvPatchField<scalar>&  srho =
    patch().lookupPatchField<volScalarField, scalar>("rho");

      const fvPatchField<scalar>&  RB =
    patch().lookupPatchField<volScalarField, scalar>("RGas");

      const fvsPatchField<scalar>&  srUA =
    patch().lookupPatchField<surfaceScalarField, scalar>("srUA");

      const fvsPatchField<scalar>&  sphi =
    patch().lookupPatchField<surfaceScalarField, scalar>("mDot");

    scalarField Dp = patch().magSf()*patch().deltaCoeffs()*srUA;

    scalarField coeff = srho*Dp/(scalar(1.0) + srho*Dp*Cu() - (sphi/
srho)*Cu()/(RB*T));


      return (coeff);
}
```

Listing 16.18 Script used to modify the diagonal coefficient of the boundary element for a total pressure boundary condition at inlet

- **totalTemp**: This function implements for the energy equation the total temperature boundary condition at a subsonic inlet. The idea is to impose a static temperature as a Dirichlet boundary condition using Eq. (16.45). For that purpose the updateCoeffs() function is modified as in Listing 16.19.

```
const fvPatchField<vector>& Up =
    patch().lookupPatchField<volVectorField, vector>("U");

const fvPatchField<scalar>& gammaB =
    patch().lookupPatchField<volScalarField, scalar>("gammaGas");

scalarField gM1ByG = (gammaB - 1.0)/gammaB;

const fvPatchScalarField& TB =
    patch().lookupPatchField<volScalarField, scalar>("T");

const fvPatchField<scalar>& RB =
    patch().lookupPatchField<volScalarField, scalar>("RGas");

scalarField psip = scalar(1.0)/(RB * TB);

operator==
(
    T0_/(1.0 + 0.5*psip*gM1ByG*magSqr(Up))
);
```

Listing 16.19 Modified updateCoeffs() function for implementing the total temperature boundary condition at a subsonic inlet

At each iteration the temperature value is updated based on the inlet total temperature and the boundary velocity.

- **totalVelocity**: This function, described in Listing 16.20, implements the updates to the velocity field required with total conditions applied at a subsonic inlet. For that purpose a Dirichlet boundary condition is used, with velocity values iteratively computed based on the calculated fluxes "mDot" at the boundary itself. The algorithm is based on grabbing the flux field "mDot" (updated with new values after solving the pressure correction equation by invoking the "flux()" function) and dividing the flux by the face area and the corresponding density.

```
const fvsPatchField<scalar>&  sphi =
    patch().lookupPatchField<surfaceScalarField, scalar>("mDot");

const fvPatchField<scalar>& rhop =
    patch().lookupPatchField<volScalarField, scalar>("rho");

vectorField n = patch().nf();
scalarField ndmagS = (n & inletDir())*patch().magSf();

scalarField clip = neg(sphi);

scalarField newvel = (sphi*clip)/(rhop*ndmagS);

operator==(inletDir()*newvel);
);
```

Listing 16.20 Updating the velocity at a subsonic inlet for a total pressure boundary condition

Additionally, a clipping variable is introduced in order to prevent any "back-flow" at the inlet. Here the "clip" variable may assume either a value of zero or one. In fact the "neg" function returns 1 and 0 for negative and positive values, respectively, preventing outward velocities to be accepted. On the other hand, the "inletDir" variable gives the velocity direction at inlet, as defined by the user.

16.10 Closure

In this chapter the incompressible segregated pressure based approach developed in the previous chapter was extended to handle compressible fluid flow at all speeds. This involved modifying the pressure correction equation to include a convection-like term that changes its type from elliptic to hyperbolic. It also required alterations to the momentum equations, the solution of the energy equation, as well as the addition of an equation of state. Just as critical, are the special boundary conditions needed in the simulation of compressible flows. A number of boundary conditions were presented as well as some implementation details.

The needed modifications to the base incompressible code represent a relatively small change to the bulk of any code and yet allow a drastic extension of its capabilities. The next chapter will present the additional techniques needed for dealing with the time averaged Navier-Stokes equations required for solving turbulent flow problems.

16.11 Exercises

Exercise 1 A portion of a gas-supply system is shown in Fig. 16.4. The mass flow rate $\dot{m}$ in a pipe section is given by

$$\dot{m} = \rho C \Delta p$$

where Δp is the pressure drop over the length of the pipe section, ρ is the gas density, and C is the gas conductance. The following data is known:

$$p_1 = 400, p_2 = 350$$
$$\dot{m}_F = 25$$
$$C_A = C_C = 1.2, C_B = 1.4, C_D = 1.6, C_E = 1.8$$

with the density related to the pressure via $p = \rho R'$ with $R' = 2000$.

If the direction of the flow is as shown in the figure, find $p_3, p_4, p_5, \dot{m}_A, \dot{m}_B, \dot{m}_C, \dot{m}_D$ and $\dot{m}_E$ using the following procedure:

- Start with a guess for p_3, p_4, and p_5.
- Compute $\dot{m}^*$ values based on the guessed pressures and densities.
- Construct the pressure-correction equations and solve for p'_3, p'_4 and p'_5.
- Update the pressures and the $\dot{m}^*$ values

Do you need to iterate? Why?

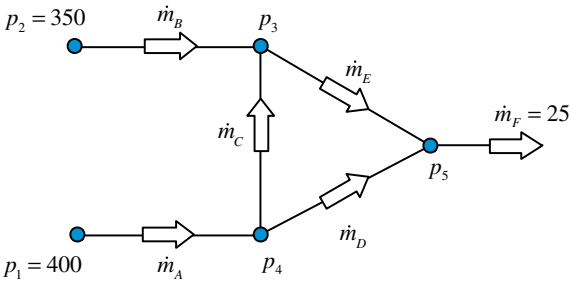


Fig. 16.4 A portion of a gas supply system

Exercise 2 Consider the flow of an ideal gas in a converging nozzle shown in Fig. 16.5, where each of the control volumes has $\Delta x = 0.5$. The area of the various surfaces are $A_{b_i} = 3$; $A_w = 2.3$; $A_e = 1.6$; $A_{b_o} = 0.9$ with $R = 2078$ and $\gamma = 1.4$.

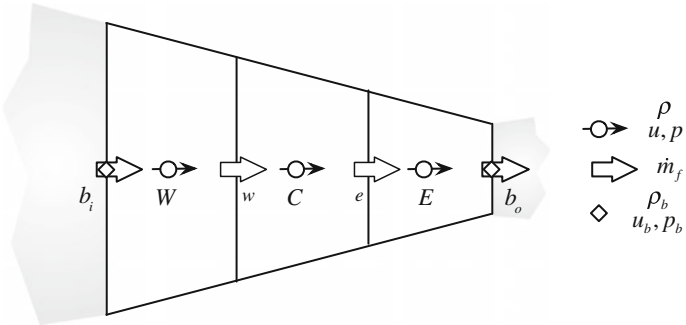
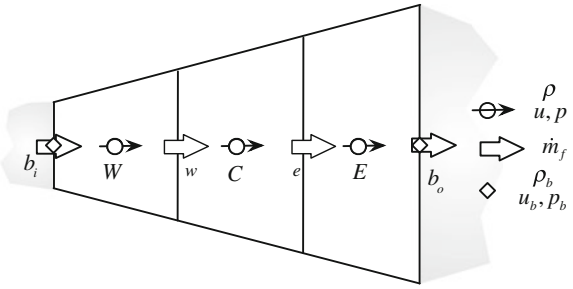


Fig. 16.5 Converging nozzle

The variable area flow is assumed to be one-dimensional and isentropic. At inlet the total pressure is $p_{o,i} = 100701.8\,\text{Pa}$ and the stagnation temperature is $T_{o,i} = 303\,\text{K}$. At exit the static pressure is $p_e = 10^5\,\text{kPa}$. Setup the momentum and pressure correction equations for the three control volumes and obtain the values of the velocity, pressure, density, and temperature starting with uniform fields of values $M = 0.1$ (M is the Mach number), $T = 290\,\text{K}$, and $p = 10^5\,\text{Pa}$. Perform three iterations. Note that there is no need to solve the energy equation as the temperature field can be extracted from the constant stagnation temperature condition.

Exercise 3 Consider the flow of an ideal gas in a diverging nozzle shown in Fig. 16.6, where each of the control volumes has $\Delta x = 0.5$. The area of the various surfaces are $A_{b_i} = 0.9$; $A_w = 1.6$; $A_e = 2.3$; $A_{b_o} = 3$ with $R = 2078$ and $\gamma = 1.4$.

Fig. 16.6 Diverging nozzle



The variable area flow is assumed to be one-dimensional and isentropic. At inlet the total pressure is $p_i = 1bar$, the Mach number is $M_i = 1.2$, and the temperature is $T_i = 303\,\mathrm{K}$. At exit the flow remains supersonic. Setup the momentum and pressure correction equations for the three control volumes and obtain the values of the velocity, pressure, density, and temperature starting with uniform fields of values $M = 1.2$, $T = 303\,\mathrm{K}$, and $p = 1bar$. Perform three iterations. Note that there is no need to solve the energy equation as the temperature field can be extracted from the constant stagnation temperature condition.

Exercise 4 (OpenFOAM®)
Define in the simpleFoamCompressible solver a new variable for the local Mach number to be visualized during simulation.

Exercise 5 (OpenFOAM®)
Modify in the totalPressureCorrectorComp boundary condition, the way Eq. (16.44) is imposed, using now the valueInternalCoeffs() function while resetting the gradientInternalCoeffs(). (Hint: consult Chaps. 11 and 19)

Exercise 6 (OpenFOAM®)
Check the rhoSimpleFoam solver that can be found in $FOAM_SRC/../applications/solvers/compressible/rhoSimpleFoam/pEqn.C and compare it with the algorithm described in this chapter.

Exercise 7 (OpenFOAM®)
Develop a compressible PISO algorithm and implement it starting with the simpleFoamCompressible code described in this chapter.

References

1. Harlow FH, Amsden AA (1968) Numerical calculation of almost incompressible flow. J Comput Phys 3:80–93
2. Harlow FH, Amsden AA (1971) A numerical fluid dynamics calculation method for all flow speeds. J Comput Phys 8(2):197–213

3. Patankar SV (1971) Calculation of unsteady compressible flows involving shocks. Mechanical Engineering Department, Imperial College, Report UF/TN/A/4
4. Issa RI, Lockwood FC (1977) On the prediction of two-dimensional supersonic viscous interactions near walls. AIAA J 15:182–188
5. Hah C (1984) A Navier-Stokes analysis of three-dimensional turbulent flows inside turbine blade rows at design and off-design conditions. ASME J Eng Gas Turbines Power 106:421–429
6. Hah C (1986) Navier-Stokes calculation of three-dimensional compressible flow across a cascade of airfoils with an implicit relaxation method. AIAA Paper 86-0555
7. Issa RI, Gosman D, Watkins A (1986) The computation of compressible and incompressible recirculating flows by a non-iterative implicit scheme. J Comput Phys 62:66–82
8. Karki KC (1986) A calculation procedure for viscous flows at all speeds in complex geometries. Ph.D. thesis, Department of Mechanical Engineering, University of Minnesota
9. Van Doormaal JP, Turan A, Raithby GD (1987) Evaluation of new techniques for the calculations of internal recirculating flows. AIAA Paper 87-0057
10. Rhie CM, Stowers ST (1987) Navier-Stokes analysis for high speed flows using a pressure correction algorithm. AIAA Paper 87-1980
11. Van Doormaal JP (1985) Numerical methods for the solution of incompressible and compressible fluid flows. Ph.D. Thesis, University of Waterloo, Ontario, Canada
12. Van Doormaal JP, Raithby GF, McDonald BD (1987) The segregated approach to predicting viscous compressible fluid flows. ASME J Turbomach 109:268–277
13. Karki KC, Patankar SV (1989) Pressure based calculation procedure for viscous flows at all speeds in arbitrary configurations. AIAA J 27:1167–1174
14. Demirdzic I, Issa RI, Lilek Z (1990) Solution method for viscous flows at all speeds in complex domains. In Wesseling P (ed) Notes on numerical fluid mechanics, vol 29, Vieweg, Braunschweig
15. Demirdzic I, Lilek Z, Peric M (1993) A collocated finite volume method for predicting flows at all speeds. Int J Numer Meth Fluids 16:1029–1050
16. Moukalled F, Darwish M (2000) A unified formulation of the segregated class of algorithms for fluid flow at all speeds. Numer Heat Transf Part B: Fundam 37:103–139
17. Rhie CM (1986) A pressure based Navier-Stokes solver using the multigrid method. AIAA paper 86-0207
18. Moukalled F, Darwish M (2001) A high resolution pressure-based algorithm for fluid flow at all-speeds. J Comput Phys 169(1):101–133
19. Turkel IE (1987) Preconditioning methods for solving the incompressible and low speed compressible equations. J Comput Phys 72:277–298
20. Turkel IE, Vatsa VN, Radespiel R (1996) Preconditioning methods for low speed flows. AIAA Paper 96-2460, Washington
21. Merkle CL, Sullivan JY, Buelow PEO, Venkateswaran S (1998) Computation of flows with arbitrary equations of state. AIAA J 36(4):515–521
22. Moukalled F, Darwish M (2006) Pressure based algorithms for single-fluid and multifluid flows. In: Minkowycz WJ, Sparrow EM, Murthy JY (eds) Handbook of numerical heat transfer, 2nd edn. Wiley, pp 325–367
23. Patankar SV (1980) Numerical heat transfer and fluid flow. Hemisphere, NewYork
24. Nerinckx K, Vierendeels J, Dick E (2005) A pressure-correction algorithm with mach-uniform efficiency and accuracy. Int J Numer Meth Fluids 47:1205–1211
25. Nerinckx K, Vierendeels J, Dick E (2006) A mach-uniform pressure-correction algorithm with AUSM + Flux definitions. Int J Numer Meth Heat Fluid Flow 16(6):718–739
26. Karimian SMH, Schneider GE (1995) Pressure-based control-volume finite element method for flow at all speeds. AIAA J 33(9):1611–1618
27. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org

Part IV
Applications

Chapter 17
Turbulence Modeling

Abstract This chapter addresses some of the challenges that arise when solving turbulent flow problems. It is not intended to provide a comprehensive account on turbulence modeling, rather, the intention is simply to introduce the subject and focus on the implementation details of some of the most popular turbulence models. The presentation is limited to incompressible turbulent fluid flow and begins with a general introduction to turbulence modeling. Then the Reynolds stress tensor that originates from the adopted averaging procedure and the Boussinesq hypothesis used in modeling the Reynolds stresses are presented. This is followed by a review of the $k-\varepsilon$ and $k-\omega$ two-equation models. These are the most popular of the high Reynolds number and low Reynolds number turbulence models, respectively. The BSL and SST models are then introduced, both are derived by combining the $k-\varepsilon$ and $k-\omega$ models so as to address their respective weaknesses. Finally the treatment of the near wall region is presented in detail.

17.1 Turbulence Modeling

In deriving the Navier-Stokes equation in Chap. 3 no mention was made to whether the flow is laminar or turbulent. Whereas laminar flows are stable, turbulent flows are chaotic, diffusive causing rapid mixing, time-dependent, and involve three-dimensional vorticity fluctuations with a broad range of time and length scales [1]. Turbulence typically develops as an instability of laminar flows appearing at a certain critical Reynolds number. In the fluid, these instabilities are caused by the amplification of the perturbation due to the highly non-linear inertial terms.

The most accepted theory of turbulence is based on the "energy cascade" concept developed by Kolmogorov [2, 3]. According to this theory, turbulence is composed of eddies of different sizes with each one possessing a certain amount of energy that depends on its dimension. The larger eddies break up transferring their energy to smaller size eddies in a chain process by which the smaller newly formed eddies undergo similar breakup processes and transfer their energy to even smaller

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_17

eddies. This break up process continues until the smallest possible eddy size is reached. The smallest eddies are of scales at which the molecular viscosity is very effective at dissipating the turbulent kinetic energy as heat.

The smallest turbulent eddies are characterized by the Kolmogorov micro length (η) and time (t_η) scales given by

$$\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{\frac{1}{4}} \quad t_\eta = \left(\frac{\nu}{\varepsilon}\right)^{\frac{1}{2}} \tag{17.1}$$

where ν is the molecular kinematic viscosity and ε the average rate of dissipation of turbulent kinetic energy that will be defined later. In addition, the size of the largest eddies, which is also known by the integral length scale, is defined as being proportional to the size of the geometry involved.

Based on the energy cascade concept a direct numerical solution of the Navier-Stokes equation for turbulent flows necessitates the use of a very small time step limited by a Courant number below 1 and a fine mesh ($\Delta x < \eta$) resulting in a large number of grid points (proportional to Re^3) to resolve the entire spectrum of temporal and spatial turbulent scales involved. This computationally demanding approach, which is denoted in the literature by Direct Numerical Simulation (DNS), has been used by few workers [4–7] in a limited number of simple studies. Due to its prohibitive computational cost the DNS approach cannot currently be employed to solve industrial problems. Future advances in computer technology may change the situation in favor of the DNS.

To reduce the large computational cost associated with a direct solution of the Navier-Stokes equation, statistical analyses can be used to simplify the resolution of turbulent flows. The time-dependent nature of turbulence together with its wide range of time scales suggest that statistical averaging techniques can be applied to approximate random fluctuations. Time averaging, however, leads to correlations that are not known a priori arising from the nonlinear terms in the equations of motion. Modeling these transpiring correlations constitutes the classic closure problem of turbulence modeling.

Following the statistical approach, workers have devised methods that are computationally less intensive than the DNS. One such method is the large eddy simulation (LES) [8–11], in which large scale turbulent structures are directly simulated whereas small turbulent scales are modeled using sub-grid scale models.

The key concept in the LES is to filter the Navier-Stokes equation to determine which scales to keep and which scales to discard. This is done by applying a spatial statistical filter of the form

$$\langle \mathbf{v}(\mathbf{x}, t) \rangle = \iiint F(\mathbf{x} - \lambda : \Delta)\mathbf{v}(\mathbf{x}, t) d^3\lambda \tag{17.2}$$

where the filter function retains values of $\mathbf{v}$ occurring on scales larger than the filter width Δ. The filter function F, is basically some function which is effectively zero

for values of **v** occurring at small scales. Here the symbol $\langle\ \rangle$ indicates a filtered variable.

Thus in the LES approach, turbulent structures of scales larger than the set minimum filter are not filtered. This has the advantage of directly resolving the larger eddies with the higher content of energy (anisotropic turbulence) while the smallest eddies are simply modeled. Accurate modeling of these eddies is possible since at the smaller scale (below the filter width) turbulence can be considered isotropic and independent of the flow type and boundary conditions. With the recent advances in computer technology the use of LES to solve industrial problems is gaining acceptance.

Nevertheless, currently the most popular approach for tackling industrial turbulent flow problems is the one based on solving the Reynolds Averaged Navier-Stokes (RANS) equations [12] where the statistical averaging is now based not on spatial averaging but on a proper time. The key approach is to decompose the flow variables into a time-mean value component and a fluctuating one (Fig. 17.1), substituting in the original equations, and time-averaging the obtained equations. Even though the name refers to the Navier-Stokes equation, the decomposition is applied to all governing equations. Two tracks for averaging the equations have been followed. The standard Reynolds averaging, which is used to derive the Reynolds-Averaged Navier-Stokes (RANS) equations [12], and the mass-weighted or Favre averaging technique employed with turbulent compressible flows and leading to the Favre-Averaged Navier-Stokes (FANS) equations [13]. Following either path, the intention is to model all scales of turbulent flow. Therefore with this approach the mesh size limitation is not as constraining as in the DNS and LES approaches.

In what follows the development of the RANS equations for an incompressible flow is presented.

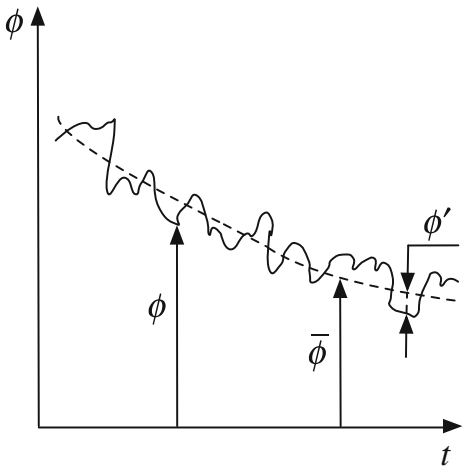


Fig. 17.1 Fluctuating and mean variable components

17.2 Reynolds Averaging

Let ϕ represents at time t and position $\mathbf{x}$ the instantaneous value of any of the flow variables involved ($\mathbf{v}$, p, e, h, T, ρ, etc.). Then, as shown in Fig. 17.1, it is decomposed into a mean value component $\overline{\phi}(\mathbf{x}, t)$ and a fluctuating component $\phi'(\mathbf{x}, t)$ such that

$$\phi(\mathbf{x}, t) = \overline{\phi}(\mathbf{x}, t) + \phi'(\mathbf{x}, t) \tag{17.3}$$

the mean value $\overline{\phi}$ is computed by any of the three Reynolds averaging techniques [12] presented below, of which the time averaging is the most widely used.

17.2.1 Time Averaging

Time averaging represents the average of a quantity over a time interval and is suitable for steady turbulent flows, that is flows which, on average, do not vary with time. If T is the interval over which averaging is performed, then $\overline{\phi}$, which depends only on location, is computed as

$$\overline{\phi}(\mathbf{x}) = \lim_{T\to\infty} \frac{1}{T} \int_{t}^{t+T} \phi(\mathbf{x}, t) dt \tag{17.4}$$

If $\overline{\phi}$ varies slowly with time in comparison with the time scale of turbulent fluctuations, the above equation is replaced by

$$\overline{\phi}(\mathbf{x}, t) = \frac{1}{T} \int_{t}^{t+T} \phi(\mathbf{x}, t) dt \tag{17.5}$$

17.2.2 Spatial Averaging

Spatial averaging represents the average of a quantity over a space interval or a volume V and is suitable for homogeneous turbulence. In this case $\overline{\phi}$, which only depends on time, is computed as

$$\overline{\phi}(t) = \lim_{V\to\infty} \frac{1}{V} \int_{V} \phi(\mathbf{x}, t) dV \tag{17.6}$$

17.2.3 Ensemble Averaging

Ensemble averaging, which is suitable for any type of turbulent flows including unsteady turbulent flows, represents the average of many identical quantities at a certain time. If the number of identical quantities is designated by N, then $\overline{\phi}$, which in this case is a function of space and time, is given by

$$\overline{\phi}(\mathbf{x},t) = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x},t) \tag{17.7}$$

17.2.4 Averaging Rules

If ϕ and φ are two variables and ϕ' and φ' are their fluctuating components, then some of the averaging rules needed in deriving the RANS equations include the following:

$$\begin{aligned}
\overline{\phi'} &= 0 \\
\overline{\phi} &= \overline{\overline{\phi}} \\
\overline{\nabla\phi} &= \nabla\overline{\phi} \\
\overline{\phi+\varphi} &= \overline{\phi} + \overline{\varphi} \\
\overline{\overline{\phi}\varphi} &= \overline{\phi}\,\overline{\varphi} \\
\overline{\overline{\phi}\varphi'} &= 0 \\
\overline{\phi\varphi} &= \overline{\phi}\,\overline{\varphi} + \overline{\phi'\varphi'}
\end{aligned} \tag{17.8}$$

17.2.5 Incompressible RANS Equations

The incompressible Reynolds-averaged Navier–Stokes equations are based on time-averaged variables. Decomposing the velocity, pressure, and temperature fields into

$$\begin{aligned}
\mathbf{v} &= \overline{\mathbf{v}} + \mathbf{v}' \\
p &= \overline{p} + p' \\
T &= \overline{T} + T' \\
\overline{\mathbf{v}} &= \overline{u}\mathbf{i} + \overline{v}\mathbf{j} + \overline{w}\mathbf{k} \\
\mathbf{v}' &= u'\mathbf{i} + v'\mathbf{j} + w'\mathbf{k}
\end{aligned} \tag{17.9}$$

and substituting $\mathbf{v}$, p, and T by their decomposed expressions in the incompressible continuity, momentum, and energy equations given by Eqs. (3.13), (3.39), and (3.78),

respectively, assuming a Newtonian fluid, and taking the time average, these equations are transformed to

$$\overline{\nabla \cdot [\rho(\overline{\mathbf{v}} + \mathbf{v}')] = 0} \tag{17.10}$$

$$\overline{\frac{\partial}{\partial t}[\rho(\overline{\mathbf{v}} + \mathbf{v}')] + \nabla \cdot \{\rho(\overline{\mathbf{v}} + \mathbf{v}')(\overline{\mathbf{v}} + \mathbf{v}')\} = -\nabla(\overline{p} + p') + \nabla \cdot \left\{\mu\left[\nabla(\overline{\mathbf{v}} + \mathbf{v}') + (\nabla(\overline{\mathbf{v}} + \mathbf{v}'))^{\mathrm{T}}\right]\right\} + \mathbf{f}_b} \tag{17.11}$$

$$\overline{\frac{\partial}{\partial t}[\rho c_p(\overline{T} + T')] + \nabla \cdot \left[\rho c_p(\overline{\mathbf{v}} + \mathbf{v}')(\overline{T} + T')\right] = \nabla \cdot \left[k\nabla(\overline{T} + T')\right] + S^T} \tag{17.12}$$

The Reynolds averaged forms of Eqs. (17.10)–(17.12) are obtained as

$$\nabla \cdot [\rho\overline{\mathbf{v}}] = 0 \tag{17.13}$$

$$\frac{\partial}{\partial t}[\rho\overline{\mathbf{v}}] + \nabla \cdot \{\rho\overline{\mathbf{v}}\overline{\mathbf{v}}\} = -\nabla\overline{p} + \left[\nabla \cdot \left(\overline{\boldsymbol{\tau}} - \rho\overline{\mathbf{v}'\mathbf{v}'}\right)\right] + \overline{\mathbf{f}}_b \tag{17.14}$$

$$\frac{\partial}{\partial t}\left[\rho c_p\overline{T}\right] + \nabla \cdot \left[\rho c_p\overline{\mathbf{v}}\overline{T}\right] = \nabla \cdot \left[k\nabla\overline{T} - \rho c_p\overline{\mathbf{v}'T'}\right] + \overline{S^T} \tag{17.15}$$

The above Reynolds averaged equations are similar to the original conservation equations with the exception of the additional averaged products of the fluctuating components due to the non-linear terms. This introduces six new unknowns (the components of the tensor $-\rho\overline{\mathbf{v}'\mathbf{v}'}$, known as the Reynolds stress tensor $\boldsymbol{\tau}^R$) to the momentum equations and three new unknown turbulent heat fluxes $\left(\dot{\mathbf{q}}^R = -\rho c_p\overline{\mathbf{v}'T'}\right)$ to the energy equation. The expanded forms of the Reynolds stress tensor $\boldsymbol{\tau}^R$ and turbulent heat flux vector are given by

$$\boldsymbol{\tau}^R = -\rho\begin{pmatrix} \overline{u'u'} & \overline{u'v'} & \overline{u'w'} \\ \overline{u'v'} & \overline{v'v'} & \overline{v'w'} \\ \overline{u'w'} & \overline{v'w'} & \overline{w'w'} \end{pmatrix} \quad \dot{\mathbf{q}}^R = -\rho c_p\begin{bmatrix} \overline{u'T'} \\ \overline{v'T'} \\ \overline{w'T'} \end{bmatrix} \tag{17.16}$$

Consequently the set of RANS equations is not a closed set and to be able to solve it additional equations for the unknown Reynolds stress components are required. The process of calculating these Reynolds stresses is denoted in the literature by turbulence modeling. Attempting to develop such equations by using the original conservation equations results in additional unknowns (such as triple products of the fluctuating components) complicating the problem further. The Reynolds stress tensor comes from the non-linear convection term of both momentum and temperature confirming that the turbulence itself is the effect of a non-linear phenomena highly sensitive to any perturbation. Therefore any linear averaging of the equations, like the Reynolds averaging techniques, cannot reduce

the order of the problem. The complexity is actually recursive in that trying to develop additional equations for the triple products, quadruple products arise and so on. To overcome this problem, any turbulence model has to close the system of equations by expressing the non-linear fluctuating stress components only in terms of the mean components as described next.

17.3 Boussinesq Hypothesis

The direct modeling of the Reynolds stress tensor is based on the Boussinesq hypothesis [14–16], which in analogy with Newtonian flows assumes the Reynolds stress to be a linear function of the mean velocity gradients such that

$$\boldsymbol{\tau}^R = -\rho\overline{\mathbf{v}'\mathbf{v}'} = \mu_t\left\{\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right\} - \frac{2}{3}[\rho k + \mu_t(\nabla\cdot\mathbf{v})]\mathbf{I} \tag{17.17}$$

which for incompressible flows reduces to

$$\boldsymbol{\tau}^R = -\rho\overline{\mathbf{v}'\mathbf{v}'} = \mu_t\left\{\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right\} - \frac{2}{3}\rho k\mathbf{I} \tag{17.18}$$

where from now on the over bar is dropped from the averaged quantities to simplify the notation, k is the turbulent kinetic energy defined as

$$k = \frac{1}{2}\overline{\mathbf{v}'\cdot\mathbf{v}'} \tag{17.19}$$

and μ_t the turbulent eddy viscosity (in analogy with molecular viscosity), which is now flow, not fluid, dependent. With this approximation, the problem of calculating the Reynolds stress components is transformed into computing the turbulent kinetic energy and turbulent viscosity. For incompressible flows the term $-(2/3)\rho k\mathbf{I}$ in the Reynolds stress is usually combined with the pressure gradient term by defining a turbulent pressure p as

$$p \leftarrow p + \frac{2}{3}\rho k \tag{17.20}$$

thereby reducing the unknowns to μ_t alone, which is evaluated using a variety of turbulence models.

In a similar way, the turbulent thermal fluxes are calculated in analogy with Fourier's law such that

$$\dot{\mathbf{q}}^R = -\rho c_p\overline{\mathbf{v}'T'} = k_t\nabla T \tag{17.21}$$

where k_t is the turbulent thermal diffusivity calculated as will be explained later.

17.4 Turbulence Models

Several turbulence models based on the Boussinesq hypothesis have been developed to express the turbulent viscosity, μ_t, in terms of a velocity $(\sqrt{k})$ and length (ℓ) scales such that

$$\mu_t = \rho \ell \sqrt{k} \tag{17.22}$$

These models are grouped, into four main categories:

- Algebraic (Zero-Equation) Models
- One-Equation Models
- Two-Equation Models
- Second-Order Closure Models

None of the developed models is universally applicable to all flow conditions. Though each group has certain advantages and strengths.

The zero-equation models [17–19] use an algebraic equation to compute μ_t without the need to solve any differential equation. The one-equation models [20–22] require solving only one transport differential equation to compute the turbulent eddy viscosity. Two-equation turbulence models [23–35] necessitate the solution of two transport equations for the calculation of μ_t. The second-order closure models [36–41] are the most computationally expensive as separate transport equations are solved for the individual turbulent fluxes (6 equations).

The two-equation turbulence models are the most popular in terms of usage in the simulation of industrial applications, requiring the solution of two transport equations while delivering accurate enough predictions. The $k-\varepsilon$ model of Jones and Launder [23] was amongst the earliest two equation models and the most popular, while the $k-\omega$ model of Wilcox [28, 29] comes as a close second. Both models had undergone many modifications and improvements [26, 30], which have greatly extended their applicability.

17.5 Two-Equation Turbulence Models

17.5.1 Standard $k-\varepsilon$ Model

The well-known $k-\varepsilon$ model of Jones and Launder [23], known as the standard $k-\varepsilon$ model, is based on the Boussinesq approximation with the turbulent viscosity μ_t and thermal diffusivity k_t formulated as

$$\begin{aligned} \mu_t &= \rho C_\mu \frac{k^2}{\varepsilon} \\ k_t &= \frac{c_p \mu_t}{Pr_t} \end{aligned} \tag{17.23}$$

where ε is the rate of dissipation of turbulence kinetic energy per unit mass due to viscous stresses given by

$$\varepsilon = \frac{1}{2}\frac{\mu}{\rho}\overline{\left\{\nabla \mathbf{v}' + (\nabla \mathbf{v}')^T\right\} : \left\{\nabla \mathbf{v}' + (\nabla \mathbf{v}')^T\right\}} \tag{17.24}$$

In the model, the turbulent kinetic energy k and the turbulent energy dissipation rate ε are computed using

$$\frac{\partial}{\partial t}(\rho k) + \nabla \cdot (\rho \mathbf{v} k) = \nabla \cdot (\mu_{eff,k} \nabla k) + \underbrace{P_k - \rho\varepsilon}_{S^k} \tag{17.25}$$

$$\frac{\partial}{\partial t}(\rho \varepsilon) + \nabla \cdot (\rho \mathbf{v} \varepsilon) = \nabla \cdot (\mu_{eff,\varepsilon} \nabla \varepsilon) + \underbrace{C_{\varepsilon 1}\frac{\varepsilon}{k}P_k - C_{\varepsilon 2}\rho\frac{\varepsilon^2}{k}}_{S^\varepsilon} \tag{17.26}$$

where

$$\mu_{eff,k} = \mu + \frac{\mu_t}{\sigma_k} \quad \mu_{eff,\varepsilon} = \mu + \frac{\mu_t}{\sigma_\varepsilon} \tag{17.27}$$

with the turbulent Prandtl number (Pr_t) and other model constants assigned the following values: $C_{\varepsilon 1} = 1.44$, $C_{\varepsilon 2} = 1.92$, $C_\mu = 0.09$, $\sigma_k = 1.0$, $\sigma_\varepsilon = 1.3$ and $Pr_t = 0.9$.

The compact form of the production of turbulent energy term is given by

$$P_k = \boldsymbol{\tau}^R : \nabla \mathbf{v} \tag{17.28}$$

while its expanded form for incompressible flow can be obtained by multiplying Eq. (3.75) by μ_t.

In the derivation of the standard $k - \varepsilon$ model the flow is assumed to be fully turbulent and the effects of molecular viscosity to be negligible. Therefore the standard $k - \varepsilon$ model is a high Reynolds number turbulence model valid only for fully turbulent free shear flows that cannot be integrated all the way to the wall.

Modeling flows close to solid walls requires integration of the two equations over a fine grid in order to correctly capture the turbulent quantities inside the boundary layer as well as the corrections for low Reynolds number effects. A turbulence model that can be integrated all the way to the wall is denoted in the literature by a low Reynolds number turbulence model or a low Reynolds number version. Several so called low Reynolds number $k - \varepsilon$ models have been proposed over the years (see Patel et al. [42] and Wilcox [29] for a review). The idea behind their development is to damp the turbulent viscosity near the wall through the use of a damping function that tends towards zero as the distance to the wall decreases, i.e., as the wall is approached. Constants multiplying source terms in the turbulent dissipation equation are in some models also damped. All models share the same basic structure differing in the tuning of the damping functions and in some extra sources in the dissipation equation.

The only exception to this rule is the $k-\omega$ turbulence model of Wilcox [28, 29], which can be integrated all the way to the wall without the need to employ damping functions. Still the model can also be used as a high Reynolds number model.

One of the main drawback of the two-equation models is the so called stagnation point anomaly. In high strain regions the two-equation models tend to over predict the turbulence kinetic energy production P_k. This was originally recognized in stagnation point flows, but it is a more widespread anomaly. The problem is an overproduction of the turbulence kinetic energy for the case when a moderate level of k is subjected to a large rate of strain. This may be attributed to an underestimation of the sink term and/or an overestimation of the turbulent viscosity. These ideas could be merged together into a bound on the local turbulent time scale $t_s=k/\varepsilon$ (Medic and Durbin [43]). The first step is to reformulate the expression for the turbulent viscosity as

$$\mu_t = \rho C_\mu k t_s \tag{17.29}$$

Then using t_s, the ε equation is modified to

$$\frac{\partial}{\partial t}(\rho\varepsilon) + \nabla\cdot(\rho\mathbf{v}\varepsilon) = \nabla\cdot(\mu_{eff,\varepsilon}\nabla\varepsilon) + C_{\varepsilon 1}\frac{1}{t_s}P_k - C_{\varepsilon 2}\rho\frac{\varepsilon}{t_s} \tag{17.30}$$

Finally, to constrain the Reynolds stress tensor to be positive definite, a limiter is applied on t_s such that

$$\begin{aligned} t_s &= \min\left[\frac{k}{\varepsilon}, \frac{\alpha}{\sqrt{6}C_\mu S_t}\right], \\ \alpha &= 0.6 \\ S_t &= \sqrt{\mathbf{S}_t\cdot\mathbf{S}_t} \\ \mathbf{S}_t &= \frac{1}{2}\left(\nabla\mathbf{v} + \nabla\mathbf{v}^T\right) \end{aligned} \tag{17.31}$$

As a consequence, at a large rate of strain, P_k grows at the rate S_t rather than S_t^2.

17.5.2 The $k-\omega$ Model

It has already been mentioned how the family of $k-\varepsilon$ models are well behaved for free-shear flows while are likely to fail in predicting flows with adverse pressure gradient. Another class of models, for which the equation for ε is replaced by an equation for ω, where ω is the rate at which turbulence kinetic energy is converted into internal thermal energy per unit volume and time, is better capable of predicting separated flows.

The first complete turbulence model in this category was proposed by Kolmogorov [27]. In addition to the same equation for k, Kolmogorov developed a

second equation for ω. The reciprocal of ω serves as a local turbulence time scale while the turbulence length scale is given by $\sqrt{k}/\omega$. The $k-\omega$ model introduced next is proposed by Wilcox [29] as an evolution to the well-known $k-\omega$ model that Wilcox reported in [28].

The $k-\omega$ model of Wilcox [29] is similar in structure to the $k-\varepsilon$ model and is also based on the Boussinesq approximation. Two transport equations are solved to determine the two (large) scales of turbulence. The specific turbulence dissipation ω is defined as

$$\omega = \frac{\varepsilon}{C_\mu k} \tag{17.32}$$

The advantages of replacing the ε-equation by the ω-equation are: (i) the second is easier to integrate (more robust), (ii) it can be integrated through the sub-layer without the need for additional damping functions, and (iii) it performs better for flows with weak adverse pressure gradient. The conservation equations are written as

$$\frac{\partial}{\partial t}(\rho k) + \nabla \cdot (\rho \mathbf{v} k) = \nabla \cdot (\mu_{eff,k} \nabla k) + \underbrace{P_k - \beta^* \rho k \omega}_{S^k} \tag{17.33}$$

$$\frac{\partial}{\partial t}(\rho \omega) + \nabla \cdot (\rho \mathbf{v} \omega) = \nabla \cdot (\mu_{eff,\omega} \nabla \omega) + \underbrace{C_{\alpha 1} \frac{\omega}{k} P_k - C_{\beta 1} \rho \omega^2}_{S^\omega} \tag{17.34}$$

with the model constants assigned the values

$$C_{\alpha 1} = 5/9, C_{\beta 1} = 0.075, \beta^* = 0.09, \sigma_{k1} = 2, \sigma_{\omega 1} = 2, Pr_t = 0.9.$$

where

$$\begin{aligned} \mu_t &= \rho \frac{k}{\omega} \\ k_t &= \frac{\mu_t}{Pr_t} \\ \mu_{eff,k} &= \mu + \frac{\mu_t}{\sigma_{k_1}} \\ \mu_{eff,\omega} &= \mu + \frac{\mu_t}{\sigma_{\omega_1}} \end{aligned} \tag{17.35}$$

The major drawback of the Wilcox model is its sensitivity to the free stream [30] specified values, which leads to strong dependence of the solution on the arbitrary specification of the free stream ω. This dependence is not present in the $k-\varepsilon$ model.

17.5.3 The Baseline (BSL) $k-\omega$ Model

The Baseline (BSL) model developed by Menter [32] combines the $k-\varepsilon$ and $k-\omega$ models so as to take advantage of their respective strength, i.e. the robustness of the $k-\omega$ model near wall surfaces due to its simple low Reynolds number formulation and its ability to compute flows with weak adverse pressure gradients accurately, and the better performance of the $k-\varepsilon$ model near the boundary layer edge and away from walls, due to its insensitivity to the free stream values. The basis of this technique is the transformation of the $k-\varepsilon$ model to a $k-\omega$ formulation. This is an exact conversion, except for small contributions from the diffusion term due to the difference in the diffusion coefficients of the k and ε equations. The $k-\omega$ formulation of the $k-\varepsilon$ model is given by

$$\frac{\partial}{\partial t}(\rho k)+\nabla\cdot(\rho\mathbf{v}k)=\nabla\cdot(\mu_{eff,k}\nabla k)+\underbrace{P_k-\beta^*\rho k\omega}_{S^k} \tag{17.36}$$

$$\frac{\partial}{\partial t}(\rho\omega)+\nabla\cdot(\rho\mathbf{v}\omega)=\nabla\cdot(\mu_{eff,\omega}\nabla\omega)+\underbrace{C_{\alpha 2}\frac{\omega}{k}P_k-C_{\beta 2}\rho\omega^2+2\sigma_{\omega 2}\frac{\rho}{\omega}\nabla k\cdot\nabla\omega}_{S^\omega} \tag{17.37}$$

The differences between this formulation and the original $k-\omega$ model are in the additional cross-diffusion term appearing in the equation for ω and in the modeling constants that are given by

$$C_{\alpha 2}=0.4404, C_{\beta 2}=0.0828, \sigma_{k2}=1.0, \sigma_{\omega 2}=0.856 \text{ and } Pr_t=0.9.$$

The BSL $k-\omega$ model is derived by multiplying the $k-\omega$ (Eqs. 17.33 and 17.34) with a blending function F_1 and the $k-\omega$ formulation of the $k-\varepsilon$ model equations (Eqs. 17.36 and 17.37) by $(1-F_1)$, yielding the following equations for k and ω [32]:

$$\frac{\partial}{\partial t}(\rho k)+\nabla\cdot(\rho\mathbf{v}k)=\nabla\cdot(\mu_{eff,k}\nabla k)+\underbrace{P_k-\beta^*\rho k\omega}_{S^k} \tag{17.38}$$

$$\frac{\partial}{\partial t}(\rho\omega)+\nabla\cdot(\rho\mathbf{v}\omega)=\nabla\cdot(\mu_{eff,\omega}\nabla\omega)+\underbrace{\tilde{C}_\alpha\frac{\omega}{k}P_k-\tilde{C}_\beta\rho\omega^2+2(1-F_1)\sigma_{\omega 2}\frac{\rho}{\omega}\nabla k\cdot\nabla\omega}_{S^\omega} \tag{17.39}$$

These equations are formally very similar to those of the standard $k-\omega$ model, however all their coefficients depend on the blending function F_1, in the form

$$\tilde{\Phi} = F_1\Phi_1 + (1 - F_1)\Phi_2 \tag{17.40}$$

with the constants of the original $k - \omega$ model used in Eq. (17.39) given by

$$C_{\alpha 1} = 0.5976, C_{\beta 1} = 0.075, \beta^* = 0.09, \sigma_{k1} = 2, \sigma_{\omega 1} = 2, Pr_t = 0.9.$$

The blending function F_1 depends on the solution variables and on the distance $d_\perp$ from the nearest wall and is given as

$$F_1 = \tanh(\gamma_1^4) \tag{17.41}$$

where

$$\begin{aligned} \gamma_1 &= \text{Min}\left(\text{Max}\left(\frac{\sqrt{k}}{\beta^*\omega(d_\perp)}, \frac{500\nu}{(d_\perp)^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}(d_\perp)^2}\right) \\ CD_{k\omega} &= \text{Max}\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\nabla k \cdot \nabla\omega, 10^{-10}\right) \end{aligned} \tag{17.42}$$

and

$$\begin{aligned} \mu_t &= \rho\frac{k}{\omega} \\ k_t &= \frac{\mu_t}{Pr_t} \\ \mu_{eff,k} &= \mu + \frac{\mu_t}{\tilde{\sigma}_k} \\ \mu_{eff,\omega} &= \mu + \frac{\mu_t}{\tilde{\sigma}_\omega} \end{aligned} \tag{17.43}$$

The BSL model has a similar performance as the $k - \omega$ model for boundary layer flows and is nearly identical to the $k - \varepsilon$ model for free shear flows. Its robustness is close to that of the $k - \omega$ model.

17.5.4 The Shear Stress Transport (SST) k − ω Model

Further modifications to the BSL model yields the Shear Stress Transport (SST) [32–35] model which, when compared to other eddy-viscosity models, has an improved adverse pressure gradient performance. The first modification is related to satisfying Bradshaw's assumption, which states that the principal shear stress and the turbulent kinetic energy in the boundary layer are linearly related via an equation of the form

$$\tau_{xy} = \rho a_1 k \tag{17.44}$$

On the other hand, the principal shear stress for conventional two-equation turbulence models can be computed as

$$\tau_{xy} = \mu_t \Omega = \rho \sqrt{\frac{\text{Production of } k}{\text{Dissipation of } k}} a_1 k \tag{17.45}$$

where Ω is the vorticity. In flows with adverse pressure gradient the ratio of production of turbulent kinetic energy to its dissipation rate could be much larger than one, thereby substantially violating Bradshaw's hypothesis. For Eq. (17.44) to be satisfied within the framework of eddy-viscosity models, Menter [35] modified the turbulent viscosity μ_t in the SST $k-\omega$ model by bounding it according to

$$\mu_t = \frac{\rho a_1 k}{\text{Max}(a_1 \omega, \sqrt{2} S_t F_2)} \tag{17.46}$$

where $a_1 = 0.31$, S_t is the magnitude of the strain rate defined in Eq. (17.31), and F_2 is given by [35]

$$F_2 = \tanh(\gamma_2^2) \text{ with } \gamma_2 = \text{Max}\left(2\frac{\sqrt{k}}{\beta^* \omega (d_\perp)}, \frac{500\nu}{(d_\perp)^2 \omega}\right) \tag{17.47}$$

Moreover, to maintain the original formulation of the eddy-viscosity for free shear layers, the same blending function approach as for the baseline model is also adopted in the SST $k-\omega$ model. The k and ω equations are given by Eq. (17.38) and Eq. (17.39), respectively.

The second modification is related to the production of turbulence kinetic energy P_k in the k equation (Eq. 17.38), which is replaced by $\tilde{P}_k$ given by

$$\tilde{P}_k = \min(P_k, c_1 \varepsilon) \tag{17.48}$$

where ε is obtained from Eq. (17.32) and the blending function F_1 is calculated as in the BSL model via Eqs. (17.41) and (17.42) with the coefficients computed by Eq. (17.40) using the following model constants:

$$C_{\alpha 1} = 0.5532, C_{\beta 1} = 0.075, \beta^* = 0.09, \sigma_{k1} = 2, \sigma_{\omega 1} = 2, c_1 = 10.$$
$$C_{\alpha 2} = 0.4403, C_{\beta 2} = 0.0828, \sigma_{k2} = 1.0, \sigma_{\omega 2} = 1.186, \text{ and } Pr_t = 0.9.$$

In addition, the turbulent thermal conductivity and effective turbulent viscosities for k and ω are computed as

$$\begin{aligned} k_t &= \frac{\mu_t}{Pr_t} \\ \mu_{eff,k} &= \mu + \frac{\mu_t}{\tilde{\sigma}_k} \\ \mu_{eff,\omega} &= \mu + \frac{\mu_t}{\tilde{\sigma}_\omega} \end{aligned} \tag{17.49}$$

17.6 Summary of Incompressible Turbulent Flow Equations

The incompressible time-averaged continuity, momentum, energy, turbulence kinetic energy, turbulence dissipation rate, and specific dissipation rate equations can be written, respectively, as

$$\nabla \cdot [\rho \mathbf{v}] = 0 \tag{17.50}$$

$$\frac{\partial}{\partial t}[\rho \mathbf{v}] + \nabla \cdot \{\rho \mathbf{v}\mathbf{v}\} = \nabla \cdot \{(\mu + \mu_t)\nabla \mathbf{v}\} + \mathbf{Q}^{\mathbf{v}} \tag{17.51}$$

$$\frac{\partial}{\partial t}(\rho c_p T) + \nabla \cdot [\rho c_p \mathbf{v} T] = \nabla \cdot \left[c_p \left(\frac{\mu}{Pr} + \frac{\mu_t}{Pr_t} \right) \nabla T \right] + Q^T \tag{17.52}$$

$$\frac{\partial}{\partial t}(\rho k) + \nabla \cdot [\rho \mathbf{v} k] = \nabla \cdot [\mu_{eff,k} \nabla k] + Q^k \tag{17.53}$$

$$\frac{\partial}{\partial t}(\rho \varepsilon) + \nabla \cdot [\rho \mathbf{v} \varepsilon] = \nabla \cdot [\mu_{eff,\varepsilon} \nabla \varepsilon] + Q^\varepsilon \tag{17.54}$$

$$\frac{\partial}{\partial t}(\rho \omega) + \nabla \cdot [\rho \mathbf{v} \omega] = \nabla \cdot [\mu_{eff,\omega} \nabla \omega] + Q^\omega \tag{17.55}$$

All these equations are similar in structure and can be written in the general form given by Eq. (3.93). As such, their discretization follows the general procedures presented in previous chapters. A summary of their discretized forms is given in the next section.

17.7 Discretization of the Turbulent Flow Equations

The discretization of the unsteady, convection, and diffusion terms appearing in the conservation equations for turbulent flows, Eqs. (17.50)–(17.55), follows the procedures described in previous chapters. Moreover the discretization of the momentum and energy equations lead to the same algebraic equations given by Eqs. (15.74) and (16.29), with molecular viscosity and thermal conductivity replaced by $(\mu + \mu_t)$ and $c_p(\mu/Pr + \mu_t/Pr_t)$, respectively. As the flow is incompressible, the terms involving $\nabla \cdot \mathbf{v}$ are set to zero in these equations. Moreover, at low speeds the dissipation terms involving Φ and Ψ in the energy equation being very small are usually neglected. Another difference is related to the implementation of boundary conditions at a wall, which will be explained later. The discretized equations for k, ε and ω are presented below assuming a first order Euler scheme for the discretization of the unsteady term and a high resolution scheme for the discretization of the convection term applied in the context of a deferred correction approach.

17.7.1 The Discretized Form of the k Equation

The final algebraic form of the turbulence kinetic energy equation can be written as

$$a_C^k k_C + \sum_{F \sim NB(C)} a_F^k k_F = b_C^k \tag{17.56}$$

where the coefficients are given by

$$\begin{aligned}
a_F^k &= -(\mu_{eff,k})_f \frac{E_f}{d_{CF}} - ||-\dot{m}_f, 0|| \\
a_C^k &= a_C^{\cdot} - \sum_{F \sim NB(C)} a_F^k + \sum_{f \sim nb(C)} \dot{m}_f + \begin{cases} \rho_C \dfrac{\varepsilon_C}{k_C} V_C & k-\varepsilon \text{ model} \\ \beta^* \rho_C \omega_C V_C & k-\omega \text{ models} \end{cases} \\
a_C^{\cdot} &= \frac{\rho_C V_C}{\Delta t} \\
a_C^{\circ} &= \frac{\rho_C^{\circ} V_C}{\Delta t} \\
b_C^k &= -\sum_{f \sim nb(C)} \dot{m}_f \left(k_f^{HR} - k_f^U \right) + a_C^{\circ} k_C^{\circ} + \begin{cases} (\tilde{P}_k)_C V_C & SST\ k-\omega \text{ model} \\ (P_k)_C V_C & otherwise \end{cases} \\
&\quad + \sum_{f \sim nb(C)} (\mu_{eff,k})_f (\nabla k)_f \cdot \mathbf{T}_f
\end{aligned} \tag{17.57}$$

The discretized form of the production of turbulent kinetic energy P_k is given by Eq. (16.27) with molecular viscosity replaced by turbulent viscosity. Moreover, similar to other variables, under relaxation of the turbulence kinetic energy equation is usually required.

17.7.2 The Discretized Form of the ε Equation

The final algebraic form of the turbulence dissipation rate equation can be written as

$$a_C^{\varepsilon} \varepsilon_C + \sum_{F \sim NB(C)} a_F^{\varepsilon} \varepsilon_F = b_C^{\varepsilon} \tag{17.58}$$

where the coefficients are given by

$$
\begin{aligned}
a_F^{\varepsilon} &= -\left(\mu_{eff,\varepsilon}\right)_f \frac{E_f}{d_{CF}} - \left\|-\dot{m}_f, 0\right\| \\
a_C^{\varepsilon} &= a_C^{\cdot} - \sum_{F\sim NB(C)} a_F^{\varepsilon} + \sum_{f\sim nb(C)} \dot{m}_f + C_{\varepsilon 2}\rho_C \frac{\varepsilon_C}{k_C} V_C \\
a_C^{\cdot} &= \frac{\rho_C V_C}{\Delta t} \\
a_C^{\circ} &= \frac{\rho_C^{\circ} V_C}{\Delta t} \\
b_C^{\varepsilon} &= \sum_{f\sim nb(C)} \left(\mu_{eff,\varepsilon}\right)_f (\nabla\varepsilon)_f \cdot \mathbf{T}_f - \sum_{f\sim nb(C)} \dot{m}_f (\varepsilon_f^{HR} - \varepsilon_f^{U}) + a_C^{\circ}\varepsilon_C^{\circ} + C_{\varepsilon 1}\frac{\varepsilon_C}{k_C}(P_k)_C V_C
\end{aligned}
\tag{17.59}
$$

Moreover, similar to other variables, under relaxation of the turbulence dissipation rate equation is usually required.

17.7.3 The Discretized Form of the ω Equation

The final algebraic form of the specific turbulence dissipation equation can be written as

$$
a_C^{\omega}\omega_C + \sum_{F\sim NB(C)} a_F^{\omega}\omega_F = b_C^{\omega} \tag{17.60}
$$

where the coefficients are given by

$$
\begin{aligned}
a_F^{\omega} &= -(\mu_{eff,\omega})_f \frac{E_f}{d_{CF}} - \|-\dot{m}_f, 0\| \\
a_C^{\omega} &= a_C^{\cdot} - \sum_{F\sim NB(C)} a_F^{\omega} + \sum_{f\sim nb(C)} \dot{m}_f + a_C^{add} \\
a_C^{\cdot} &= \frac{\rho_C V_C}{\Delta t} \\
a_C^{\circ} &= \frac{\rho_C^{\circ} V_C}{\Delta t} \\
b_C^{\omega} &= \sum_{f\sim nb(C)} (\mu_{eff,\omega})_f (\nabla\omega)_f \cdot \mathbf{T}_f - \sum_{f\sim nb(C)} \dot{m}_f (\omega_f^{HR} - \omega_f^{U}) + a_C^{\circ}\omega_C^{\circ} + b_C^{add}
\end{aligned}
\tag{17.61}
$$

The expressions for a_C^{add} and b_C^{add} depend on the formulation of the $k-\omega$ model used and the different expressions are given as follows:

Original formulation of the $k-\omega$ model

$$\begin{aligned} a_C^{add} &= C_{\beta 1}\rho_C\omega_C V_C \\ b_C^{add} &= C_{\alpha 1}\frac{\omega_C}{k_C}(P_k)_C V_C \end{aligned} \tag{17.62}$$

The $k-\omega$ formulation of the $k-\varepsilon$ model

$$\begin{aligned} a_C^{add} &= C_{\beta 1}\rho_C\omega_C V_C + ||-2\sigma_{\omega 2}\frac{\rho_C}{\omega_C^2}(\nabla k\cdot\nabla\omega)_C, 0||V_C \\ b_C^{add} &= C_{\alpha 1}\frac{\omega_C}{k_C}(P_k)_C V_C + ||2\sigma_{\omega 2}\frac{\rho_C}{\omega_C}(\nabla k\cdot\nabla\omega)_C, 0||V_C \end{aligned} \tag{17.63}$$

The BSL and SST formulation of the $k-\omega$ model

$$\begin{aligned} a_C^{add} &= \tilde{\beta}\rho_C\omega_C V_C + ||-2(1-F_1)_C\,\sigma_{\omega 2}\frac{\rho_C}{\omega_C^2}(\nabla k\cdot\nabla\omega)_C, 0||V_C \\ b_C^{add} &= \tilde{\alpha}\frac{\omega_C}{k_C}(P_k)_C V_C + ||2(1-F_1)_C\,\sigma_{\omega 2}\frac{\rho_C}{\omega_C}(\nabla k\cdot\nabla\omega)_C, 0||V_C \end{aligned} \tag{17.64}$$

The discretized form of $(\nabla k\cdot\nabla\omega)_C$ is computed as

$$(\nabla k\cdot\nabla\omega)_C = \left(\frac{\partial k}{\partial x}\right)_C\left(\frac{\partial\omega}{\partial x}\right)_C + \left(\frac{\partial k}{\partial y}\right)_C\left(\frac{\partial\omega}{\partial y}\right)_C + \left(\frac{\partial k}{\partial z}\right)_C\left(\frac{\partial\omega}{\partial z}\right)_C \tag{17.65}$$

Again under relaxation of the specific turbulence dissipation equation is usually required.

17.8 Boundary Conditions

17.8.1 Modeling Flow Near the Wall

As a turbulent flow approaches a wall its mean and fluctuating components of velocity, and consequently k, vanish creating large gradients. In addition, the very high turbulent stresses away from the wall, decrease in the near wall layer to values of magnitude comparable to those of the viscous stresses. Therefore if the near wall layer is to be resolved, a substantial number of grid points will be required.

Low Reynolds number turbulence models are capable of simulating the dampening effects of the wall but at the expense of using a very large number of grid

points. This is the unavoidable cost that has to be payed if accurate solutions of the flow in the near wall region is required.

On the other hand, the high Reynolds number turbulence approach, exemplified by the standard $k - \varepsilon$ model, avoids the need to resolve the near wall layer through the use of wall functions. In this method, theoretical profiles between the boundary surface and the first near-wall node are assumed and superimposed. Compared to the previous approach, wall functions reduce significantly the computational cost. The main disadvantage of this methodology however, is related to the validity of these profiles, which are only known and justified in near-equilibrium boundary layers. Details regarding this special treatment is explained next.

17.8.2 Standard Wall Functions

The wall functions approach is based on the universal flow profiles in the boundary layer along a wall, which can be divided into three regions [1] designated by the viscous sublayer ($0 < d^+ < 5$), the buffer sublayer ($5 < d^+ < 30$), and the inertial sublayer ($30 < d^+ < 200$), respectively, with the normalized distance to the wall d^+ defined as

$$d^+ = \frac{d_\perp u_\tau}{v} = y^+ \tag{17.66}$$

where $d_\perp$ is the normal distance to the wall, v is the kinematic viscosity ($= \mu/\rho$), and u_τ is the friction velocity expressed in terms of the wall shear stress τ_w as

$$u_\tau = \sqrt{\frac{|\tau_w|}{\rho}} \tag{17.67}$$

where $|\tau_w|$ is the magnitude of the wall shear stress. Measurements and direct numerical simulations have shown that turbulence is negligible in the viscous sublayer, viscous effects are small in the inertial sublayer, while both effects are important in the buffer layer [44] with the maximum turbulent production occurring at nearly $d^+ = 12$, with the location slightly dependent on the Reynolds number making modeling of the flow in the buffer region very difficult. Because of this, turbulence models avoid the buffer layer near a wall by placing the first internal grid point either in the viscous or the inertial sublayer. The practice of placing the first grid point in the viscous sublayer is adopted with low Reynolds number turbulence models while the other practice is used with high Reynolds number turbulence models. The empirical relations applicable in the viscous sublayer are given by [45]

$$\begin{aligned} u^+ &= d^+ \\ k^+ &= 0.1 d^{+2} \\ \varepsilon^+ &= 2\frac{k^+}{d^{+2}} = 0.2 \\ \omega^+ &= \frac{6}{C_{\beta 1} d^{+2}} \end{aligned} \tag{17.68}$$

where u^+, k^+, ε^+, and ω^+ are the normalized velocity parallel to the wall, normalized turbulence kinetic energy, normalized turbulence dissipation rate, and normalized turbulence frequency, respectively. In the general case of a moving wall with a velocity $\mathbf{v}_w$, these variables are defined as

$$\begin{aligned} u^+ &= \frac{|\mathbf{v} - \mathbf{v}_w|_{||}}{u_\tau} \\ k^+ &= \frac{k}{u_\tau^2} \\ \varepsilon^+ &= \frac{\varepsilon \nu}{u_\tau^4} \\ \omega^+ &= \frac{\omega \nu}{u_\tau^2} \end{aligned} \tag{17.69}$$

where $|\mathbf{v} - \mathbf{v}_w|_{||}$ is the magnitude of the velocity parallel to the wall.

A comparison of the above profiles with data obtained from direct numerical simulation reveals that velocity and dissipation rate remain in good agreement up to $d^+ = 10$, while the turbulence kinetic energy is over estimated at values of $d^+ > 5$. Finally in the low Reynolds turbulence formulation, the $k - \omega$ based models just require a boundary treatment that satisfies the model asymptotic values. On the other hand, in the $k - \varepsilon$ based models a damping function is added for the eddy viscosity equation that mimics the direct effect of molecular viscosity on the shear stress [25]. In the inertial sub-layer, the momentum profile is derived assuming a one-dimensional Couette flow with zero pressure gradient. Profiles for the turbulence quantities can be derived for a specific turbulence model. For the $k - \varepsilon$ and $k - \omega$ models these profiles are given by

$$\begin{aligned} u^+ &= \frac{1}{\kappa} Ln(d^+) + B \\ k^+ &= \frac{1}{\sqrt{C_\mu}} = \frac{1}{\sqrt{\beta^*}} \\ \varepsilon^+ &= \frac{\nu}{u_\tau \kappa d_\perp} \\ \omega^+ &= \frac{\nu}{u_\tau \kappa d_\perp \sqrt{\beta^*}} \end{aligned} \tag{17.70}$$

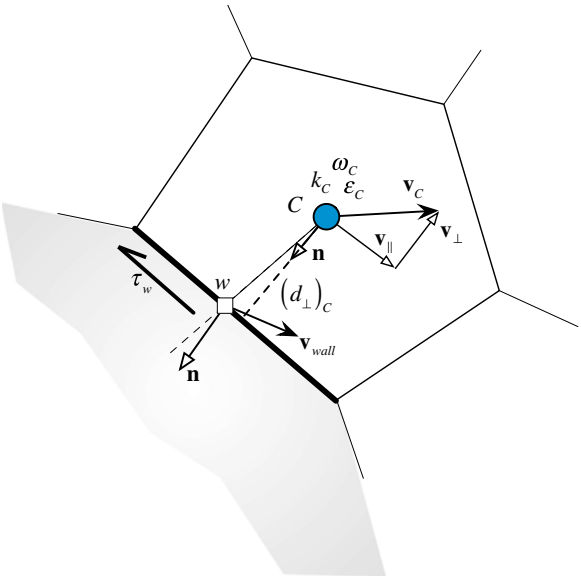


Fig. 17.2 A boundary control volume next to a wall

where the von Karman constant κ is assigned the value 0.41, $C_\mu = \beta^* = 0.09$ and $B = 5.25$. Data obtained from Direct Numerical Simulation (DNS) indicate excellent agreement for the velocity profile. On the other hand, turbulent quantities are less accurate.

As shown in Fig. 17.2, when solving a turbulent flow problem, modifications to the conservation equations are made at the first interior point C in the control volume next to the wall. The value of d^+ at that location, denoted by d_C^+, is first calculated to infer whether the point lies in the viscous or inertial sublayer. The value of d_C^+ is computed from the definition of d^+ and the value of u_τ obtained from the law of the wall as

$$\left.\begin{aligned} d_C^+ &= \frac{(d_\perp)_C u_\tau}{\nu} \\ k_C^+ = \frac{k_C}{u_\tau^2} &= \frac{1}{\sqrt{C_\mu}} \Rightarrow u_\tau = C_\mu^{1/4} k_C^{1/2} \end{aligned}\right\} \Rightarrow d_C^+ = \frac{C_\mu^{1/4} k_C^{1/2}}{\nu} (d_\perp)_C \qquad (17.71)$$

The transition from the viscous to the inertial layer is assumed to occur at a limiting value of d^+, denoted by d_{lim}^+. Different values for d_{lim}^+ are reported in different sources. All values however are between 11 and 12. A value of 11.06 is adopted here. This limiting value marks the intersection between the logarithmic and the linear profile. If $d_C^+ < d_{\text{lim}}^+$ then the grid point lies in the viscous sublayer, otherwise it is located in the inertial sublayer.

If the first grid point is in the viscous sublayer, then the flow is assumed to be laminar and the viscosity at the wall is set equal to the laminar viscosity μ and the shear stress is computed as for laminar flows. A fixed value of zero is imposed on the turbulence kinetic energy with the production of turbulence kinetic energy at the

first interior point modified by assuming that the shear stress is constant over the control volume with its value computed as

$$P_k \approx \tau_w \frac{\partial(\mathbf{v}_C - \mathbf{v}_w)_{||}}{\partial(d_\perp)}\bigg|_w = \mu \frac{(|\mathbf{v}_C - \mathbf{v}_w|_{||})^2}{(d_\perp)_C^2} \tag{17.72}$$

In the standard $k-\varepsilon$ model the dissipation rate of turbulence kinetic energy at the centroid of the first control volume next to the wall is computed by setting the laminar viscosity equal to the turbulent viscosity to yield

$$\varepsilon_C = \frac{C_\mu \rho k_C^2}{\mu} \tag{17.73}$$

whereas in the $k-\omega$ model the value of the turbulence frequency ω_C is computed from the analytical solution in the viscous sublayer as

$$\omega_C = \frac{6\nu}{C_{\beta 1}(d_\perp)_C^2} \tag{17.74}$$

If $d_C^+ > d_{\text{lim}}^+$ then the grid point is located in the inertial sublayer and the logarithmic wall functions are applied at the first interior point C. The implementation process involves computing the shear stress using the logarithmic wall function as

$$|\tau_w| = \rho u_\tau^2 = \frac{\rho u_\tau |\mathbf{v}_C - \mathbf{v}_w|_{||}}{\frac{1}{\kappa} Ln(d_C^+) + B} \Rightarrow \tau_w = -\frac{\rho u_\tau}{\frac{1}{\kappa} Ln(d_C^+) + B} (\mathbf{v}_C - \mathbf{v}_w)_{||} \tag{17.75}$$

where the fact that

$$\tau_w = -|\tau_w| \frac{(\mathbf{v}_C - \mathbf{v}_w)_{||}}{|\mathbf{v}_C - \mathbf{v}_w|_{||}} \tag{17.76}$$

has been used. This shear stress is used in solving the momentum equation either by invoking its value directly as a source term ($\tau_w S_b$) or via a modified viscosity at the wall μ_w computed as

$$|\tau_w| = \mu_w \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{(d_\perp)_C} = \frac{\rho u_\tau |\mathbf{v}_C - \mathbf{v}_w|_{||}}{\frac{1}{\kappa} Ln(d_C^+) + B} \Rightarrow \mu_w = \frac{\rho u_\tau (d_\perp)_C}{\frac{1}{\kappa} Ln(d_C^+) + B} \tag{17.77}$$

with the vector form of the wall shear stress expressed as

$$\tau_w = -\frac{\mu_w}{(d_\perp)_C} (\mathbf{v}_C - \mathbf{v}_w)_{||} \tag{17.78}$$

In either case the implementation follows the procedures described in Chap. 15.

It is worth noting that the shear stress can also be formulated in terms of the normalized quantities as

$$\begin{aligned} |\tau_w| &= \mu_{lam} \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{(d_\perp)_C} \frac{d^+}{u^+} \\ &= \tau_{lam} \frac{d^+}{u^+} \end{aligned} \tag{17.79}$$

In solving the turbulence kinetic energy equation, the value of k is assumed to prevail over the control volume (i.e., a zero gradient for k is used) with the term representing the production of the turbulence kinetic energy at the first interior point next to the wall modified by assuming that the shear stress is constant over the control volume with its value equal to that at the wall, while the velocity gradient is computed from the wall function as

$$\begin{aligned} u^+ = \frac{1}{\kappa} ln(d^+) + B \Rightarrow \frac{|\mathbf{v} - \mathbf{v}_w|_{||}}{u_\tau} &= \frac{1}{\kappa} ln\left(\frac{d_\perp u_\tau}{\nu}\right) + B \\ \Rightarrow \left.\frac{d\left(|\mathbf{v} - \mathbf{v}_w|_{||}\right)}{d(d_\perp)}\right|_w &= \frac{u_\tau}{\kappa (d_\perp)_C} \end{aligned} \tag{17.80}$$

Therefore, if the first interior point lies in the inertial sublayer, the production term in the turbulence kinetic energy equation is computed as

$$P_k = |\tau_w| \frac{u_\tau}{\kappa (d_\perp)_C} \tag{17.81}$$

It is also useful to formulate the production term P_k in terms of the normalized parameters. Starting with Eq. (17.72), the generic production term can be written as

$$\begin{aligned} P_k &\approx \tau_w \left.\frac{\partial (\mathbf{v}_C - \mathbf{v}_w)_{||}}{\partial (d_\perp)}\right|_w \\ &= \rho u_\tau^2 \left.\frac{\partial (\mathbf{v}_C - \mathbf{v}_w)_{||}}{\partial (d_\perp)}\right|_w \\ &= \rho u_\tau^2 \frac{\partial u^+}{\partial d^+} \frac{u_\tau^2}{\mu_{lam}} \\ &= \frac{\tau_{lam}^2}{\mu_{lam}} \left(\frac{d^+}{u^+}\right)^2 \frac{\partial u^+}{\partial d^+} \end{aligned} \tag{17.82}$$

In the $k - \varepsilon$ model the ε equation is not solved at the first interior point next to the wall. Rather its value is set by requiring the dissipation of turbulence kinetic energy to be equal to its production rate such that

$$\left.\begin{aligned} \rho\varepsilon_C = P_k = |\tau_w|\frac{u_\tau}{\kappa(d_\perp)_C} = \frac{\rho u_\tau^3}{\kappa(d_\perp)_C} \\ u_\tau = C_\mu^{1/4} k_C^{1/2} \end{aligned}\right\} \Rightarrow \varepsilon_C = \frac{C_\mu^{3/4} k_C^{3/2}}{\kappa(d_\perp)_C} \tag{17.83}$$

If the $k-\omega$ model is used with wall functions, the same procedure is used. For the k equation the same modified production term is obtained since $C_\mu = \beta^*$. The ω equation is not solved for the first interior point and its value is set again by requiring dissipation of turbulence kinetic energy to be equal to its production rate such that

$$\left.\begin{aligned} \rho\varepsilon_C = P_k = \rho\frac{C_\mu^{3/4} k_C^{3/2}}{\kappa(d_\perp)_C} \\ \omega_C = \frac{\varepsilon_C}{C_\mu k_C} \end{aligned}\right\} \Rightarrow \omega_C = \frac{k_C^{1/2}}{\kappa C_\mu^{1/4}(d_\perp)_C} \tag{17.84}$$

17.8.3 Improved Wall Functions

The above formulation is valid under local equilibrium conditions and results in a zero viscosity (Eq. 17.77) when τ_w vanishes, since $u_\tau = \sqrt{\tau_w/\rho}$, as is the case at reattachment and separating points. A generalization to the above formulation valid under local non-equilibrium conditions has been proposed for the $k-\varepsilon$ model by Launder and Spalding [46]. In their work, $\sqrt{k}$ is used as the characteristic turbulent velocity scale, instead of the friction velocity, through the identity

$$u^* = C_\mu^{1/4}\sqrt{k} \tag{17.85}$$

such that the wall viscosity and shear stress become

$$\begin{aligned} \mu_w &= \frac{\rho u^*(d_\perp)_C}{\frac{1}{\kappa}Ln(d_C^*) + B} \\ \tau_w &= \rho u_\tau u^* \\ u_\tau &= \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{\frac{1}{\kappa}Ln(d_C^*) + B} \end{aligned} \tag{17.86}$$

where d_C^* is defined using u^* as

$$d_C^* = \frac{(d_\perp)_C u^*}{\nu} \tag{17.87}$$

Equation (17.86) clearly indicates that the turbulent viscosity does not vanish when $\tau_w = 0$.

Similar to the standard wall functions case, the value of d_C^* is first computed. If the value of $d_C^* < d_{\text{lim}}^* = 11.06$ then the first point is in the viscous sublayer and the procedure described for the standard wall functions is used.

If the value of $d_C^* > d_{\text{lim}}^*$ the first interior grid point is located in the inertial sublayer and the wall viscosity is computed via Eq. (17.86). Using this viscosity the wall shear stress is obtained and implemented as explained earlier. To find k_C, the k-conservation equation is solved. The procedure described with the standard wall functions for calculating the production and dissipation terms in the near wall control volume, assumed that the values of k_C and ε_C prevail over the entire control volume. Since the P_k and $\boldsymbol{\varepsilon}$ values vary drastically across the near-wall cell, evaluating them at the cell centre in discretizing the k equation leads to inaccurate approximations. To improve predictions, Launder and Spalding [46] suggested suitable approximations for their cell-average values. Starting with the production term, its average value is computed as

$$\begin{aligned}\overline{P}_k &= \frac{1}{(d_\perp)_C}\int_0^{(d_\perp)_C} P_k d(d_\perp) \\ &= \frac{1}{(d_\perp)_C}\int_0^{(d_\perp)_C} |\tau_w|\frac{d(|\mathbf{v}-\mathbf{v}_w|_{||})}{d[(d_\perp)]}d(d_\perp) = \frac{1}{(d_\perp)_C}|\tau_w||\mathbf{v}_C-\mathbf{v}_w|_{||} = \mu_w\frac{\left[|\mathbf{v}_C-\mathbf{v}_w|_{||}\right]^2}{(d_\perp)_C^2}\end{aligned} \tag{17.88}$$

Invoking Eq. (17.86), the average production term is found to be

$$\overline{P}_k = \frac{\rho C_\mu^{1/4}\sqrt{k_C}}{(d_\perp)_C\left(\frac{1}{\kappa}Ln(d_C^*)+B\right)}\left[|\mathbf{v}_C-\mathbf{v}_w|_{||}\right]^2 \tag{17.89}$$

Then, the volume integral of the production term is obtained as

$$\int_V P_k dV = \overline{P}_k V_C \tag{17.90}$$

To calculate an average turbulence dissipation rate, the integrated value of dissipation is set equal to the production rate resulting in

$$\int_0^{(d_\perp)_C}\rho\varepsilon d(d_\perp) = \int_0^{(d_\perp)_C} P_k d(d_\perp) = \int_0^{(d_\perp)_C}|\tau_w|\frac{d\left(|\mathbf{v}-\mathbf{v}_w|_{||}\right)}{d[(d_\perp)]}d(d_\perp) \tag{17.91}$$

The integral is evaluated by expressing the shear stress and velocity gradients as

$$|\tau_w| = \rho u_\tau^2$$
$$\frac{d\left(|\mathbf{v}-\mathbf{v}_w|_{||}\right)}{d[(d_\perp)]} \approx \frac{\Delta\left(|\mathbf{v}-\mathbf{v}_w|_{||}\right)}{\Delta[(d_\perp)]} = \frac{|\mathbf{v}_C-\mathbf{v}_w|_{||}}{(d_\perp)_C} = \frac{u_\tau}{(d_\perp)_C}\left(\frac{1}{\kappa}Ln(d_C^*)+B\right) \tag{17.92}$$

Thus

$$\begin{aligned}\int_0^{(d_\perp)_C} \rho\varepsilon d(d_\perp) &= \int_0^{(d_\perp)_C} \rho u_\tau^2 \frac{u_\tau}{(d_\perp)_C}\left(\frac{1}{\kappa}Ln(d_C^*)+B\right)d(d_\perp) \\ &= \rho u_\tau^3\left(\frac{1}{\kappa}Ln(d_C^*)+B\right) = \rho C_\mu^{3/4}k^{3/2}\left(\frac{1}{\kappa}Ln(d_C^*)+B\right)\end{aligned} \tag{17.93}$$

and the average turbulence dissipation rate is computed as

$$\overline{\varepsilon_C} = \frac{1}{\rho(d_\perp)_C}\int_0^{(d_\perp)_C} \rho\varepsilon d(d_\perp) = \frac{C_\mu^{3/4}k_C^{3/2}}{(d_\perp)_C}\left(\frac{1}{\kappa}Ln(d_C^*)+B\right) \tag{17.94}$$

while its volume integral is found to be

$$\int_V \rho\varepsilon dV = \rho\overline{\varepsilon_C}V_C \tag{17.95}$$

As for the standard wall functions, the ε equation is not solved at the first interior point next to the wall and its value is set using Eq. (17.83). Moreover, similar equations can be developed for use with the $k-\omega$ model by transforming ε into ω via Eq. (17.32).

17.8.4 Scalable Wall Functions

The wall functions approach is most accurate when the first grid point in the near wall region lies in the inertial sublayer at a normalized distance $d^* > d_{low}^*$ where, depending on the numerical formulation, $d_{low}^* \approx 20$. This represents a serious limitation in situations where the boundary layer is very thin as it cannot be resolved with a coarse near-wall grid. The scalable wall function approach developed in [47, 48] overcomes this hurdle by slightly modifying the calculation of the wall shear stress by redefining u^+ as

$$u^+ = \frac{1}{\kappa}\ln\tilde{d}^* + B \tag{17.96}$$

where

$$\tilde{d}^* = \max(d^*, d^*_{\text{lim}}) \quad d^*_{\text{lim}} \approx 11.06 \tag{17.97}$$

With this formulation, the definition of $\tilde{d}^*$ becomes independent of the grid spacing as it prevents the first grid point from being located in the viscous sublayer leading to consistent results for grids of arbitrary refinement. The simulation error introduced is related to not accounting for the viscous sublayer, which is the case for all wall function formulations. It should be clarified however that this error can be significant for flows with relatively low Reynolds number. The implementation of the scalable wall functions is straightforward whereby the procedures described above remains unchanged with the exception of replacing d^* or d^+ by, respectively, $\tilde{d}^*$ or $\tilde{d}^+$.

This approach is usually used when the details of the boundary layer is not of interest. If the purpose of using a fine grid near the wall is to examine the details of the boundary layer then a low Reynolds number model should be used as explained next.

17.8.5 Wall Boundary Conditions for Low Reynolds Number Models

With low Reynolds number turbulence models a fine grid resolution must be used in the near wall region in order to properly resolve the viscous sublayer allowing laminar flow boundary conditions to be applied. Therefore, the boundary condition to be used at the wall in the momentum equation is the no slip condition presented in Chap. 15 with no need to be repeated here.

The $k-\omega$ and $k-\varepsilon$ turbulence models and their variants are examples of this type. For all these models, the asymptotic boundary conditions applicable for k, ε and ω are set to

$$\begin{aligned} k_w &\to 0 \\ \varepsilon_w &\to 2\frac{vk}{(d_\perp)^2_C} \\ \omega_w &\to 10\frac{6v}{C_{\beta 1}(d_\perp)^2_C} \end{aligned} \tag{17.98}$$

The value for ω_w given by Eq. (17.98), which is intended to be applied at the first interior grid point next to the wall, is valid for grid points located at d^+ or $d^* < 2.5$ with 15–20 nodes needed in that region for grid independent solutions. The factor of 10 is introduced into Eq. (17.98) based on the recommendation of Menter as it eliminates the need to specify ω at any other internal point beyond the one next to the wall.

17.8.6 Automatic Near-Wall Treatment

With the wall functions method the normalized distance to the wall is first calculated to infer wether the first interior point lies in the viscous or inertial sublayer. In the viscous sublayer the flow is treated as laminar. Because the standard $k-\varepsilon$ model is valid in the fully turbulent regions and does not include damping functions to model viscous effects it introduces error into the solution. To eliminate this error care should be exercised to make sure that the first interior point is in the log region. This however eliminates the influence of the viscous sublayer, which may be significant on the solution. Therefore it is desirable to have the option of resolving the viscous sublayer without the stringent requirement of a very fine grid near the wall. This is possible with the $k-\omega$ model and its variants as the ω equation can be integrated all the way to the wall without the need for any additional damping functions. The idea is to develop a method that switches automatically between the low and the high Reynolds number formulation based on the value of the normalized distance to the wall. This task is achievable by the ω equation because of its known analytical solutions in the viscous and inertial sublayers. For that purpose, both solutions are blended smoothly according to [34]

$$\omega = \left[\omega_{vis}^2 + \omega_{\log}^2\right]^{0.5} \tag{17.99}$$

where

$$\omega_{vis} = \frac{6\nu}{C_{\beta 1} d_\perp^2} \quad \omega_{\log} = \frac{u^*}{\kappa d_\perp \sqrt{\beta^*}} = \frac{(u^*)^2}{\kappa \nu d^* \sqrt{\beta^*}} \tag{17.100}$$

The shear stress in the momentum equation is computed as

$$|\tau_w| = \rho u_\tau u^* = \rho \frac{u^*}{u^+} |\mathbf{v}_C - \mathbf{v}_w|_{||} \tag{17.101}$$

with the values of the velocities u_τ and u^* in the near wall region calculated using

$$u_\tau^{vis} = \sqrt{\frac{\mu}{\rho} \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{(d_\perp)_C}} \quad u_\tau^{\log} = \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{\frac{1}{\kappa}\ln d_C^+ + B} \quad u_\tau = \left[\left(u_\tau^{vis}\right)^4 + \left(u_\tau^{\log}\right)^4\right]^{0.25} \tag{17.102}$$

$$u_{vis}^* = \sqrt{\frac{\mu}{\rho} \frac{|\mathbf{v}_C - \mathbf{v}_w|_{||}}{(d_\perp)_C}} \quad u_{\log}^* = \beta^{*1/4} k^{1/2} \quad u^* = \left[\left(u_{vis}^*\right)^4 + \left(u_{\log}^*\right)^4\right]^{0.25} \tag{17.103}$$

In solving the k equation, the gradient at the wall is set to zero and the production in the near wall cell is altered to

$$P_k = |\tau_w| \frac{u_\tau}{\kappa (d_\perp)_C} = \rho \left(\frac{u^*}{u^+}\right)^2 \frac{\partial u^+}{\partial d^+} \left(|\mathbf{v}_C - \mathbf{v}_w|_{||}\right)^2 \tag{17.104}$$

Based on the grid spacing, this blending approach permits a smooth shift of the wall treatment from a viscous sublayer to a wall function.

17.8.7 Near-Wall Heat Transfer

Similar to velocity profiles, near wall temperature profiles should also be corrected when high Reynolds number turbulence models are used. These profiles are obtained by an adjusted Reynolds analogy from the velocity profiles by altering the log-law to provide an equation that relates the value of temperature at the near wall node T_C, to that at the wall T_w, and to the wall heat flux q_w. The procedure starts by defining a normalized temperature T^+ as

$$T^+ = \frac{T_w - T}{T^*} \tag{17.105}$$

where T^* is the modified friction temperature given by

$$T^* = \frac{q_w}{\rho c_p u^*} \tag{17.106}$$

and other variables are as defined earlier. Then combining the above two equations, the normalized temperature equation becomes

$$T^+ = \frac{\rho c_p u^*}{q_w}(T_w - T) \tag{17.107}$$

For the standard wall functions formulation, if the first interior point is located in the viscous sublayer then the normalized temperature is calculated as

$$T^+ = Pr\, d^* \tag{17.108}$$

whereas if it lies in the inertial sublayer then it is computed using the law of the wall as

$$T^+ = 2.12Ln(d^*) + \beta(Pr) \tag{17.109}$$

where

$$\beta(Pr) = (3.85Pr^{1/3} - 1.3)^2 + 2.12Ln(Pr) \tag{17.110}$$

and Pr is the laminar Prandtl number. For the scalable wall functions formulation the same equation is used with d^+ replaced by $\tilde{d}^+$, as explained earlier. With the automatic near wall treatment approach, the equation suggested by Kader [49], which blends the viscous sublayer with the law of the wall, is used. In fact this

approach may also be used with the standard and scalable wall functions approaches. According to this formula the normalized temperature is computed as

$$T^+ = Pr\, d^* e^{-\Gamma} + [2.12Ln(1 + d^*) + \beta(Pr)]e^{-1/\Gamma} \tag{17.111}$$

with the blending function Γ given by

$$\Gamma = \frac{0.01(Pr\, d^*)^4}{1 + 5Pr^3 d^*} \tag{17.112}$$

Moreover, Eq. (17.107) can be interpreted in two different ways depending on the physical boundary condition imposed on temperature. For a given wall heat flux boundary condition, the numerical condition for T_w will be a fixed value, iteratively updated, given by

$$\begin{aligned} T_w &= T_C + \frac{q_w T_C^+}{\rho c_p u^*} \\ &= T_C + \frac{q_w T_C^+}{\rho c_p C_\mu^{1/4} \sqrt{k_C}} \end{aligned} \tag{17.113}$$

If, instead, temperature is imposed on the boundary, then the temperature gradient must be held fixed numerically and calculated as

$$\begin{aligned} q_w &= \frac{\rho c_p u^*}{T_C^+}(T_w - T_C) \\ &= \frac{\rho c_p C_\mu^{1/4} \sqrt{k_C}}{T_C^+}(T_w - T_C) \end{aligned} \tag{17.114}$$

17.8.8 Other Boundary Conditions

Beside walls, conditions at other boundaries are needed. This include inlet, outlet, and symmetry boundary conditions. At inlet to a domain the values of the turbulence kinetic energy and dissipation rate, which are usually unknown, are required. If the values are known from measurements then they should be used, otherwise they must be estimated. The easiest way is to assign the values for k and ε. Values may also be specified via the turbulence intensity I defined as

$$I = \frac{\sqrt{\overline{\mathbf{v}' \cdot \mathbf{v}'}}}{\sqrt{\mathbf{v} \cdot \mathbf{v}}} \tag{17.115}$$

from which the turbulence kinetic energy is obtained as

$$k = \frac{1}{2} I^2 (\mathbf{v} \cdot \mathbf{v}) \tag{17.116}$$

A value of turbulence intensity between 1 and 10 % is usually used with values less than 1 % considered low while values higher than 10 % considered high. The values of the turbulence dissipation rate ε and turbulence frequency ω are computed from Eqs. (17.23) and (17.32) by specifying a turbulence length scale whose value is dependent on the largest eddy dimension but it is usually set to one-tenth of the width of a shear layer or the domain size, with their expressions given by

$$\begin{aligned} \varepsilon &= C_\mu \frac{k^{3/2}}{\ell} \\ \omega &= \frac{k^{1/2}}{\ell} \end{aligned} \tag{17.117}$$

The value for ε and ω may also be computed from knowledge of k and the turbulent to laminar viscosity ratio using

$$\begin{aligned} \varepsilon &= C_\mu \rho \frac{k^2}{\mu} \left(\frac{\mu}{\mu_t} \right) \\ \omega &= \rho \frac{k}{\mu} \left(\frac{\mu}{\mu_t} \right) \end{aligned} \tag{17.118}$$

At an outlet and a symmetry boundary, the treatment for k, ε and ω is similar to that for the general scalar variable ϕ presented in previous chapters and is deemed unnecessary to be repeated.

17.9 Calculating Normal Distance to the Wall

In the BSL and SST turbulence models, the normal distance to the nearest wall $d_\perp$ is needed over the entire domain to determine the interface between the computation regions of $k - \varepsilon$ and $k - \omega$, as reflected by Eq. (17.42) for the BSL model and (17.47) for the SST model. Search procedures to compute $d_\perp$ are computationally expensive in three-dimensional situations even for fixed grids. The situation gets worse with a moving grid as the search has to be repeated at every time step. This has forced workers to introduce approximations in the calculation of $d_\perp$ that incur large errors.

To avoid the expensive search procedures, techniques based on solving differential equations for $d_\perp$ have been developed. These methods are based on solving a Poisson, Eikonal, or Hamilton-Jacobi equations [50–53]. The attraction in these equations is their composition which involves gradient and/or Laplacian operators

that are readily available in CFD solvers. This renders the approach easy to implement, stable, and economical especially with moving grids.

Adopting a Poisson-like approach, the following differential equation for a variable ϕ is solved:

$$\nabla^2\phi = -1 \tag{17.119}$$

subject to

$$\begin{cases} \phi = 0 & \text{on walls} \\ \nabla\phi \cdot \mathbf{n} = 0 & \text{elsewhere} \end{cases} \tag{17.120}$$

The normal distance to the nearest wall is computed using the predicted value of ϕ and its gradient as

$$\begin{aligned} d_\perp &= -|\nabla\phi| + \sqrt{|\nabla\phi|^2 + 2\phi} \\ &= -\sqrt{\left(\frac{\partial\phi}{\partial x}\right)^2 + \left(\frac{\partial\phi}{\partial y}\right)^2 + \left(\frac{\partial\phi}{\partial z}\right)^2} + \sqrt{\left(\frac{\partial\phi}{\partial x}\right)^2 + \left(\frac{\partial\phi}{\partial y}\right)^2 + \left(\frac{\partial\phi}{\partial z}\right)^2 + 2\phi} \end{aligned} \tag{17.121}$$

While solving a turbulent flow problem, Eq. (17.119) is discretized on the grid network generated for solving the problem using the finite volume method and following the procedures described in Chap. 8. During the solution procedure, a converged solution for Eq. (17.119) is first obtained, from which the normal distances to the wall are computed, prior to solving the turbulent flow problem. Referring to Fig. 17.3, and recalling that the term $gDiff_f$ is defined as

$$gDiff_f = \frac{E_f}{d_{CF}} \tag{17.122}$$

the discretized form of Eq. (17.119) can be written as

$$a_C\phi_C + \sum_{F\sim NB(C)} a_F\phi_F = b_C \tag{17.123}$$

where

$$\begin{aligned} a_F &= FluxF_f = -gDiff_f \\ a_C &= \sum_{f\sim NB(C)} FluxC_f = -\sum_{f\sim NB(C)} FluxF_f = \sum_{f\sim NB(C)} gDiff_f \\ b_C &= V_C + \sum_{f\sim nb(C)} \left((\nabla\phi)_f \cdot \mathbf{T}_f\right) \end{aligned} \tag{17.124}$$

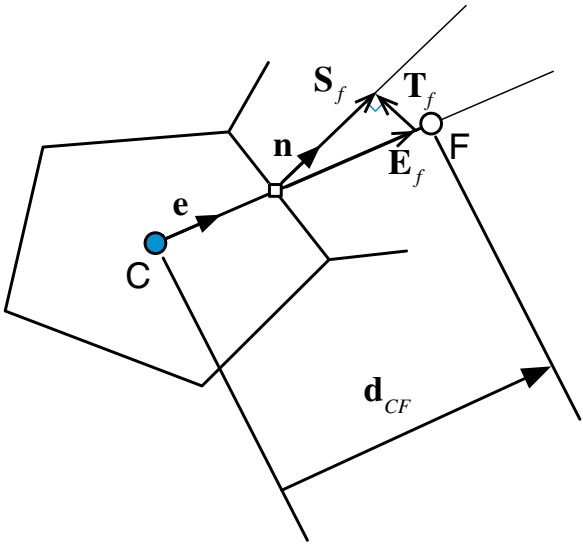


Fig. 17.3 A two-dimensional control volume with its geometrical quantities

Moreover, the Dirichlet and Von Neumann boundary conditions described by Eq. (17.120) are treated as explained in Chap. 8.

Following the solution for Eq. (17.119), the normal distance to the wall at all cell centroids in the domain are calculated using Eq. (17.121) with the gradient calculated as detailed in Chap. 9.

17.10 Computational Pointers

OpenFOAM® [54] implements several LES and RANS turbulence models for both compressible and incompressible flows. The root directory for all models is denoted by "FOAM_SRC/turbulenceModels". Incompressible turbulence models are located in the sub-directory "FOAM_SRC/turbulenceModels/incompressible" within which the three sub-sub-directories "LES" (refers to the large eddy simulation approach), "RAS" (refers to the Reynolds-averaged Navier-Stokes approach), and "turbulenceModel" reside. The first two sub-sub-directories "LES" and "RAS" define the special features of the LES and RAS models, respectively, while in "turbulenceModel" the abstract base classes of both incompressible RAS and LES models are defined. The base class defines a series of abstract virtual functions that have to be specified for any derived class as shown in Listing 17.1.

```
        //- Return the turbulence viscosity
        virtual tmp<volScalarField> nut() const = 0;

        //- Return the effective viscosity
        virtual tmp<volScalarField> nuEff() const = 0;

        //- Return the turbulence kinetic energy
        virtual tmp<volScalarField> k() const = 0;
…
        //- Return the turbulence kinetic energy dissipation rate
        virtual tmp<volScalarField> epsilon() const = 0;

        //- Return the Reynolds stress tensor
        virtual tmp<volSymmTensorField> R() const = 0;
```

Listing 17.1 Script used to define virtual functions

The base class also defines the normal distance to the wall, which is a useful quantity for turbulence models that can be used with all derived classes. The statement used for that is given in Listing 17.2 as

```
        //- Return the near wall distances
        const nearWallDist& y() const
        {
            return y_;
        }
```

Listing 17.2 Statement used to define the normal distance to the wall

To better understand the code structure that defines the OpenFOAM® turbulence models, in the following the $k - \varepsilon$ model with the Spalding wall functions and the SST $k - \omega$ model are used as examples.

The model definition can be found in the directory "FOAM_SRC/turbulenceModels/incompressible/RAS" where all the RANS turbulence models are placed and defined. For the RANS models, OpenFOAM® defines an additional non virtual base class named RASModel (deriving the turbulenceModel class) from which all models are derived (Listing 17.3).

```
class RASModel
:
    public turbulenceModel,
    public IOdictionary
{
…
        //- Allow omegaMin to be changed
        dimensionedScalar& omegaMin()
        {
            return omegaMin_;
        }
        //- Const access to the coefficients dictionary
        virtual const dictionary& coeffDict() const
        {
                return coeffDict_;
        }
        //- Return the effective viscosity
        virtual tmp<volScalarField> nuEff() const
        {
            return tmp<volScalarField>
            (
                new volScalarField("nuEff", nut() + nu())
            );
        }
```

Listing 17.3 Script used to define the non virtual base class RASModel

This class is just a wrapper mainly for bounding the values of the effective viscosity and turbulence quantities, as well as a dictionary definition.

17.10.1 The $k - \varepsilon$ Model

The "kEpsilon" class implements the standard version of the $k - \varepsilon$ model given by Eqs. (17.25) and (17.26). It defines the necessary constants and set of variables used in the model, as shown in Listing 17.4.

```
class kEpsilon
:
    public RASModel
{
protected:

    // Protected data
        // Model coefficients
            dimensionedScalar Cmu_;
            dimensionedScalar C1_;
            dimensionedScalar C2_;
            dimensionedScalar sigmaEps_;
        // Fields
            volScalarField k_;
            volScalarField epsilon_;
            volScalarField nut_;
```

Listing 17.4 Script defining the $k - \varepsilon$ model

It is selectable from the "RASProperties" dictionary with the name "kEpsilon" defined under the "TypeName" of the class as (Listing 17.5)

```
    //- Runtime type information
    TypeName("kEpsilon");
```

Listing 17.5 Statement used to select the $k-\varepsilon$ model

Being derived from the base virtual class "turbulenceModel", all its virtual base functions have to be defined as follows (Listing 17.6).

```
   //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

   //- Return the source term for the momentum equation
      virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;
```

Listing 17.6 Statements used to define the effective stress tensor and the diffusion field in the momentum equation

The "divDevReff" function returns the diffusion contribution in the momentum equation including the Reynolds stress, i.e.,

$$\text{divDevReff}(\mathbf{v}) = \left[\nabla \cdot \left(\overline{\boldsymbol{\tau}} - \rho\overline{\mathbf{v}'\mathbf{v}'}\right)\right] \tag{17.125}$$

In this case the type of data returned by the function (Listing 17.7) is in the form of an fvMatrix defined as

```
tmp<fvVectorMatrix> kEpsilon::divDevReff(volVectorField& U) const
{
    return
    (
      - fvm::laplacian(nuEff(), U)
      - fvc::div(nuEff()*dev(T(fvc::grad(U))))
    );
}
```

Listing 17.7 Implicit and explicit contributions of the Laplacian term

in which the Laplacian of the velocity field is split into an implicit and an explicit matrix contribution. It is important to mention that the divDevReff term depends only on the velocity gradient field and the total or effective viscosity (nuEff()). This

means that OpenFOAM® implements the wall shear stress contribution by modifying only the turbulent viscosity at the wall; thus implementing Eq. (17.77).

The assembly and solution of the turbulence model equations are defined via the "correct()" member function as in Listing 17.8.

```
void kEpsilon::correct()
{
…
```

Listing 17.8 Assembly and solution of the turbulence model

Here OpenFOAM® first assembles and then solves the ε equation before the k equation, using the script in Listing 17.9 as

```
  volScalarField G(GName(), nut_*2*magSqr(symm(fvc::grad(U_))));
 // Update epsilon and G at the wall
  epsilon_.boundaryField().updateCoeffs();
  // Dissipation equation
  tmp<fvScalarMatrix> epsEqn
  (
      fvm::ddt(epsilon_)
    + fvm::div(phi_, epsilon_)
    - fvm::laplacian(DepsilonEff(), epsilon_)
   ==
      C1_*G*epsilon_/k_
    - fvm::Sp(C2_*epsilon_/k_, epsilon_)
  );
  epsEqn().relax();
  epsEqn().boundaryManipulate(epsilon_.boundaryField());
  solve(epsEqn);
  bound(epsilon_, epsilonMin_);
```

Listing 17.9 Assemble and solve the ε equation

where the G field defines the global production term previously defined in Eq. (17.28) as P_k. Based on the wall functions approach, it is necessary before solving the ε equation, to modify the value of ε at all centroids of cells attached to the wall using Eqs. (17.81) and (17.83). In OpenFOAM® the values of ε and P_k are altered at the wall by defining a special wall boundary condition for the field ε with the modification forced through the function "epsilon_.boundaryField(). updateCoeffs();" with the dedicated boundary definition for the dissipation rate ε found under the directory "FOAM_SRC/turbulenceModels/incompressible/RAS/derivedFvPatchFields/wallFunctions/epsilonWallFunctions/epsilonWallFunction".

Based on the wall function model this class has to update the values of the two variables ε and P_k. This operation is performed by the function "calculate" in which,

after defining all necessary variables, a loop cycle changes the corresponding values (with G in Listing 17.10 corresponding to the P_k production term). The "w" variable is just a weight factor to take into account boundary cells in contact with more than one wall (i.e., corners). For standard faces "w" has a value of one.

```
label cellI = patch.faceCells()[faceI];
scalar w = cornerWeights[faceI];

epsilon[cellI] += w*Cmu75*pow(k[cellI], 1.5)/(kappa_*y[faceI]);

G[cellI] +=
    w
   *(nutw[faceI] + nuw[faceI])
   *magGradUw[faceI]
   *Cmu25*sqrt(k[cellI])
   /(kappa_*y[faceI]);
```

Listing 17.10 Calculating ε and modifying the production of turbulence kinetic energy in the near wall cells

It is important to note that while G or P_k is a source term, ε is a field that is solved by a transport equation. To impose in a cell a value previously calculated, the matrix has to be manipulated at the right location, in order to return the correct value.Thus the class "epsilonWallFunctionFvPatchScalarField" is derived, as shown in Listing 17.11, from the class "fixedInternalValueFvPatchField" according to

```
class epsilonWallFunctionFvPatchScalarField
:
    public fixedInternalValueFvPatchField<scalar>
{
…
```

Listing 17.11 Creating the class "epsilonWallFunctionFvPatchScalarField" from the class "fixedInternalValueFvPatchField"

with class "fixedInternalValueFvPatchField", shown in Listing 17.12, being just a wrapper class containing a special function to manipulate the matrix by imposing the expected values of the variable in a list of cells.

```
template<class Type>
class fixedInternalValueFvPatchField
:
    public zeroGradientFvPatchField<Type>
…
template<class Type>
void Foam::fixedInternalValueFvPatchField<Type>::manipulateMatrix
(
    fvMatrix<Type>& matrix
)
{

    // Apply the patch internal field as a constraint in the matrix
    matrix.setValues(this->patch().faceCells(), this->patchInternalField());
}
```

Listing 17.12 The functionality of the "fixedInternalValueFvPatchField" class

This function is called form form the "kEpsilon::correct()" class after assembling the matrix with "epsEqn().boundaryManipulate(epsilon_.boundaryField());".

Once the ε equation is solved, OpenFOAM® proceeds to assembling and solving the k equation, as shown in Listing 17.13.

```
    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(epsilon_/k_, k_)
    );
    kEqn().relax();
    solve(kEqn);
    bound(k_, kMin_);
```

Listing 17.13 Assemble and solve the k equation

In this case no additional manipulation is required as the production term P_k is already changed at the wall and the boundary condition for k is just a zero gradient ("zeroGradient") type.

After calculating the k and ε values, the turbulent eddy viscosity ν_t ($=\mu_t/\rho$) is updated and then corrected at wall boundaries according to Eq. (17.77). This is accomplished using the following statements in Listing 17.14:

```
    // Re-calculate viscosity
    nut_ = Cmu_*sqr(k_)/epsilon_;
    nut_.correctBoundaryConditions();
```

Listing 17.14 Updating the turbulent eddy viscosity at walls

The "volScalarField" nut (ν_t) is defined in the constructor of the kEpsilon class shown in Listing 17.15 as

```
nut_
(
    IOobject
    (
        "nut",
        runTime_.timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    autoCreateNut("nut", mesh_)
)
```

Listing 17.15 Script used to define the turbulent eddy viscosity

in which the function "autoCreateNut" is used. This function is defined in the file "backwardsCompatibilityWallFunctions.C", which is placed in the directory "FOAM_SRC/src/turbulenceModels/incompressible/RAS/backwardsCompatibility/wallFunctions". The "autoCreateNut" acts in order to create the (ν_t) object by reading the file and the related boundary types only if it is already present inside the working directory (Listing 17.16).

```
if (nutHeader.headerOk())
{
  return tmp<volScalarField>(new volScalarField(nutHeader, mesh));
}
else
{
```

Listing 17.16 "IF" statement for checking the definition of the nut file

For the case when the file "nut" is not found, the standard Spalding wall function is applied using the boundary patch class definition "nutkWallFunction FvPatchScalarField", through the following statement (Listing 17.17):

```
if (isA<wallFvPatch>(bm[patchI]))
 {
  nutBoundaryTypes[patchI] =
  nutkWallFunctionFvPatchScalarField::typeName;
 }
```

Listing 17.17 Application of the standard Spalding wall function

The class "nutkWallFunctionFvPatchScalarField" described in Listing 17.18 is located in the "FOAM_SRC/turbulenceModels/incompressible/RAS/derivedFvPatchFields/wallFunctions/nutWallFunctions" directory and inherits a base class named "nutWallFunctionFvPatchScalarField", i.e.,

```
class nutkWallFunctionFvPatchScalarField
:
    public nutWallFunctionFvPatchScalarField
{
…
```

Listing 17.18 Class definition

The "nutWallFunctionFvPatchScalarField" in Listing 17.19 is a base class that wraps the changing of the boundary value of the eddy viscosity "nut" (ν_t) by defining the related "updateCoeffs" function as

```
void nutWallFunctionFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    operator==(calcNut());

    fixedValueFvPatchScalarField::updateCoeffs();
}
```

Listing 17.19 The updateCoeffs() function definition

where the "calcNut" function is defined as pure virtual and it has to be delineated from the derived class. Based on that the derived class "nutkWallFunction FvPatchScalarField" implements the "calcNut" function according to Spalding assumption and Eq. (17.77) as Listing 17.20.

```
label faceCellI = patch().faceCells()[faceI];

scalar yPlus = Cmu25*y[faceI]*sqrt(k[faceCellI])/nuw[faceI];

if (yPlus > yPlusLam_)
{
    nutw[faceI] = nuw[faceI]*(yPlus*kappa_/log(E_*yPlus) - 1.0);
}
```

Listing 17.20 Calculate eddy viscosity at the wall according to Spalding wall function

Once the turbulent eddy viscosity is updated based on the wall function value the momentum shear stress is then correctly evaluated.

17.10.2 The SST $k - \omega$ Model

The "kOmegaSST" class implements the version of the SST $k - \omega$ model described by Eqs. (17.38) through (17.49). The set of variables and constants used in the model are defined by the script shown in Listing 17.21.

```
class kOmegaSST
:
    public RASModel
{

protected:

    // Protected data

        // Model coefficients
            dimensionedScalar alphaK1_;
            dimensionedScalar alphaK2_;

            dimensionedScalar alphaOmega1_;
            dimensionedScalar alphaOmega2_;

            dimensionedScalar gamma1_;
            dimensionedScalar gamma2_;

            dimensionedScalar beta1_;
            dimensionedScalar beta2_;

            dimensionedScalar betaStar_;

            dimensionedScalar a1_;
            dimensionedScalar b1_;
            dimensionedScalar c1_;

            Switch F3_;


        //- Wall distance field
        //  Note: different to wall distance in parent RASModel
        wallDist y_;

        // Fields

            volScalarField k_;
            volScalarField omega_;
            volScalarField nut_;
```

Listing 17.21 Script defining the SST $k - \omega$ model

```
// Protected Member Functions

    tmp<volScalarField> F1(const volScalarField& CDkOmega) const;
    tmp<volScalarField> F2() const;
    tmp<volScalarField> F3() const;
    tmp<volScalarField> F23() const;

    tmp<volScalarField> blend
    (
        const volScalarField& F1,
        const dimensionedScalar& psi1,
        const dimensionedScalar& psi2
    ) const
    {
        return F1*(psi1 - psi2) + psi2;
    }

    tmp<volScalarField> alphaK(const volScalarField& F1) const
    {
        return blend(F1, alphaK1_, alphaK2_);
    }

    tmp<volScalarField> alphaOmega(const volScalarField& F1) const
    {
        return blend(F1, alphaOmega1_, alphaOmega2_);
    }

    tmp<volScalarField> beta(const volScalarField& F1) const
    {
        return blend(F1, beta1_, beta2_);
    }

    tmp<volScalarField> gamma(const volScalarField& F1) const
    {
        return blend(F1, gamma1_, gamma2_);
    }
```

Listing 17.21 (continued)

Additional private member functions (*blend*, *alphaK*, *alphaOmega,* etc.) are now defined in order to represent the blended coefficients appearing in Eq. (17.39). Functions F1 and F2 describe the variables given by Eqs. (17.41) and (17.47), respectively. Function F3 does not appear in Menter's original model and is a modification for rough walls introduced by Hellsten [55].

As shown in Listing 17.22, the model is selectable from the "RASProperties" dictionary with the name "kOmegaSST" defined under the "TypeName" of the class as

```
//- Runtime type information
TypeName("kOmegaSST");
```

Listing 17.22 Statement used to select the SST $k - \omega$ model

The kOmegaSST class is derived from the base virtual class "turbulenceModel", and as such all its virtual base functions are specialized accordingly (Listing 17.23).

```
   //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

   //- Return the source term for the momentum equation
      virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevRhoReff
        (
            const volScalarField& rho,
            volVectorField& U
        ) const;
```

Listing 17.23 Specialization of the pure virtual functions defined in the base class "RASModel"

The implementation of the model is detailed in "FOAM_SRC/turbulence Models/incompressible/RAS/kOmegaSST/kOmegaSST.C". As in the kEpsilon class the *correct* function solves the full set of equations of the SST turbulence model. First the ω equation is setup and solved using the script shown in Listing 17.24.

```
void kOmegaSST::correct()
{
    RASModel::correct();

    const volScalarField S2(2*magSqr(symm(fvc::grad(U_))));
    volScalarField G(GName(), nut_*S2);

    // Update omega and G at the wall
    omega_.boundaryField().updateCoeffs();

    const volScalarField CDkOmega
    (
        (2*alphaOmega2_)*(fvc::grad(k_) & fvc::grad(omega_))/omega_
    );

    const volScalarField F1(this->F1(CDkOmega));

    // Turbulent frequency equation
    tmp<fvScalarMatrix> omegaEqn
    (
        fvm::ddt(omega_)
      + fvm::div(phi_, omega_)
      - fvm::laplacian(DomegaEff(F1), omega_)
     ==
        gamma(F1)*S2
      - fvm::Sp(beta(F1)*omega_, omega_)
      - fvm::SuSp
        (
            (F1 - scalar(1))*CDkOmega/omega_,
            omega_
        )
    );
    omegaEqn().relax();
    omegaEqn().boundaryManipulate(omega_.boundaryField());
    solve(omegaEqn);
    bound(omega_, omegaMin_);
```

Listing 17.24 Assemble and solve the ω equation

In the script, the G field represents the production term P_k defined in Eq. (17.28), while the CDkOmega and F1 fields are evaluated based on Eqs. (17.41), (17.42), and (17.47).

As in the kEpsilon class the values of ω and P_k are modified at the wall following the wall functions approach by defining a wall boundary condition class for ω. Modifications to the field are then enforced through the function "omega_.boundary Field().updateCoeffs();". The boundary definition for the eddy frequency ω can be found under the directory "FOAM_SRC/turbulenceModels/incompressible/RAS/derivedFvPatchFields/wallFunctions/omegaWallFunctions/omegaWallFunction".

Based on the wall functions model, this class has to update the values of the two variables ω and P_k. This operation is performed, as shown in Listing 17.25, by the usual "calculate" function using Eqs. (17.81), (17.99) and (17.100) (with G corresponding to the P_k production term).

```
label cellI = patch.faceCells()[faceI];
scalar w = cornerWeights[faceI];

scalar omegaVis = 6.0*muw[faceI]/(rhow[faceI]*beta1_*sqr(y[faceI]));

scalar omegaLog = sqrt(k[cellI])/(Cmu25*kappa_*y[faceI]);

omega[cellI] += w*sqrt(sqr(omegaVis) + sqr(omegaLog));


G[cellI] +=
    w
   *(mutw[faceI] + muw[faceI])
   *magGradUw[faceI]
   *Cmu25*sqrt(k[cellI])
   /(kappa_*y[faceI]);
```

Listing 17.25 Calculating ω and modifying the production of turbulence kinetic energy in the near wall cells

To impose the previously calculated cell value, the matrix has to be manipulated following the same procedure used with the "kEpsilon" model. Once the ω equation is solved, OpenFOAM® proceeds to assembling and solving the k equation, as shown in Listing 17.26.

```
    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(epsilon_/k_, k_)
    );
    kEqn().relax();
    solve(kEqn);
    bound(k_, kMin_);
```

Listing 17.26 Assemble and solve the k equation

In this case no additional manipulation is required as the production term P_k is already modified at the wall and the boundary condition for k is just a zero gradient ("zeroGradient") type.

Once the k and ω values are calculated the turbulent eddy viscosity $\nu_t(=\mu_t/\rho)$ is updated using the script in Listing 17.27, which is in accordance with Eq. (17.46).

```
    // Re-calculate viscosity
    nut_ = a1_*k_/max(a1_*omega_, b1_*F23()*sqrt(S2));
    nut_.correctBoundaryConditions();
```

Listing 17.27 Updating the turbulent eddy viscosity

The Menter reliability constraint is applied to the eddy viscosity where the constant b1_ takes the value of 1.0, while the F23 function, described in Listing 17.28, returns by default the proper F2 function defined by Eq. (17.47).

```
tmp<volScalarField> kOmegaSST::F23() const
{
    tmp<volScalarField> f23(F2());
    if (F3_)
    {
        f23() *= F3();
    }
    return f23;
}
```

Listing 17.28 The F23 function definition

Finally the eddy viscosity is corrected at wall boundaries according to Eq. (17.77) using the same procedure as in the kEspilon class in which the Spalding wall function is applied by the class "nutkWallFunctionFvPatchScalarField".

17.10.3 simpleFoamTurbulent

The simpleFoamTurbulent solver is an extension, for turbulent flow simulations, of the simpleFoamImproved solver described in Chap. 15. As previously described, turbulence affects the diffusion terms of the transport equations. Its effects are embodied into the equations through the effective viscosity, which is the sum of both laminar and eddy viscosities.

In order to include turbulence in the OpenFOAM® solver, the virtual base class *RASModel* is invoked and few modifications are introduced. The first one is to substitute the constant laminar transport properties with the variable turbulent ones

through the turbulence model. Therefore in the "*createFields.H*" file, the object turbulence of type RASModel is instantiated, as shown in Listing 17.29.

```
Info<< "Reading transportProperties\n" << endl;

singlePhaseTransportModel laminarTransport(U,mdotf);

autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, mdotf , laminarTransport)
);
```

Listing 17.29 Turbulence model definition

For the definition of the RASModel, it is necessary to activate the *single PhaseTransportModel* class that defines the general transport model for the laminar viscosity (from dictionary it could be set as constant or as function of temperature like in the Sutherland's model), the velocity, and the mass flux. As described earlier, these quantities are necessary to define the transport equations of the turbulent quantities. To be noticed in Listing 17.29 is the definition of object *turbulence* as "*autoPtr*", which basically can be treated as a standard pointer in C^{++}.

The second main modification, as depicted in Listing 17.30, is to the momentum equation where diffusion is now computed using the *divDevReff(U) term* previously described.

```
fvVectorMatrix UEqn
(
        fvm::ddt(U)
      + fvm::div(mdotf, U)
      + fvm::SuSp(-fvc::div(mdotf),U)
      + turbulence->divDevReff(U)
);
```

Listing 17.30 Momentum equation detail

The last main modification displayed in Listing 17.31 is in the main solver file shown below.

```
…
#       include "UEqn.H"
#       include "ppEqn.H"

        turbulence->correct();
…
```

Listing 17.31 Main file modification: turbulence model solution

The addition of the statement *turbulence->correct()* activates, each time it is called, the solution of the turbulence model equations allowing the calculation of the eddy viscosity, which is used in the momentum equations.

17.11 Closure

The extra step needed to model incompressible turbulent flows was introduced. The $k-\varepsilon$, $k-\omega$, and some of their variants were discussed. Modeling of the near wall region using wall functions was detailed. This concludes the developments intended to be discussed in this book. The next chapter will discuss the implementation of boundary conditions in OpenFOAM® and uFVM.

17.12 Exercises

Exercise 1
Given the following turbulent intensities and integral length scales, calculate the corresponding $k-\omega-\varepsilon$ values (Table 17.1):

Table 17.1 Data for exercise 1

I	0.01	0.05	0.1	0.25	0.5
ℓ	0.0001	0.1	0.2	1	10
k					
ε					
ω					

Exercise 2
Given the following turbulent intensities and viscosity ratios calculate the corresponding $k-\omega-\varepsilon$ values (Table 17.2):

Table 17.2 Data for exercise 2

I	0.01	0.05	0.1	0.25	0.5
μ_t/μ	0.1	1	10	100	1000
k					
ε					
ω					

Exercise 3
Modify the standard $k-\varepsilon$ turbulence model described above to include the realizability constraint described in Eqs. (17.29–17.31).

Exercise 4
Formulate the realizability constraint of the standard $k-\omega$ turbulence model.

Exercise 5
Starting with the BSL $k-\omega$ model, show that, if the F_1 function is identically one, the ω-equation corresponds, with minor simplifications, to the standard ε equation of the $k-\varepsilon$ turbulence model.

Exercise 6
Using a Newton-Raphson linearization (i.e., $y(x) \approx y(0) + xy'(x)$), formulate the diagonal and the source term coefficients of Eq. (17.63).

Exercise 7
Formulate Eqs. (17.83) and (17.84) based on turbulent dimensionless quantities, i.e.,

$$\varepsilon = \varepsilon(u^*, d^+, k, \nu)$$
$$\omega = \omega(u^*, d^+)$$

Exercise 8
Develop an OpenFOAM® application that implements normal distance to the wall using Eqs. (17.119), (17.120), (17.121).

Exercise 9
Experiments in roughened surfaces indicate that near rough walls the classical logarithmic law of the wall has a different intercept. This is due to a higher wall shear stress, which shifts the logarithmic velocity profile downward. With the assumption that the following formula is applicable:

$$u^+ = \frac{1}{\kappa} Ln(d^+) + B - \Delta B$$

where $\Delta B = \frac{1}{\kappa} Ln(1 + h_s^+)$ (h_s^+ is the equivalent sand grain roughness), find an equivalent d_{eff}^+ formulation, valid for both smooth and rough walls, allowing the logarithmic law of the wall to be re-expressed in its classical form, i.e.,

$$u^+ = \frac{1}{\kappa} Ln(d_{eff}^+) + B$$

Exercise 10 (OpenFOAM®)
Using the Doxygen documentation [56], list all derived classes of the base class *RASModel* for incompressible flows (i.e., turbulence models available in OpenFOAM®).

Exercise 11 (OpenFOAM®)
Identify the turbulent quantities that are resolved in each incompressible turbulence model defined in Exercise 10.

Exercise 12 (OpenFOAM®)
Using the Doxygen documentation, list all derived classes of the base class *nutWallFunctionFvPatchScalarField.*

Exercise 13 (OpenFOAM®)
The base class *nutWallFunctionFvPatchScalarField* defines an additional virtual base function *virtual tmp<scalarField>yPlus () const = 0.* Describe the function definition for each of the derived classes, commenting on the differences.

Exercise 14 (OpenFOAM®)
Compare the implementation in "FOAM_SRC/turbulenceModels/incompressible/RAS/derivedFvPatchFields/wallFunctions/epsilonWallFunctions/epsilonLowReWallFunction" with the Low Reynolds number models formulation given by Eq. (17.98).

References

1. Tennekes H, Lumley JL (1972) A first course in turbulence. MIT Press, Cambridge. ISBN 978-0-262-20019-6
2. Kolmogorov AN (1941) The local structure of turbulence in incompressible viscous fluids at very large Reynolds numbers. Doklady AN SSSR 30:299–303
3. Kolmogorov AN (1941) Dissipation of energy in isotropic turbulence. Dokl Akad Nauk SSSR 32:19–21.é
4. Moser RD, Kim J, Mansour NN (1999) Direct numerical simulation of turbulent channel flow up to $Re_\tau = 590$. Phys Fluids 11(4):943–945
5. Scardovelli R, Zaleski S (1999) Direct numerical simulation of free-surface and interfacial flow. Ann Rev Fluid Mech. 31:567–603
6. Le H, Moin P, Kim J (1997) Direct numerical simulation of turbulent flow over a backward-facing step. J Fluid Mech 330:349–374
7. Choi H, Moin P, Kim J (1993) Direct numerical simulation of turbulent flow over Riblets. J Fluid Mech 255:503–539
8. Leonard A (1974) Energy cascade in large-eddy simulations of turbulent fluid flows. Adv Geophys A 18:237–248
9. Sagaut P (2006) Large eddy simulation for incompressible flows-an introduction. Springer, Berlin
10. Ferziger JH (1995) Large eddy simulation. In: Hussaini MY, Gatski T (eds) Simulation and modeling of turbulent flows. Cambridge University Press, New York
11. Nieuwstadt FTM, Mason PJ, Moeng C-H, Schuman U (1991) Large eddy simulation of the convective boundary layer: a comparison of four computer codes. In: Durst F et al (eds) Turbulent shear flows, 8th edn. Springer, Berlin
12. Reynolds O (1895) On the dynamical theory of incompressible viscous fluids and the determination of the criterion. Philos Trans Royal Soc London A 186:123–164
13. Favre A (1965) Equations des Gas Turbulents Compressibles. Journal de Mecanique 4 (3):361–390
14. Boussinesq J (1877) Essai sur la théorie des eaux courantes. Mémoires présentés par divers savants à l'Académie des Sciences 23(1):1–680
15. Schlichting H (1968) *Boundary-layer theory,* 6th edn. Chapter XIX. McGraw Hill
16. Schmitt FG (2007) About Boussinesq's turbulent viscosity hypothesis: historical remarks and a direct evaluation of its validity. Comptes Rendus Mécanique 335(9 and 10):617–627
17. Prandtl L (1925) Uber die ausgebildete Turbulenz. ZAMM 5:136–139

18. Baldwin BS, Lomax H (1978) Thin-Layer approximation and algebraic model for separated turbulent flows. AIAA Paper, Huntsville, pp 78–257
19. Cebeci T, Smith AMO (1974) Analysis of turbulent boundary layers. Ser Appl Math Mech, vol XV, Academic Press, Waltham
20. Baldwin BS, Barth TJ (1990) A one-equation turbulence transport model for high reynolds number wall-bounded flows. NASA TM-102847
21. Goldberg UC (1991) Derivation and testing of a one-equation model based on two time scales. AIAA J 29(8):1337–1340
22. Spalart PR, Allmaras SR (1992) A one-equation turbulence model for aerodynamic flows. AIAA Paper, Reno, pp 92–439
23. Jones WP, Launder BE (1972) The prediction of laminarization with a two-equation model of turbulence. Int J Heat Mass Transf 15:301–314
24. Launder BE, Sharma BI (1974) Application of the energy dissipation model of turbulence to the calculation of flow near a spinning disk. Lett Heat Mass Transfer 1(2):131–138
25. Chien K-Y (1982) Predictions of channel and boundary-layer flows with a low-reynolds-number turbulence model. AIAA J 20(1):33–38
26. Myong HK, Kasagi N (1990) A new approach to the improvement of k-ε turbulence model for wall-bounded shear flows. JSME Int J 33:63–72
27. Kolmogorov AN (1942) Equations of turbulent motion of an incompressible fluid. Izvestia Acad Sci USSR Phys 6(1 and 2):56–58
28. Wilcox D (1988) Reassessment of the scale-determining equation for advanced turbulence models. AIAA J 26(11):1299–1310
29. Wilcox DC (1998) Turbulence modeling for CFD, 2nd edn. DCW Industries, US
30. Menter FR (1992) Influence of freestream values on $k - \omega$ turbulence model predictions. AIAA J 30(6):1657–1659
31. Menter FR (1993) Zonal two-equation k − ω turbulence model for aerodynamic flows. AIAA Paper, Orlando, pp 1993–2906
32. Menter F (1994) Two-equation eddy-viscosity turbulence models for engineering applications. AIAA J 32(8):1598–1605
33. Menter FR, Kuntz M, Langtry R (2003) Ten years of industrial experience with the SST turbulence model, 4th edn. Turbulence, Heat and Mass Transfer, Antalya, pp 73–86
34. Menter FR, Carregal Ferreira J, Esch T, Konno B (2003) The SST turbulence model with improved wall treatment for heat transfer predictions in gas turbines. In: Proceedings of the international gas turbine congress, Tokyo, IGTC2003-TS-059
35. Menter FR (2009) Review of the shear-stress transport turbulence model experience from an industrial perspective. Int J Comput Fluid Dyn 23(4):305–316
36. Daky BJ, Harlow FH (1970) Transport equations in turbulence. Phys Fluids 13:2634–2649
37. Fu S, Launder BE, Tselepidakis DP (1987) Accommodating the effects of high strain rates in modelling the pressure-strain correlation. Report no. TFD/87/5, Mechanical Engineering Department, Manchester Institute of Science and Technology, England
38. Gibson MM, Launder BE (1986) Ground effects on pressure fluctuations in the atmospheric boundary layer. J Fluid Mech 86(Pt. 3):491–511
39. Gibson MM, Younis BA (1986) Calculation of swirling jets with a reynolds stress closure. Phys Fluids 29:38–48
40. Wilcox DC, Rubesin MW (1980) Progress in turbulence modeling for complex flow fields including effects of compressibility. NASA TP-1517
41. Wilcox DC (1988) Multiscale model for turbulent flows. AIAA J 26(11):1311–1320
42. Patel VC, Rodi W, Scheuerer G (1985) Turbulence models for near-wall and low reynolds number flows: a review. AIAA J 23(9):1308–1319
43. Medic G, Durbin PA (2002) Toward improved prediction of heat transfer on turbine blades. ASME J Turbomach 124(2):187–192
44. Sahay A, Sreenivasan KR (1999) The wall-normal position in pipe and channel flows at which viscous and turbulent shear stresses are equal. Phys Fluids 11(10):3186–3188

45. Bredberg J (2000) On the wall boundary condition for turbulence models. Department of Thermo and Fluid Dynamics, Chalmers University of Technology, Internal report 00/4, Goteborg
46. Launder BE, Spalding DB (1974) The numerical computation of turbulent flows. Comput Methods Appl Mech Eng 3:269–289
47. Grotjans H, Menter F(1998) Wall function for general application cfd codes. In: Computational fluid dynamics 1998, Proceedings fourth European CFD Conference ECCOMAS, Wiley, Chichester
48. Menter F, Esch T (2001) Elements of industrial heat transfer prediction. In: Proceedings 16th Brazilian congress of mechanical engineering (COBEM), pp 117–127
49. Kader BA (1981) Temperature and concentration profiles in fully turbulent boundary layers. Int J Heat Mass Transf 24:1541–1544
50. Tucker PG (2003) Differential equation-based wall distance computation for DES and RANS. J Comput Phys 190:229–248
51. Sethian JA (1999) Fast marching methods. SIAM Rev 41(2):199–235
52. Tucker PG, Rumsey CL, Spalart PR, Bartels RE, Biedron RT (2004) Computations of wall distances based on differential equations. AIAA Paper 2004–2232
53. Xu J-L, Yan C, Fan J-J (2011) Computations of wall distances by solving a transport equation. Appl Math Mech 32(2):141–150
54. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
55. Hellsten A (1998) Some improvements in menter's k-omega-SST turbulence model. In: 29th AIAA fluid dynamics conference, AIAA-98-2554
56. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/

Chapter 18
Boundary Conditions in OpenFOAM® and uFVM

Abstract This chapter reviews the implementation of boundary conditions in OpenFOAM® and uFVM. Details on the data structure needed for their implementation are presented along with information on how to add new boundary conditions. The procedure is illustrated through the implementation of the no-slip wall boundary condition. This is shown to differ from the Dirichlet type currently implemented in OpenFOAM®.

18.1 Boundary Conditions in OpenFOAM®

Each boundary condition has a physical meaning described mathematically via an equation, which in the context of a numerical method has to be translated into an algebraic relation. An *inlet* boundary condition for instance, describes a known flow behavior where velocity and pressure satisfy specified physical conditions expressed using proper mathematical equations. These include a Dirichlet and a Neumann condition, which should be defined in order to connect the mathematical model with the boundary conditions of the problem. The implementation of these conditions will affect the mathematical operator or term to which they apply (i.e., divergence, laplacian, gradient, etc.).

In OpenFOAM® [1] almost all definitions of boundary conditions are stored in the following directory (Listing 18.1):

```
src/finiteVolume/fields/fvPatchFields
```

Listing 18.1 OpenFOAM® directory where boundary condition definitions are stored

with the main implemented *types* of boundary conditions stored in the sub-directories listed in Listing 18.2.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_18

```
basic
constraint
derived
fvPatchField
```

Listing 18.2 Sub-directories where the main types of boundary conditions are implemented

A brief description of these sub-directories is given in what follows.

The fvPatchField directory contains the general class definition of a boundary condition, which represents the base class. This class defines the main functions and data structures that will be used and will be inherited by the genuine classes.

The *basic* directory contains the basic mathematically defined boundary conditions. These are the Dirichlet type (*fixedValue*), the Neuman type (*zeroGradient* and *fixedGradient*), and the Robin type (*mixed*) boundary conditions. One additional entry included in *basic* is the *coupled* boundary condition that implements a patch to patch type condition, i.e., coupling two boundary patches together (coupled boundaries).

The *constraint* directory contains geometric type boundary conditions that derive from the *coupled* boundary class. An example is the periodicity boundary condition depicted schematically in Fig. 18.1. In this case each cell is related to the cell of the corresponding patch allowing the boundary cells to be treated as internal ones.

Finally the *derived* directory includes all boundary conditions that are derived from the basic Dirichlet, Neumann, and Robin boundary conditions. These derived boundary conditions are simply specializations of the basic types.

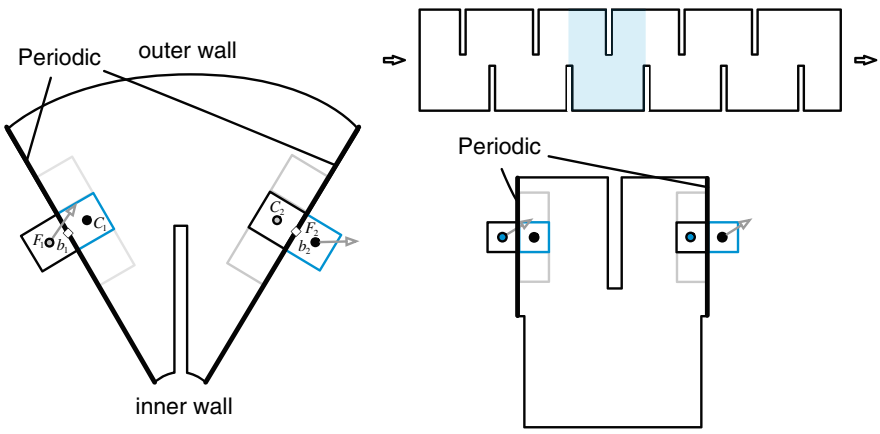


Fig. 18.1 A schematic of a periodic boundary condition

18.2 Boundary Condition Customization

To write a new boundary condition it is essential to understand the role of five main functions, namely, *updateCoeffs, valueInternalCoeffs, valueBoundaryCoeffs, gradientInternalCoeffs,* and *gradientBoundaryCoeffs.*

updateCoeffs: This member function is responsible for the explicit update of the values at the boundary face centers. The function is called whenever the patch field values need to be updated iteratively. For example in the *totalTemperature FvPatchScalarField* class, *updateCoeffs* is used to compute, as shown in Listing 18.3, the static temperature from the specified Total Temperature value according to the relation $T = T_0 - 0.5(\gamma - 1)U^2/(\gamma R)$.

```
void Foam::totalTemperatureFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
    const fvPatchVectorField& Up =
        patch().lookupPatchField<volVectorField, vector>(UName_);

    const fvPatchField<scalar>& psip =
        patch().lookupPatchField<volScalarField, scalar>(psiName_);

    scalar gM1ByG = (gamma_ - 1.0)/gamma_;
    operator==

    (
        T0_/(1.0 + 0.5*psip*gM1ByG*magSqr(Up))
    );

    fixedValueFvPatchScalarField::updateCoeffs();
}
```

Listing 18.3 Script used to iteratively update the static temperature from total temperature using the *updateCoeffs* function

In Listing 18.3 the *operator* == assigns the computed value to the static temperature.

Another example is setting a mean value to an entire patch. The idea is to impose an average value at all faces of a boundary patch based on the average of the values at the centroids of their associated boundary cells and a specified mean value. The required *updateCoeffs* function is written as shown in Listing 18.4.

```
template<class Type>
void fixedMeanFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    Field<Type> newValues(this->patchInternalField());
    Type meanValuePsi = gSum(this->patch().magSf()*newValues)
       /gSum(this->patch().magSf());
    newValues += (meanValue_ - meanValuePsi);
    this->operator==(newValues);
    fixedValueFvPatchField<Type>::updateCoeffs();
}
```

Listing 18.4 Another example of using the function *updateCoeffs* to update boundary values

Note that the function in this case is a template. Therefore it can be used in conjunction with a variety of types such as scalars, vectors, or tensors.

With *updateCoeffs()* the values at the face centroids are set explicitly as for a Dirichlet condition. However the other functions, *valueInternalCoeffs*, *valueBoundaryCoeffs*, *gradientInternalCoeffs*, and *gradientBoundaryCoeffs* are used to linearize the boundary condition in order to complement the *updateCoeffs()* function. The *valueInternalCoeffs* and *valueBoundaryCoeffs* are generally used to linearize the boundary condition for a divergence operator since for that operator the value at the patch faces is needed. On the other hand, the *gradientInternalCoeffs* and *gradientBoundaryCoeffs* are used to linearize the boundary condition for a laplace type operator since for that operator the gradient at the patch faces is needed. This is summarized in Table 18.1.

To clarify the above, the implementations of two boundary conditions are considered. The first is the Neuman (zeroFlux) boundary condition, while the second is the Dirichlet (specifiedValue) boundary condition.

For a Neumann (zeroFlux) boundary condition and assuming an orthogonal mesh, the value of the boundary patch is equal to the value of the boundary element since the normal gradient should be equal to zero. For the divergence term where the boundary patch value will be needed, the specified value at the boundary patch is written in term of the *valueInternalCoeffs* and *valueBoundaryCoeffs*, which represent the linearization of the boundary element value and its non-linearizable

Table 18.1 A summary of the coefficients used to linearize boundary conditions

	Diagonal Coeff	Source term
Divergence	valueInternalCoeffs	valueBoundaryCoeffs
Laplacian	gradientInternalCoeffs	gradientBoundaryCoeffs

part, respectively. For example, the boundary value of a zero gradient boundary condition can be written as

$$\begin{aligned}\phi_b &= \mathit{FluxCb}\,\phi_C + \mathit{FluxVb} \\ &= \mathit{valueInternalCoeffs}\,\phi_C + \mathit{valueBoundaryCoeffs} \\ &= 1\,\phi_C + 0\end{aligned} \tag{18.1}$$

The syntax used to implement the zeroFlux condition for the divergence operator is shown in Listing 18.5.

```
template<class Type>
tmp<Field<Type> > zeroGradientFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::one)
    );
}
template<class Type>
tmp<Field<Type> > zeroGradientFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
```

Listing 18.5 Implementation of the zeroFlux boundary condition for the divergence operator

For the laplacian operator, the gradient at the boundary is needed. In this case the gradient at the boundary patch is set to zero (zeroFlux). This is done through the use of the gradient linearization, which is written as

$$\begin{aligned}\nabla\phi_b &= \mathit{gradientInternalCoeffs}\,\phi_C + \mathit{gradientBoundaryCoeffs} \\ &= 0\,\phi_C + 0\end{aligned} \tag{18.2}$$

where now the *gradientInternalCoeffs* is the linearized coefficient and *gradientBoundaryCoeffs* is the non-linearized component of the gradient as shown in Listing 18.6.

```
template<class Type>
tmp<Field<Type> >
zeroGradientFvPatchField<Type>::gradientInternalCoeffs() const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}


template<class Type>
tmp<Field<Type> >
zeroGradientFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
```

Listing 18.6 Implementation of the zeroFlux boundary condition for the laplacian operator

Table 18.2 A summary of the coefficients for a zeroFlux boundary condition

Neumann (zero order)	Diagonal Coeff	Source term
Divergence	Value(1)	0
Laplacian	0	0

A summary of the value of the coefficients used to implement a zeroFlux boundary condition is given in Table 18.2.

In the above, the value at the boundary is set equal to the value of the boundary element since a zero flux condition is specified.

For a Dirichlet boundary condition, the contribution to the matrix of coefficients will be just a source term on the right hand side of the equations. In this case the boundary condition does not alter the diagonal. The *valueInternalCoeffs* and *valueBoundaryCoeffs* are defined inside OpenFOAM® as

$$\begin{aligned} \phi_b &= FluxCb\,\phi_C + FluxVb \\ &= valueInternalCoeffs\,\phi_C + valueBoundaryCoeffs \\ &= 0\,\phi_C + \phi_{specified} \end{aligned} \tag{18.3}$$

For the laplacian operator, the gradient at the boundary is based on the Dirichlet value. In this case the gradient at the boundary patch is set again through the use of the gradient linearization, which is written as

Table 18.3 A summary of the coefficients for a Dirichlet boundary condition

Dirichlet	Diagonal Coeff	Source term
Divergence	0	Boundary value
Laplacian	Delta	Boundary value and delta

$$\nabla\phi_b = gradientInternalCoeffs\,\phi_C + gradientBoundaryCoeffs$$
$$= \frac{-\phi_C + \phi_b}{d} = (-\phi_C + \phi_b)delta = -\phi_C\,delta + \phi_b delta$$

A summary of the coefficient values is given in Table 18.3, while the template code is shown in Listing 18.7.

```
template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return *this;
}
…
template<class Type>
tmp<Field<Type> >
fixedValueFvPatchField<Type>::gradientInternalCoeffs() const
{
    return -pTraits<Type>::one*this->patch().deltaCoeffs();
}

template<class Type>
tmp<Field<Type> >
fixedValueFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return this->patch().deltaCoeffs()*(*this);
}
```

Listing 18.7 Syntax used for the implementation of the Dirichlet boundary condition

18.3 Development of a New BC: No Slip Wall Condition

The task is to define a new boundary condition type for the proper implementation of the no slip condition in the context of the finite volume discretization in OpenFOAM® (Fig. 18.2).

The no slip condition is a fundamental boundary condition in solving flow problems. According to Newton's law, the shear stress experienced by a viscous fluid flowing past a wall is proportional to the normal gradient of the velocity parallel to the wall and is expressed mathematically as

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} (1-n_x^2) & -n_y n_x \\ -n_y n_x & (1-n_y^2) \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.4}$$

where C refers to values at the centroid of the boundary cell. This equation suggests that the no slip condition is anisotropic because it is function only of the velocity component parallel to the wall and the normal distance to the wall. For the case when the wall is aligned with the x-axis, only the x-component of velocity is expected to affect the shear stress value. In fact from Eq. (18.4) the shear stress equation becomes

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.5}$$

where it is evident that only the diagonal coefficient of the x-component of velocity has to be injected into the matrix.

A common simplification for this boundary condition, used in OpenFOAM®, is its treatment as a Dirichlet boundary condition, i.e., a *fixedValue* (*0 0 0*) boundary condition.

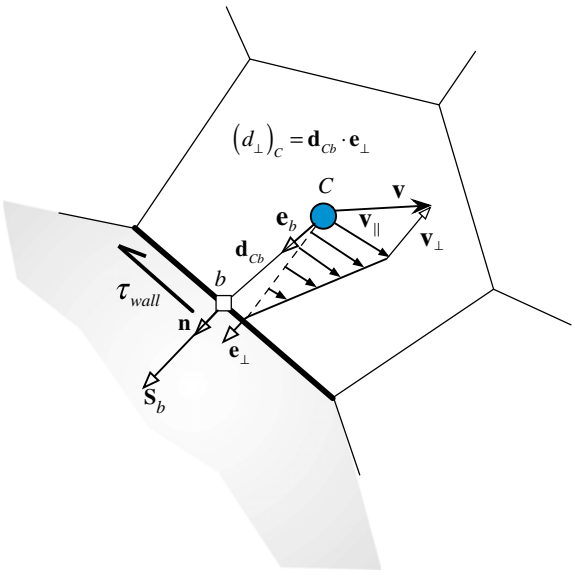


Fig. 18.2 No-slip wall shear stress

Using a Dirichlet condition for the implementation of the no-slip condition at a wall introduces an important error. This error is due to treating the shear stress distribution as isotropic with its value influenced by the velocity component (of the internal cells) normal to the wall. In fact using a Dirichlet boundary condition for the case when the wall is parallel to the x-velocity component, results in a shear stress evaluated as

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\mu \frac{\partial \mathbf{v}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.6}$$

which, when compared with Eq. (18.2), clearly demonstrates the unphysical dependence of the wall shear stress on the y-component of velocity.

In the following, the proper implementation of the no slip wall condition in OpenFOAM® is described. The suggested implementation should always be used instead of the simplified assumption of a Dirichlet boundary condition for momentum discretization at a no-slip wall.

As stated above, the definition of a new boundary condition necessitates redefining the proper functions. A good starting point is to use an existing boundary class, copying it, and modifying it with the new algorithms. For this purpose, the *fixedValue class* can be used and altered for vector type. The .H file defining the new virtual no slip wall class is depicted in Listing 18.8.

```
namespace Foam
{
/*---------------------------------------------------------------------------*
\
                Class noSlipWallFvPatchVectorField Declaration
\*---------------------------------------------------------------------------
*/
class noSlipWallFvPatchVectorField
:
    public fixedValueFvPatchField<vector>
{
protected:

    // Protected data
public:
    //- Runtime type information
    TypeName("noSlipWall");
.
.
.
//- Return the matrix diagonal coefficients corresponding to the
//  evaluation of the gradient of this patchField
virtual tmp<Field<vector> > gradientInternalCoeffs() const;

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

Listing 18.8 Synopsis of the .H file for the declaration of the noSlipWallFvPatchVectorField class

As noticed, just the *gradientCoeffs* function is redefined, since in the *fixedValue* class the *valueCoeffs* and the *updateCoeffs* functions are already correctly implemented.

Once the declarations of the functions are ready, implementation proceeds with the modifications to the .C file. The starting point is Eq. (18.1) but now written in three dimensions as

$$\begin{aligned}\tau_{wall} &= -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \\ &\simeq \frac{\mu}{(d_{\perp})_C}\mathbf{v}_{||wall} - \frac{\mu}{(d_{\perp})_C}\begin{bmatrix} (1-n_x^2) & -n_y n_x & -n_z n_x \\ -n_y n_x & (1-n_y^2) & -n_z n_y \\ -n_z n_x & -n_z n_y & (1-n_z^2) \end{bmatrix}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix}\end{aligned} \tag{18.7}$$

Equation (18.7) indicates that all elements in the matrix have implicit contributions because they depend on the internal cell value and have to be implemented with the *gradientInternalCoeffs*. The other term is the tangential component of the wall velocity and it depends on the face value only. In this case it is necessary to be described through the function *gradientBoundaryCoeffs*. Moreover, a closer look at the implicit contribution suggests storing its mixed terms on the right hand side of the equation to be treated explicitly since the momentum equations are solved in a segregated fashion. Thus Eq. (18.7) is rewritten as

$$\begin{aligned}\tau_{wall} \simeq &-\frac{\mu}{(d_{\perp})_C}\underbrace{\begin{bmatrix} (1-n_x^2) & 0 & 0 \\ 0 & (1-n_y^2) & 0 \\ 0 & 0 & (1-n_z^2) \end{bmatrix}}_{gradientInternalCoeffs}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix} \\ &-\frac{\mu}{(d_{\perp})_C}\underbrace{\begin{bmatrix} 0 & -n_y n_x & -n_z n_x \\ -n_y n_x & 0 & -n_z n_y \\ -n_z n_x & -n_z n_y & 0 \end{bmatrix}}_{gradientBoundaryCoeffs}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix} + \underbrace{\frac{\mu}{(d_{\perp})_C}\mathbf{v}_{||wall}}_{gradientBoundaryCoeffs}\end{aligned} \tag{18.8}$$

Based on the above formulation, the *gradienInternalCoeffs* function reads as shown in Listing 18.9.

```
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * *
* * * //
tmp<Field<vector> >
noSlipWallFvPatchVectorField::gradientInternalCoeffs() const
{
    vectorField normal = this->patch().nf();
    vectorField impCoeff(this->size(),pTraits<vector>::zero);
    forAll(impCoeff,faceI)
    {
       impCoeff[faceI][0] = (scalar(1.0) - pow(normal[faceI][0],2));
       impCoeff[faceI][1] = (scalar(1.0) - pow(normal[faceI][1],2));
       impCoeff[faceI][2] = (scalar(1.0) - pow(normal[faceI][2],2));
    }

    return -impCoeff*this->patch().deltaCoeffs();
}
```

Listing 18.9 Script used for calculating the implicit contribution

On the other hand, the calculation of the *gradientBoundaryCoeffs* is as shown in Listing 18.10.

```
tmp<Field<vector> >
noSlipWallFvPatchVectorField::gradientBoundaryCoeffs() const
{
    vectorField normal = this->patch().nf();
    vectorField expCoeff(this->size(),pTraits<vector>::zero);
    vectorField boundTanField = *this - ((*this) & normal) * normal;
    vectorField intField = this->patchInternalField();
    forAll(expCoeff,faceI)
    {
        expCoeff[faceI][0] = boundTanField[faceI][0]
                            +normal[faceI][0]*normal[faceI]
                             [1]*intField[faceI][1]
                            +normal[faceI][2]*normal[faceI]
                             [0]*intField[faceI][2];
        expCoeff[faceI][1] = boundTanField[faceI][1]
                            +normal[faceI][1]*normal[faceI]
                             [0]*intField[faceI][0]
                            +normal[faceI][2]*normal[faceI]
                             [1]*intField[faceI][2];
        expCoeff[faceI][2] = boundTanField[faceI][2]
                            +normal[faceI][2]*normal[faceI]
                             [0]*intField[faceI][0]
                            +normal[faceI][1]*normal[faceI]
                             [2]*intField[faceI][1];
    }

    return this->patch().deltaCoeffs()*expCoeff;
}
```

Listing 18.10 Script used for calculating the explicit contribution

The new boundary condition can now be used with any wall by just defining the patch type as *noSlipWall*, as described in Listing 18.11.

```
wall
{
    type            noSlipWall;
    value           uniform (0 0 0);
}
```

Listing 18.11 Script needed to use the no-slip boundary condition at runtime

18.4 The No-Slip Boundary Condition in uFVM

Adding a new boundary condition in uFVM requires that the boundary be implemented for each term to which it can be applied, thus is not as modular as in OpenFOAM®. Still it can be quite straightforward as demonstrated for the no-slip boundary condition.

It is important to emphasize that the no-slip boundary condition is somewhat a hybrid condition where a flux (the shear stress) has to be computed while ensuring that the boundary velocities are set to specified values, in this case zero. So it relates somewhat to the Dirichlet and Neumann conditions.

In uFVM information about boundary conditions are stored in a patch-based structure composed of the following four arrays:

1. thePatchFlux.FLUXC1f : the linearization coefficient for the owner cell
2. thePatchFlux.FLUXC2f : the linearization coefficient for the neighbor cell
3. thePatchFlux.FLUXVf: the non-linearized part
4. thePatchFlux.FLUXTf: the total flux at the face

in such a manner that the boundary flux is expressed as

$$\text{thePathFlux.FLUXTf} = \text{thePathFlux.FLUXC1f phi\_owner} + \text{thePathFlux.FLUXC2 phi\_boundary} + \text{thePathFlux.FLUXVf} \tag{18.9}$$

Equation (18.9) shows how the total flux at any internal face is linearized in terms of the owner and neighbor elements sharing the face, except that for a patch the neighbor node is basically the boundary node. For a boundary face, where there is no neighbor element defined, thePatchFlux.FLUXC2f is always set to zero.

For the no slip condition where the total flux (i.e., the shear stress) depends on the change of the velocity component parallel to the wall, the expressions of the various contributions become

$$\begin{aligned}
\text{theFluxes.FLUXC1f(iBFaces)} &= \text{area.} * \text{TM.} * (1 - \text{dot}(\text{nc}', \text{nc}')');\\
\text{theFluxes.FLUXC2f(iBFaces)} &= 0;\\
\text{theFluxes.FLUXVf(iBFaces)} &= \text{Fc} - \text{area.} * \text{TM.} * (1 - \text{dot}(\text{nc}', \text{nc}')'). * \text{velc(iOwners)};\\
\text{theFluxes.FLUXTf(iBFaces)} &= \text{theFluxes.FLUXC1f(iBFaces)}. * \text{velc(iOwners)}\\
&\quad + \text{theFluxes.FLUXC2f(iBFaces)}. * \text{velc(iBElements)}\\
&\quad + \text{theFluxes.FLUXVf(iBFaces)};
\end{aligned} \tag{18.10}$$

where F_c is the actual shear flux computed at the patch wall faces as

$$F_c = -\mu \frac{\mathbf{v} \cdot \mathbf{e}_{\parallel}}{d_{\perp}} \tag{18.11}$$

The complete code for the implementation of the boundary condition is shown in Listing 18.12.

```
function theFluxes =
cfdAssembleStressTermWallNoSlipBC(iPatch,theFluxes,theEquationName,the
Term,iComponent)

theMesh = cfdGetMesh;
theFluidTag = cfdGetFluidTag(theEquationName);

theBoundary = theMesh.boundaries(iPatch);
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;
numberOfBFaces = theBoundary.numberOfBFaces;

%
iFaceStart = theBoundary.startFace;
iFaceEnd = iFaceStart+numberOfBFaces-1;
iBFaces = iFaceStart:iFaceEnd;
%
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
iBElements = iElementStart:iElementEnd;

%
%
% specifiy the Term Field
%
if(isempty(theTerm.variableName))
    theTermFieldName = theEquationName;
else
    theTermFieldName = theTerm.variableName;
end
theTermField = cfdGetMeshField(theTermFieldName);
velc = theTermField.phi(:,iComponent);

%phiGradient = theTermField.phiGradient(:,:,iComponent);
```

Listing 18.12 Script used for the implementation of the no-slip boundary condition in uFVM

```
if(iComponent==1)
    e = [1;0;0];
elseif(iComponent==2)
    e = [0;1;0];
elseif(iComponent==3)
    e = [0;0;1];
end
%
% specifiy the Term Coefficient Field
%
theTermCoefficientField = cfdGetMeshField(['Viscosity' theFluidTag]);
visc = theTermCoefficientField.phi;

%---------------------- End Term Info ----------------------%
%
%
geodiff = [theMesh.faces(iBFaces).geoDiff]';
Tf = [theMesh.faces(iBFaces).T]';
area = [theMesh.faces(iBFaces).area]';
n = [theMesh.faces(iBFaces).Sf]'./[area area area];
iOwners = [theMesh.faces(iBFaces).iOwner]';

TM=cfdGetUCoef(iPatch,theFluidTag);

%/////////////////////////////////////////////////////////
vel = theTermField.phi(iOwners,:);
velb =  theTermField.phi(iBElements,:);

mag = dot(vel',n')';

vel_n = [mag mag mag].*n;
vel_t = vel - vel_n;

magb = dot(velb',n')';
velb_n = [magb magb magb].*n;
velb_t = velb - velb_n;

vel_wall = velb_t;
nc = n*e;

Fc = -area.*TM.*((vel_t - vel_wall)*e);
%/////////////////////////////////////////////////////////
theFluxes.FLUXC1f(iBFaces) =   area.*TM.*(1- dot(nc',nc')');
theFluxes.FLUXC2f(iBFaces) = 0;
theFluxes.FLUXVf(iBFaces)  = - Fc - area.*TM.*(1-
dot(nc',nc')').*velc(iOwners);
theFluxes.FLUXTf(iBFaces)  = theFluxes.FLUXC1f(iBFaces) .*
velc(iOwners) + theFluxes.FLUXC2f(iBFaces) .* velc(iBElements) +
theFluxes.FLUXVf(iBFaces);

end
```

Listing 18.12 (continued)

18.5 Closure

The chapter discussed the implementation of boundary conditions in OpenFOAM®. It also presented the needed steps for implementing new boundary conditions in OpenFOAM® by detailing the various required stages for properly adding a no-slip boundary condition. A brief discussion of boundary conditions in uFVM was also presented. The next chapter is devoted to detailing the steps needed to solve a turbulent flow problem in OpenFOAM®.

Reference

1. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org

Chapter 19
An OpenFOAM® Turbulent Flow Application

Abstract The OpenFOAM® solvers and boundary conditions developed in previous chapters are used here to solve a small scale turbulent incompressible flow problem. Details on the case structure, mesh, and solver setup are presented along with information on monitoring convergence and post processing results.

19.1 Introduction

The design of a car body is a very demanding task that necessitates finding a compromise between the stylistic constraint of the brand and consumer taste on one hand, and the engineering requirements of efficient aerodynamics for improved fuel economy on the other. The use of a CFD tool in this situation is critical, especially in investigating the aerodynamic forces, the interplay of viscous flow effects and turbulent boundary layer, the sensitivity of the flow around the vehicle to changes in its shape, and finally the drag coefficient of the car under various operating conditions.

In this chapter the simpleFoamTurbulent solver and the boundary conditions developed and implemented in previous chapters are used to analyze the flow around a well investigated test case, namely the Ahmed bluff body.

19.2 The Ahmed Bluff Body

In automotive applications, the major factor contributing to drag is the wake that develops behind the vehicle, with its prediction representing a difficult task in CFD. This is so because flow separation in turbulent flows is still a challenge to simulate numerically. Nonetheless determining the size of the flow separation zone greatly influences the predicted drag force acting on the body. Thus, accurate simulation of

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_19

the induced vortex flow and of the separation process is essential for correct predictions of aerodynamic efficiency.

Current automobile designs have many complex geometrical features that make them a challenge to model and/or investigate experimentally. Thus for the limited investigation considered here, the Ahmed bluff body [1, 2], which is schematically depicted in Fig. 19.1, is chosen. Figure 19.1a shows the side, front, and top views from which dimensions can be inferred, while Fig. 19.1b presents a three dimensional visualization of the body. Even though it represents simple geometry, Ahmed body allows for a three-dimensional region of separated flow to be developed and different flow phenomena to be studied and compared to experimental data.

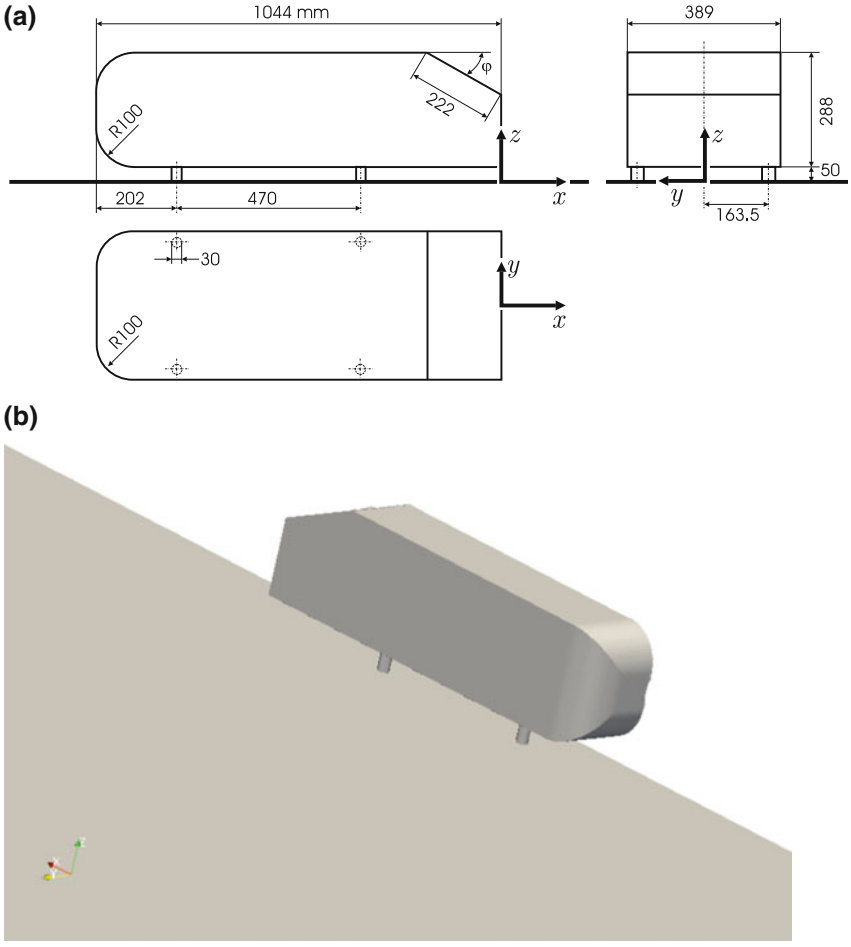


Fig. 19.1 **a** *Side*, *front*, and *top views* of Ahmed body, **b** a three dimensional visualization of Ahmed body geometry

The Ahmed body is a well known configuration and is widely used as benchmark. The slant geometry at its back end generates counter-rotating vortices at the side edges, whose strength is mainly determined by the base slant angle. It comes in two configurations differing in the slant angle value (25° and 35°). In this simulation the configuration with a 25° slant angle is considered.

19.3 Domain Discretization

The computational domain is depicted in Fig. 19.2 with a symmetry condition imposed along the mid section of the Ahmed body shape. Symmetry is exploited to reduce the computational domain and to mitigate the transient flow behavior expected to develop because of vortex shedding at the body's rear end. This should also enhance numerical stability.

Inflow and outflow boundary conditions are placed far from the body to minimize any unwanted interaction with the main flow region, especially between the outlet and the flow at the rear end of the body.

The mesh is generated by *snappyHexMesh*, a utility that is part of the OpenFOAM® [3] package. s*nappyHexMesh* generates three dimensional meshes containing hexahedra (hex) and split-hexahedra (split-hex) elements generated automatically from triangulated surface geometry in the Stereolithography (STL) format. The mesh gradually conforms to the surface by iteratively refining a starting mesh and morphing the resulting split-hex mesh to the surface. An optional phase will shrink back the resulting mesh and insert cell layers. The specification of mesh refinement level is very flexible and the surface handling is robust with a pre-specified final mesh quality. It runs in parallel with a load balancing step at every iteration [4].

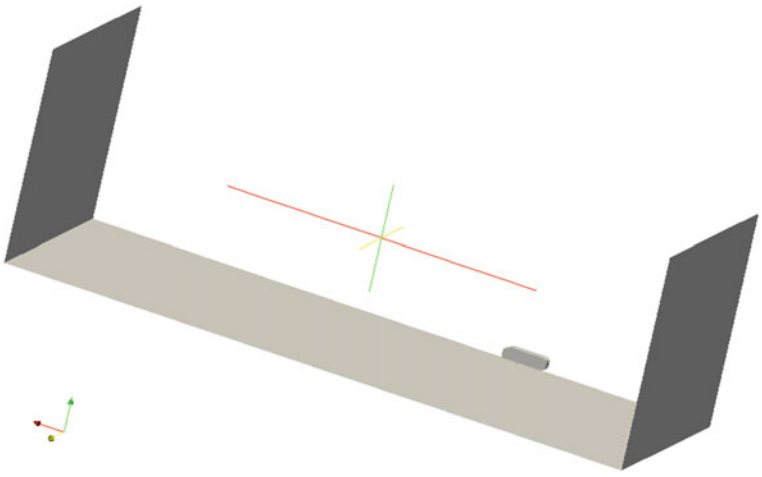


Fig. 19.2 Computational domain for Ahmed body

It is beyond the purpose of this book to give a detailed introduction of the *snappyHexMesh* application but a commented setup file can be found in the case directory "*Ahmed_body/system/snappyHexMeshDict*". Listing 19.1 shows an extract of the file.

```
// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;


// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
    ahmed_body.stl
    {
        type triSurfaceMesh;
        name ahmed_body;
        appendRegionName false;
    }
…

// Settings for the castellatedMesh generation.
castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~~

    // If local number of cells is >= maxLocalCells on any processor
    // switches from from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.
    maxLocalCells 300000;

…
```

Listing 19.1 An extract of the snappyHexMeshDict file

The mesh is generated by executing, from the case directory, the following commands in the shown sequence:

blockMesh—dict system/blockMeshDict this defines the general computational domain within which the snappyHexMesh will operate, the domain should have within its volume the STL geometry.

snappyHexMesh **snappyHexMesh** generates the mesh in three phases with the result of each phase written to a folder (1/2/ and 3/).

mv 3/polyMesh constant/ this moves the final mesh generated by snappyHexMesh into the constant directory to be used for computing the solution.
createPatch—overwrite this deletes any empty patches.

The body fitted grid is composed of about 800,000 polyhedral cells mostly located in two regions around the car body. This allows for a good resolution of the wake region while keeping low the number of elements used in the computational domain. Details of the grid generated around the body is shown in Fig. 19.3.

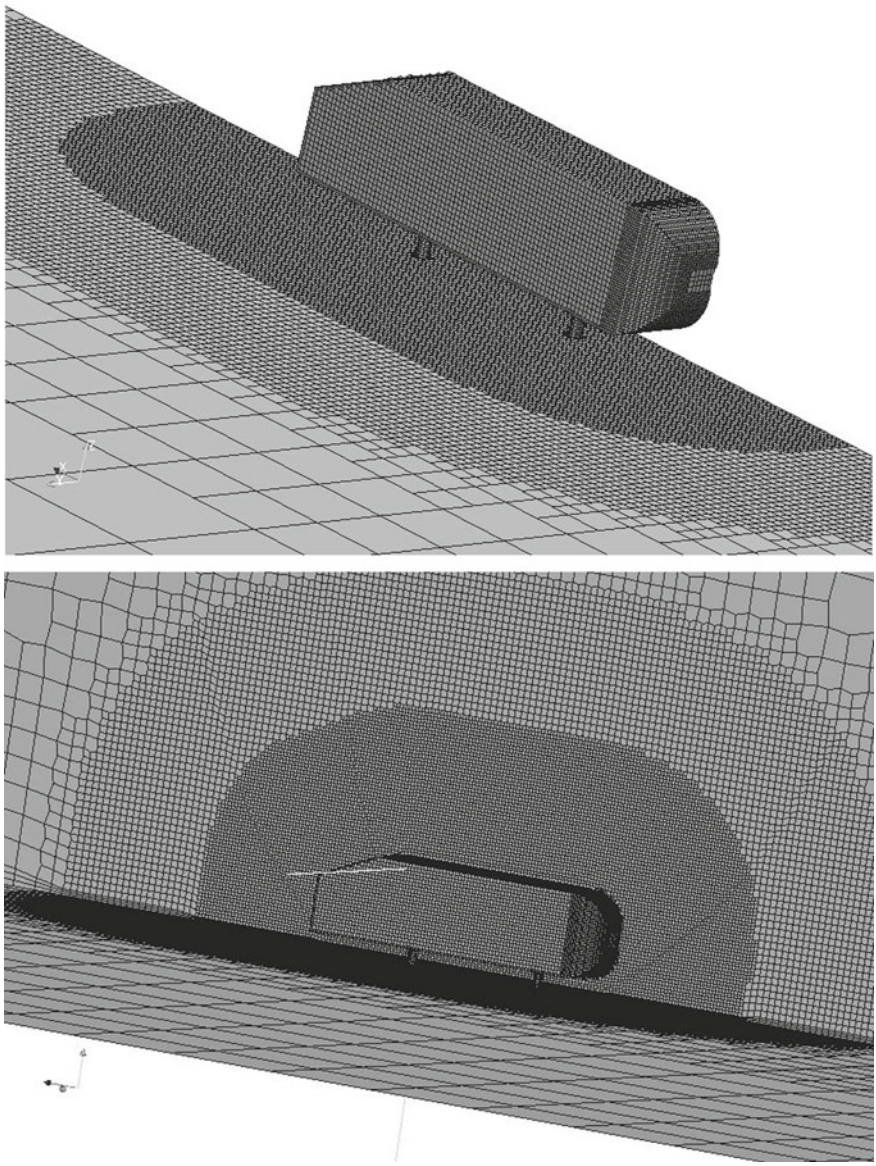


Fig. 19.3 Grid details around Ahmed body

Clustering of mesh elements in the region of interest is a common practice as it reduces the computational cost while properly resolving the important features of the flow.

The generated mesh is fully unstructured and uses polyhedral elements with up to 15 faces. The quality of the mesh is best evaluated using **checkMesh**, the output of such an operation is shown in Listing 19.2.

```
Mesh stats
    points:           806136
    faces:            2331063
    internal faces:   2292117
    cells:            762865
    faces per cell:   6.060285896
    boundary patches: 10
    point zones:      0
    face zones:       0
    cell zones:       0

Overall number of cells of each type:
    hexahedra:     744610
    prisms:        1057
    wedges:        0
    pyramids:      0
    tet wedges:    1
    tetrahedra:    0
    polyhedra:     17197
    Breakdown of polyhedra by number of faces:
        faces   number of cells
            4   82
            5   59
            6   4523
            7   1563
            8   410
            9   6961
           10   5
           11   3
           12   2782
           14   2
           15   807

…

Checking geometry...
    Overall domain bounding box (-5 -0.00189057368 0) (10 5 5)
    Mesh (non-empty, non-wedge) directions (1 1 1)
    Mesh (non-empty) directions (1 1 1)
    Boundary openness (-2.589889552e-17 -1.395011501e-15 -4.461826685e-14) OK.
    Max cell openness = 3.962970386e-16 OK.
    Max aspect ratio = 13.98146561 OK.
    Minimum face area = 1.211564662e-06. Maximum face area = 0.1568483398.  Face area
magnitudes OK.
    Min volume = 8.48251732e-09. Max volume = 0.05873313819.  Total volume = 374.944525.
Cell volumes OK.
    Mesh non-orthogonality Max: 62.12745867 average: 5.023035258
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 2.516852961 OK.
    Coupled point location match (average 0) OK.

Mesh OK.

End
```

Listing 19.2 Grid quality check and details

The physical boundary conditions are set in the *boundary* file located in the "Ahmed_body/constant/polyMesh/" directory. This is displayed in Listing 19.3.

```
10
(
    inlet
    {
        type            patch;
        physicalType    inlet;
        nFaces          298;
        startFace       2292117;
    }
    outlet
    {
        type            patch;
        physicalType    outlet;
        nFaces          298;
        startFace       2292415;
    }
    ffmaxy
    {
        type            patch;
        nFaces          640;
        startFace       2292713;
    }
    floor
    {
        type            wall;
        inGroups        1(wall);
        nFaces          13779;
        startFace       2293353;
    }
    top
    {
        type            patch;
        nFaces          520;
        startFace       2307132;
    }
    ahmed_body_body
    {
        type            wall;
        inGroups        1(wall);
        nFaces          5750;
        startFace       2307652;
    }
    ahmed_body_body_front_h
    {
        type            wall;
        inGroups        1(wall);
        nFaces          864;
        startFace       2313402;
    }
    ahmed_body_body_front_v
    {
        type            wall;
        inGroups        1(wall);
        nFaces          1270;
        startFace       2314266;
    }
    ahmed_body_stilts
    {
        type            wall;
        inGroups        1(wall);
        nFaces          256;
        startFace       2315536;
    }
    cp
    {
        type            wall;
        inGroups        1(wall);
        nFaces          15271;
        startFace       2315792;
    }
)
```

Listing 19.3 Script used to set the physical boundary conditions

The patches *inlet* and *outlet* are defined as type *patch*, while *cp* is set as a *wall*, rather than a *symmetry plane*. In order to reduce the complexity of resolving the flow, a *slip* boundary condition will be used for the *cp* patch. The "*ahmed_body**" patches that represent the car body are defined as "*wall*", while the remaining patches define the external domain.

19.3.1 Initial and Boundary Conditions

The 0 directory must contain the following four basic files for a turbulent incompressible flow simulation: the velocity U, the pressure p, and for the kEpsilon turbulence model, the turbulent kinetic energy k, and the dissipation rate ε.

The file U defines the boundary conditions for the velocity (Listing 19.4).

```
dimensions      [0 1 -1 0 0 0 0];

internalField  uniform ( 40 0 0 );

boundaryField
{
    inlet
    {
        type            surfaceNormalFixedValue;
        refValue        uniform -40;
    }
    outlet
    {
        type            inletOutlet;
        inletValue      uniform ( 0 0 0 );
        value           uniform ( 0 0 0 );
    }
    ffmaxy
    {
        type            slip;
    }
    top
    {
        type            slip;
    }
    ahmed_body_body
    {
        type            noSlipWall;
        value           uniform ( 0 0 0 );
    }
    ahmed_body_body_front_h
    {
        type            noSlipWall;
        value           uniform ( 0 0 0 );
    }
    ahmed_body_body_front_v
    {
        type            noSlipWall;
        value           uniform ( 0 0 0 );
```

Listing 19.4 Script of the ∪ file

```
    }
    ahmed_body_stilts
    {
        type            noSlipWall;
        value           uniform ( 0 0 0 );
    }
    floor
    {
        type            noSlipWall;
        value           uniform ( 0 0 0 );
    }
    cp
    {
        type            slip;

    }
}
```

Listing 19.4 (continued)

The new *noSlipWall* boundary condition is employed to impose the no slip condition at the walls including the body car and the floor. The fluid velocity at inlet is set at 40 m/s. Along the cp plane, a *slip* wall boundary condition is used where the viscous forces are set to zero to mimic a symmetry plane (the slight geometric error due to meshing on patch *cp* does not allow the direct use of a symmetryPlane boundary condition).

The *zeroGradient* pressure boundary condition uses a zero order extrapolation to compute the pressure at the boundary. At the outlet, a Dirichlet boundary condition is applied in order to set a reference pressure (Listing 19.5).

```
boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    ffmaxy
    {
        type            zeroGradient;
    }
…
```

Listing 19.5 Script of the p file

For the kEpsilon model, the turbulent intensity and integral length scale are set at the inlet boundary, while the standard wall functions are employed along all the walls automatically (Chap. 17). The k file is shown in Listing (19.6) and the epsilon file in Listing (19.7).

```
boundaryField
{
    inlet
    {
        type            turbulentIntensityKineticEnergyInlet;
        intensity       0.01;
        value           uniform 0.002;
    }
    outlet
    {
        type            zeroGradient;
    }
…
```

Listing 19.6 Script of the k file

```
boundaryField
{
    inlet
    {
        type            turbulentMixingLengthFrequencyInlet;
        mixingLength    0.01;
        phi             phi;
        k               k;
        value           uniform 0.002;
    }
    outlet
    {
        type            zeroGradient;
    }
…
```

Listing 19.7 Script of the epsilon file

19.3.2 Systems Files

The *controlDict* file (Listing 19.8) is configured to perform 500 SIMPLE iterations. Further, in order to use the new *noSlipWall* boundary condition for the velocity, the corresponding library has to be included together with the proper libraries for inlet turbulence boundary conditions. These are set using the *libs* declaration. A specific run-time post processing configuration is also included to monitor iteratively

variations in the lift and drag coefficients, which are set using the *functions* declaration. Usually for external aerodynamic applications it is more convenient to monitor loads or forces on the body, rather than the level of residuals, for checking convergence.

```
startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         500;

…

libs (
"libNoSlip.so"
"libincompressibleRASModels.so"
"libincompressibleRASModels.so"
);

functions
(

    forceCoeffs1
    {
        type        forceCoeffs;
        functionObjectLibs ( "libforces.so" );
        outputControl timeStep;
        outputInterval  1;
        log         yes;

        patches     ( "ahmed_body*" );
        pName       p;
        UName       U;
        rhoName     rhoInf;      // Indicates incompressible
        log         true;
        rhoInf      1;           // Redundant for incompressible
        liftDir     (0 0 1);
        dragDir     (1 0 0);
        CofR        (0.72 0 0);  // Axle midpoint on ground
        pitchAxis   (0 1 0);
        magUInf     40;
        lRef        1.45;        // Wheelbase length
        Aref        2.618;        // Estimated
    }

)
```

Listing 19.8 Script of the controlDict file

Linear solvers specifications and relaxation factors are configured in the *fvSolution* file displayed in Listing 19.9. Here preconditioned conjugate gradient methods with incomplete factorization pre-conditioners are chosen for all equations. For the pressure correction equation (*pp*), a symmetric matrix solver is specified due to the laplacian nature of the equation. It is worth mentioning that with a pressure correction formulation there is no need for non-orthogonal correction iterations.

```
solvers
{
    pp
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.01;
    }

    "(U|k|epsilon)"
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-15;
        relTol          0.1;
    }
}
SIMPLE
{
    nNonOrthogonalCorrectors 0;
}
relaxationFactors
{
        pp              0.3;
        U               0.7;
        k               0.7;
        epsilon         0.7;
}
```

Listing 19.9 Script of the fvSolution file

Spatial discretization settings are configured in the *fvSchemes* file depicted in Listing 19.10. For this simulation a first order upwind discretization for convection, a Gaussian method for gradient reconstruction, and a linear profile for variable face interpolation are used.

```
ddtSchemes
{
    default         steadyState;
}
gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;
}
divSchemes
{
    default          Gauss upwind;
    div(phi,U)       Gauss upwind;
    div(phi,k)       Gauss upwind;
    div(phi,epsilon)  Gauss upwind;
    div(R)          Gauss linear;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
laplacianSchemes
{
    default         Gauss linear corrected;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon) Gauss linear corrected;
    laplacian(DREff,R) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}
snGradSchemes
{
    default         corrected;
}
fluxRequired
{
    default         no;
    pp               ;
}
```

Listing 19.10 fvSchemes file details

19.3.3 Running the Solver

At the start of the run, the solver provides information about equation residuals and force coefficients (Listing 19.11). Based on the controlDict setup, force coefficients are also written in the file Ahmed_body/postProcessing/forceCoeffs1/0/forceCoeffs.dat that can be used for checking the convergence history.

```
Starting time loop

Time = 1

DILUPBiCG:  Solving for Ux, Initial residual = 1, Final residual = 0.02308834211, No
Iterations 2
DILUPBiCG:  Solving for Uy, Initial residual = 1, Final residual = 0.07259752325, No
Iterations 1
DILUPBiCG:  Solving for Uz, Initial residual = 1, Final residual = 0.05706016776, No
Iterations 1
DICPCG:  Solving for pp, Initial residual = 1, Final residual = 0.009688743115, No
Iterations 140
DILUPBiCG:  Solving for epsilon, Initial residual = 0.999533875, Final residual =
2.962105429e-05, No Iterations 1
DILUPBiCG:  Solving for k, Initial residual = 1, Final residual = 0.000587089458, No
Iterations 1
ExecutionTime = 9.73 s  ClockTime = 9 s

forceCoeffs output:
    Cm    = 5.971060664
    Cd    = 18.59033988
    Cl    = 3.888152128
    Cl(f) = 7.915136728
    Cl(r) = -4.0269846
```

Listing 19.11 Solver verbosity

It is worth noting that for the pressure correction equation the residual at any iteration is always 1, which is due to the way OpenFOAM® normalizes residuals (Check the computational pointers in Chap. 10).

Figure 19.4a shows the convergence history plots of the residuals of the momentum equations, while Fig. 19.4b shows the changes in the lift and drag coefficients as the simulation progresses. Results clearly show that in 300 iterations the solution has completely converged with changes in the lift and drag coefficient values becoming negligible.

Figure 19.5 shows the static pressure contours around the body. High pressure values are evident on the front of the body due to the recovery of the dynamic pressure contribution. Contours also show a low pressure region immediately

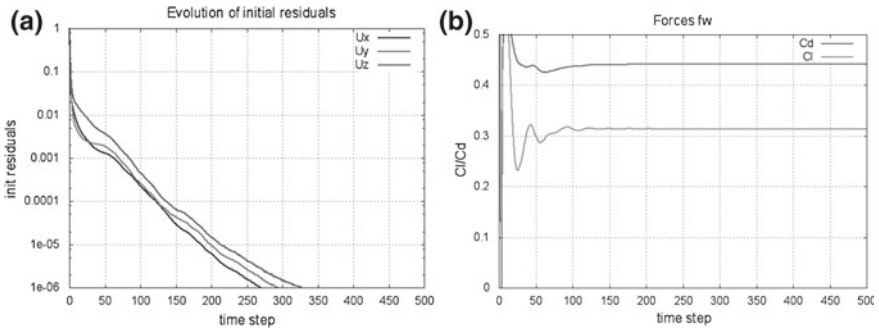


Fig. 19.4 **a** Convergence history plots of the residuals of the momentum equations, **b** variations of the drag and lift coefficients with time

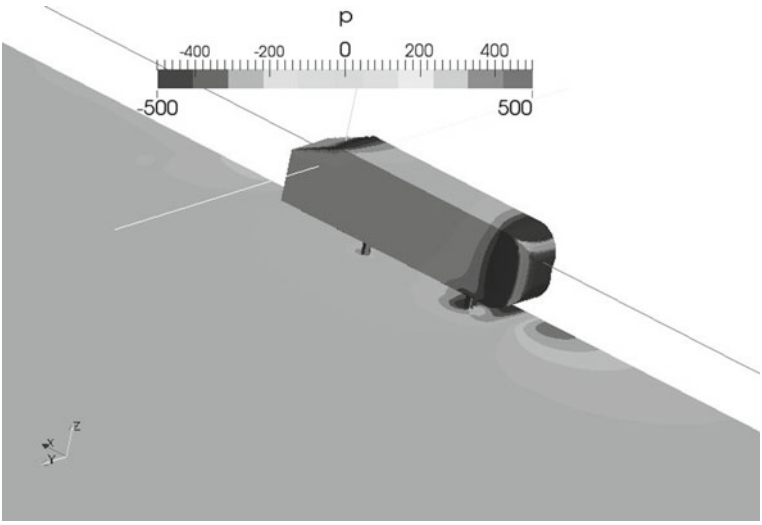


Fig. 19.5 Pressure contours around the body

downstream the front location of the body as a result of the acceleration of the fluid on a curved surface.

In Fig. 19.6 the recirculation region is evident on the back side of the body where a low speed region appears.

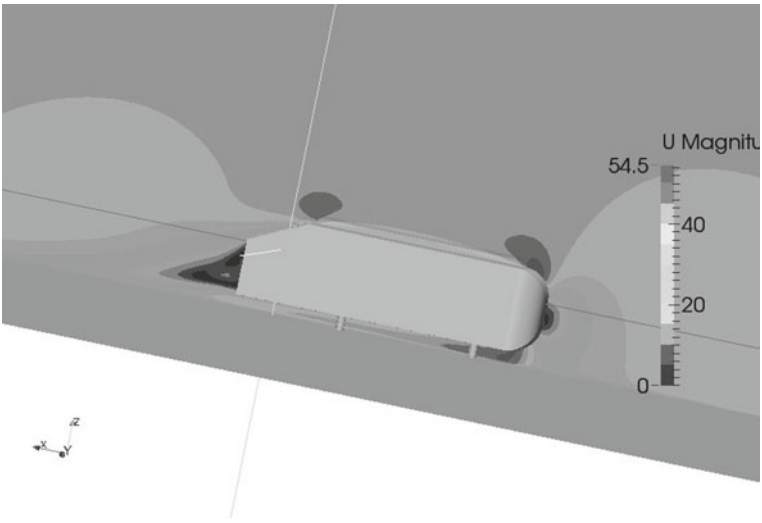


Fig. 19.6 Velocity contours around the body

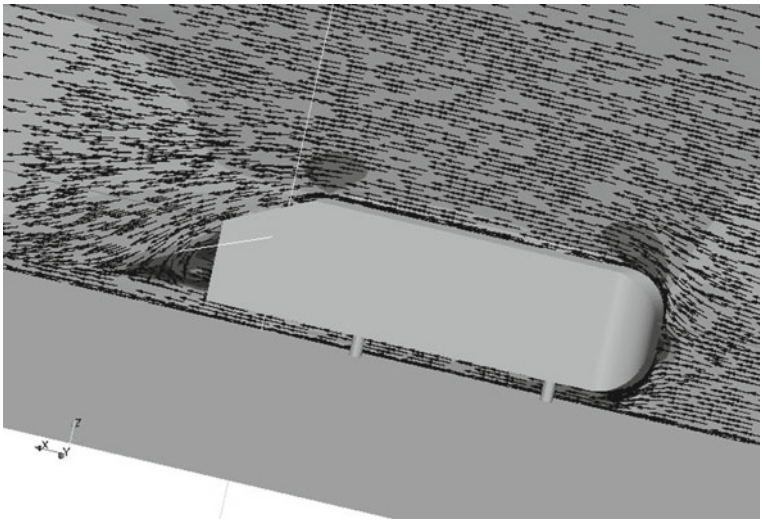


Fig. 19.7 Vector plots around body

The separation region is obvious in the vector plots shown in Fig. 19.7 where a wake is seen to form behind the body. The formation of this vortex region is responsible for pressure losses and is the main contribution to the drag on the body.

19.4 Conclusion

The simpleFoamTurbulent solver and boundary conditions developed in previous chapters were used to solve for the turbulent flow around the Ahmed bluff body. Despite the low order convection scheme and the relatively coarse mesh used, the important features of the flow were demonstrated.

References

1. Ahmed SR, Ramm G, Faltin G (1984) Some salient features of the time averaged ground vehicle wake. SAE Paper 840300
2. Lienhart H, Stoots C, Becker S (2000) Flow and turbulence structures in the wake of a simplified car model (Ahmed Model). In: DGLR Fach Symposium der AG STAB
3. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
4. OpenFOAM® User Guide. http://www.openfoam.org/docs/user/snappyHexMesh.php. London 2012, 2014

Chapter 20
Closing Remarks

The book covered the foundations of the Finite Volume Method, a method that is currently considered to be one of the most reliable and efficient methods for the simulation of flow and heat transfer problems. It is, thus, not surprising that it is the main numerical technique used in CFD whether in leading commercial codes or in open source libraries and frameworks.

While focused on scalar transport and single incompressible and compressible fluid flow problems, the material presented in this book is quite substantial, ranging from fundamental issues related to the numerics of the FVM to implementation details and CFD applications. All topics were introduced from first principles but were treated thoroughly leading in many cases to covering advanced concepts.

Much of the success of the FVM is attributable to a very active community of developers who have generously shared their techniques and discoveries in journal articles, conference proceedings, and books. Nonetheless it all started with a handful of researchers who were instrumental in setting the foundations for the method. The following is a very short and incomplete list of those early contributors whose publications were, at some time, the basic teaching documents for the method, at least for us the authors. In some way their combined work provided us with much of our early insight into the method, as such we wish to acknowledge their contributions:

B. Spalding, S. Patankar, G. Raithby, B.P. Leonard, F. Harlow, I. Raad, A. Gosman, J.P van Doormaal, S. Muzaferija, I. Demirdzic, M. Peric, G.E. Schneider, P. Galpin, S. Majumdar, C.W. Hirt, C. Hirsch, and V. Voller.

Finally as with many endeavors, there is space for much improvements. Thus it is our hope that you will share with us your ideas and suggestions on how to make the book with the associated codes a better and more useful educational and support tool.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_20

Erratum to: The Finite Volume Method in Computational Fluid Dynamics

F. Moukalled, L. Mangani and M. Darwish

Errattum to:
F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*, Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6

Initially the book was published without the extras material. Now it has been included through this erratum. The material is available at http://extras.springer.com/978-3-319-16873-9

The online version of the original book can be found under
DOI 10.1007/978-3-319-16874-6

F. Moukalled (✉)
Department of Mechanical Engineering, American University of Beirut, Beirut, Lebanon
e-mail: memouk@aub.edu.lb

L. Mangani
Engineering and Architecture, Lucerne University of Applied Science and Arts, Horw, Switzerland
e-mail: luca.mangani@hslu.ch

M. Darwish
Department of Mechanical Engineering, American University of Beirut, Beirut, Lebanon
e-mail: darwish@aub.edu.lb

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*, Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_21

Appendix
uFVM

A.1 Introduction

uFVM is a general three dimensional unstructured finite volume-based code developed in Matlab® for the solution of general single fluid flow and transport phenomena problems. The code is written for academic purpose and emphasizes programming simplicity and readability over performance. The code is capable of dealing with a wide range of flow problems and is easily extensible.

A.2 The Base Structure

The code is composed of task specific functions that mimic the numerical functions used in the finite volume method. The user can setup a case by writing a script to read the geometry and define the mathematical model with its associated initial and boundary conditions. An illustrative script file is presented in Listing A.1 showing a test case that involves the solution of a simple scalar equation.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6

```
% Convection-Diffusion Problem Solved on Static Grid

   1.clear all;
   2.clc;
   3.global Domain
   4.cfdSetupDomain;

%Reading the Geometry from OpenFOAM
   5.cfdReadOpenFoamMesh('Domain25CV');

% setup Fluid
   6.cfdSetupFluid('water','MW',18);
   7.cfdSetIsTransient

%Creating the Property Fields
   8.cfdSetupProperty('Density:water','constant','1000');
   9.cfdSetupProperty('SpecificHeat:water','constant','4.186');
   10.cfdSetupProperty('conduction:water','constant','4.186');
   11.cfdSetupProperty('Velx:water','constant','10');
   12.cfdSetupProperty('Vely:water','constant','10');
   13.cfdSetupProperty('Velz:water','constant','0');
%=======================
% Setting the equation:
%=======================
   14.cfdSetupEquation('T:water','ic','0','urf',1);
% Adding the terms constituting the equation with appropriate coeffi-
cients
   15.cfdAddTerm('T:water','Transient','coefficientName','Density:wa-
     ter',...
'coeffiecientName','SpecificHeat:water');
   16.cfdAddTerm('T:water','Diffusion','coefficientName','conduc-
     tion:water');
   17.cfdAddTerm('T:water','Convection','coefficientName','Density:wa-
     ter',...
'scheme','UPWIND');
% Specifying the Boundary Conditions for the equation
   18.cfdSetBC('T:water',1,'type','Specified  Value','value','10');  %
     Inlet BC
   19.cfdSetBC('T:water',2,'type','Outlet'); % Outlet BC
   20.cfdSetBC('T:water',3,'type','Specified  Value','value','0');  %
     Side Walls
   21.cfdSetBC('T:water',4,'type','empty'); % Front & Back

% Creating an Mdot Field
   22.cfdSetupMdotFields;

% Initializing the special array for each field
   23.cfdInitializeFields;

   24.time_i=0;
   25.time_f=7;
   26.dt=1;
```

Listing A.1 A script for solving a convection-diffusion problem using uFVM

```
% Starting the Time loop
   27.for time=time_i:dt:time_f
   28.k=1;
   29.time_p=time;
   30.time_c=time_p+dt;
   31.cfdSetTime(time_c)
   32.cfdSetDt(dt)
   33.cfdTransientUpdate;
   34.fprintf('%s %d \n', 'Time:', time_c);
   35.disp('----------------------');
   36.for iter=1:100
   37.cfdUpdateFields;
   38.fprintf('%s\n %d \n','Iteration: ',iter);
   39.cfdAssembleAndCorrectEquation('T:water');
   40.end

% Plotting the temperature field and the residuals of the equation
   41.iFigure=time_c;
   42.cfdPlotField('T:water',k);
   43.colorbar;
   44.cfdPlotResiduals;

% A step to save the Phi Field at each time step
   45.theMeshField=cfdGetMeshField('T:water');
   46.S=theMeshField.phi;
   47.Domain.FieldPhi(k).S=S;

   48.k=k+1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   49.end
```

Listing A.1 (continued)

A.3 Reading the Mesh

The uFVM code can read an OpenFOAM® mesh directly from an OpenFOAM® case directory. Specifically from the polyMesh subfolder that contains the mesh geometric and topological information. The standard structure of an OpenFOAM® case folder is presented in Fig. A.1.

The '0' directory is the initialization sub-folder containing the initial and boundary conditions of the fields defined in the problem model. The 'constant' directory contains the dynamicMeshDict, which is a dictionary for dynamic meshes, and a 'polyMesh' directory special for the description of the problem's geometry. The 'system' directory contains dictionaries that define the case setup. The controlDict is concerned with the general control parameters of the test case, the fvSchemes defines the discretization schemes, while the fvSolution contains information about the solution algorithms and relaxations to be used in the simulation.

uFVM reads and processes the polyMesh folder information in the **cfdReadOpenFoamMesh** function. The function starts by reading the points file and storing the *x, y, z* coordinates into a structure array called the nodes array, then the geometric information is read from the faces file in the form of a list of node indices for each face. This information is processed and stored in a structure array

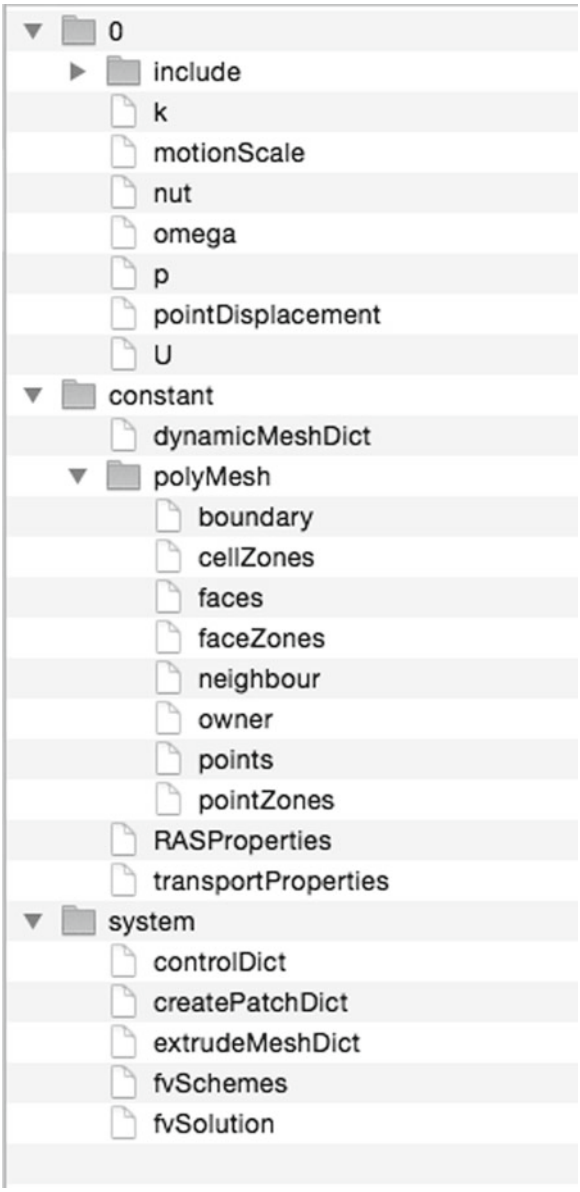


Fig. A.1 The standard structure of an OpenFOAM® case folder

called faces. Information about the patches and the associated patch faces is then read from the boundary file and stored in a structure array denoted by boundary. Finally, the owners and neighbors files are read and the elements structure array is composed. Then, the topological information for elements, faces, nodes and their connectivities are processed in function **cfdProcessOpenFoamMesh**. This is in addition to secondary geometric information such as the volume and centroids of

elements, centroids and area of faces, as well as interpolation factors, and other geometrical entities.

At this stage, the mesh contains: the nodes array storing for each node its coordinates, index, and the element/s and faces to which it belongs; the faces array storing for each face its index, the coordinates of its centroid, area, interpolation factor, area vector, the owner and the neighbor, patch index and the nodes that construct it; the elements array storing for each element the index, the indices of the

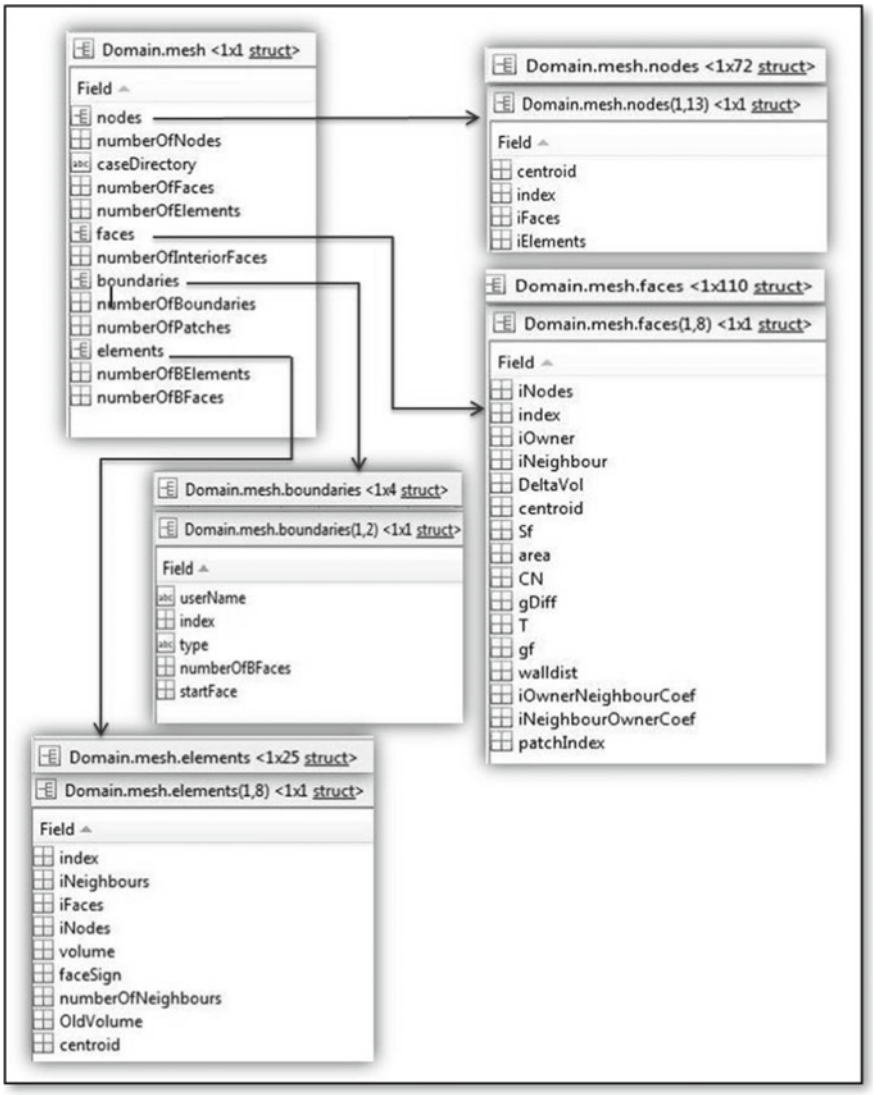


Fig. A.2 The internal data structure in uFVM

neighbors, the indices of the faces and the nodes constructing the element, volume, face sign and the coordinates of its centroid. Moreover, a boundary structure array is created that contains the boundary types included in the problem and read from the OpenFoam® boundary file. The array contains an index of the boundary patches and their types, their number of faces and the starting face of each patch. An illustration of the above described structure is displayed in Fig. A.2.

A.4 Setting-Up the Model

Before setting up the equations underlying the physical model and representing the major constituents of any case, the fluid/s involved in the problem should be described. This is done through the function **cfdSetupFluid**, which is responsible for defining the type of the fluid, its username, molecular weight, type (compressible or incompressible), in addition to some other related information. It should be noted at this stage that all the data that will be used by other functions are saved in a global structure array named Domain. The thermo-physical variables appearing in the governing equations should be defined as well. This is the role of the special function **cfdSetupProperty**. In this function, the user should insert, among others, the name of the property, its type, and the under relaxation factor needed when updating its value. All data is stored in an element mesh field.

Now the setup of the governing equations can proceed for any problem by using the function **cfdSetupEquation**. This is where the type of the equation (scalar or vector) is defined along with the initial conditions and the under-relaxation factor. The user can also choose the gradient type to be used for evaluating the derivatives appearing in the conservation equation. A field for each equation is stored on an element mesh size array in the global structure array Domain. For each defined equation, the user can add the various terms (*transient*, *convection*, *diffusion*, *pressure gradient*, *stress*, *source*, *electric potential*, *Darcy*, *buoyancy*, and *drag* among others) that constitute the equation. Each term is defined in the code using the function **cfdAddTerm**. This function relates each term along with the coefficients (density, viscosity, and diffusion coefficient among others) to its equation where the information is stored for later use during assembly. The associated boundary conditions of each equation are then added using the function **cfdAddBC**, where the type of the boundary condition (inlet, outlet, specified value, specified flux, slip, no slip among others) and its value, if needed, are specified. If any of the equations contains a convective term, then the user has to setup a '**mdot**' (mass flow rate: density multiplied by the dot product of the velocity vector and the area vector) field, which creates a mesh field covering the faces of the domain.

The internal structure of an equation (e.g., 'Velocity:water') is shown in Listing A.2.

```
velocityModel = cfdGetModel('Velocity:water')

            name: 'Velocity_fluid01'
        userName: 'Velocity:water'
           class: 'Equation'
            type: 'Vector'
              ic: '[1;0;0]'
             urf: 0.8000
     isTransient: 0
    gradientType: 'GAUSS0'
           terms: {1x3 cell}
       residuals: [1x3 double]
          source: ''
             bcs: {1x6 cell}
             tag: '_fluid01'
         rhoName: 'Density:water'
        resArray: [918x3 double]
```

Listing A.2 Internals of an equation (model) structure

The structure of a term is shown in Listing A.3.

```
>> velocityModel.terms{1}


               name: 'Convection'
       variableName: 'Velocity_fluid01'
    coefficientName: 'Density:water'
               type: 'Residual'
               sign: 1
             scheme: 'DEFAULT'

>> velocityModel.terms{2}


               name: 'Stress'
       variableName: 'Velocity_fluid01'
    coefficientName: ''
               type: 'Residual'
               sign: 1
             scheme: 'DEFAULT'
        coefficient: 'Viscosity:water'


>> velocityModel.terms{3}

               name: 'Pressure Gradient'
       variableName: 'Velocity_fluid01'
    coefficientName: ''
               type: 'Residual'
               sign: 1
             scheme: 'DEFAULT'
```

Listing A.3 Internals of the term structure

A.5 Setup the Computational Fields

This section describes how fields are initialized in the code, each on its prescribed locale (elements, faces, nodes). Adding the function **cfdInitializeFields** into the script file will automatically initialize the equation field, the property field, and the 'mdot' field. An equation is initialized over elements by computing and distributing the initial conditions onto each interior element along with the previously defined boundary conditions. After that, a property is initialized over the associated mesh field, whether it should be computed from a formula or has a constant value, by evaluating or placing the values over elements or faces. The '**mdot**' field is initialized as well by calling the density field (which is a property already defined and initialized) along with the velocity field to compute the value over each face of the mesh. Storing all the initialized fields in the appropriate arrays for later communication is done by each associated function (**cfdInitializeEquation, cfdInitiazeProperty, cfdInitializeMdotField**).

A.6 Equation Discretization (Assembly)

After all fields have been initialized and the environment has become suitable to start the solution process, the function **cfdAssembleAndCorrectEquation** is invoked in order to assemble, solve, and correct the equations governing the modeled problem.

A.6.1 Equation Assembly

For each equation, the internal function **cfdAssembleEquation** assembles the terms that have already been associated with the specified equation. A coefficient array containing the coefficients of the variable, of size equal to the number of control volumes, is setup using the cfdSetupCoefficients function, as illustrated in Fig. A.3.

A process is initiated in **cfdAssembleEquationTerms** to loop over each term and calculate its fluxes according to the selected or default scheme. The calculated face fluxes and element fluxes are then assembled according to their discretized form into the global coefficient matrix by **cfdAssembleIntoGlobalMatrix FaceFluxes** and **cfdAssembleIntoGlobalMatrixElementFluxes**, respectively. After assembling all the terms of a single equation and obtaining the complete array of coefficients, the residuals of the equation are computed using the function **cfdComputeResidual** according to the prescribed equation of the residuals. After that the under-relaxation factor, already specified for each equation, is applied on the coefficients.

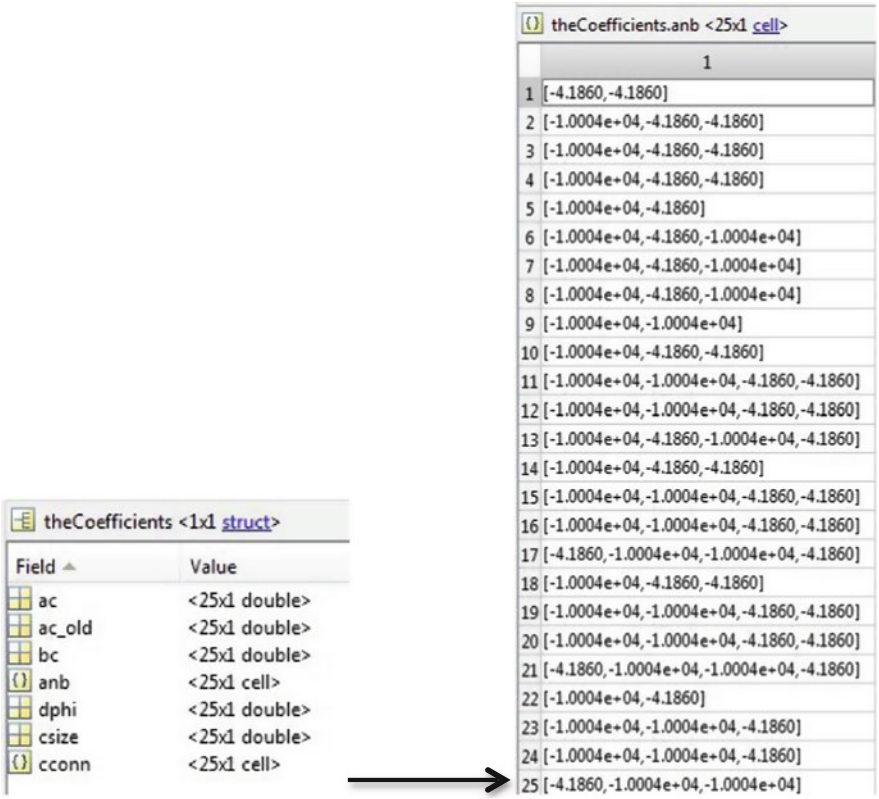


Fig. A.3 Illustration of the array of coefficients

A.6.2 Solving the Equations

Two algebraic solvers are implemented in uFVM, which are the successive over-relaxation method (SOR) and the ILU(0) method. The SOR method is simply the under-relaxed Gauss-Siedel method. With SOR the array containing the coefficients is imported to the solver, which in turn loops over the elements of the domain to solve and update the values of the unknown field ϕ. The system of equations is solved in residual form according to the following equation:

$$\phi'_C = \frac{b_C - a_C\phi_C - \sum\limits_{F\sim NB(C)} a_F\phi_F}{a_C} \tag{A.1}$$

The ILU(0) method follows the methodology described in Chap. 10, where also the system of equations is solved in residual form.

Whether solving a simple scalar equation or a set of scalar and vector equations (i.e., the Navier-Stokes equations), the solutions of these equations have to be

corrected using the **cfdCorrectEquation** function along with several internal functions specific for the correction of each type of equations: velocity, scalar or pressure (which has a specific treatment). After the correction field is obtained, the phi field at internal nodes is updated. Boundary conditions are corrected, each according to its specified type through the use of any of the following functions: **cfdCorrectWallZeroFluxBC, cfdCorrectWallSpecifiedFluxBC, cfdCorrect InletInletBC**, and **cfdCorrectOutletZeroFluxBC** among others. The pressure equation is corrected by executing **cfdCorrectPPField** and **cfdCorrect PressureEquation** because of the need to correct the pressure correction and pressure fields. After that, the velocity and the '**mdot**' fields are corrected at the interior element faces as well as at boundaries.

A.6.3 Computing the Residuals

The residual of each equation is calculated using the **cfdComputeResidual**. First a scaled ϕ for the equation to be solved is calculated as

$$\phi_{scale} = \max(\phi_{\max} - \phi_{\min}, abs(\phi_{\max})) \tag{A.2}$$

then the residuals are scaled using

$$R^{\phi}_{C,scaled} = \sum_{all\ elements} \frac{b_C - a_C\phi_C - \sum\limits_{F \sim NB(C)} a_F\phi_F}{a_C\phi_{scale}} \tag{A.3}$$

and finally the root-mean square residual is computed over the domain as

$$R^{\phi}_{C,rms} = \sqrt{\frac{\sum\limits_{C \sim all\ cells} \left(R^{\phi}_{C,scaled}\right)^2}{number\ of\ elements}}. \tag{A.4}$$

The obtained value is stored since it is needed as an indicator for the convergence of the solution and can be plotted as explained later.

A.7 Plotting Utilities

Several plotting functions are available in uFVM allowing results in addition to mesh and geometry to be visualized. A summary of these functions is given below.

cfdPlotMesh plots the domain under consideration along with the mesh that covers it.

cfdPlotElements plots any element the user choose or a set of specified elements using the index of each.

cfdPlotFaces plots the faces of the domain using their indices.

cfdPlotPatches plots the full boundary patch that the user choose by using the index of the patch already defined in earlier stages of a case problem.

cfdPlotContours plots contours of the variable over the domain.

cfdPlotField plots any field defined in the solver, as shown in Fig. A.4.

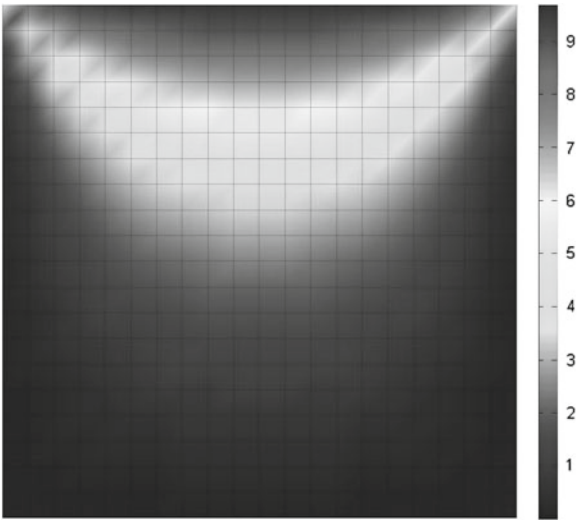


Fig. A.4 The ϕ field over the domain of interest

cfdPlotResiduals plots the residual value of each variable per iteration, as shown in Fig. A.5.

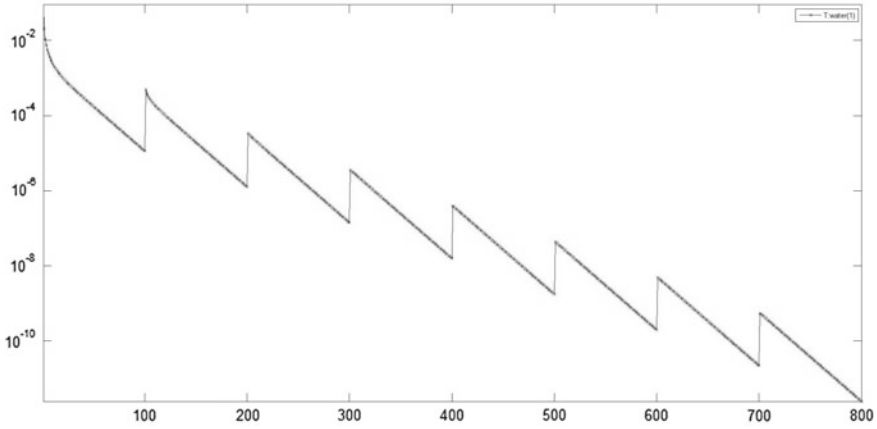


Fig. A.5 Variation of the residual of ϕ with iterations for a transient run

cfdPlotVelocity is responsible for plotting the mesh and the velocity vectors on the element centroids and on boundary faces, as shown in Fig. A.6.

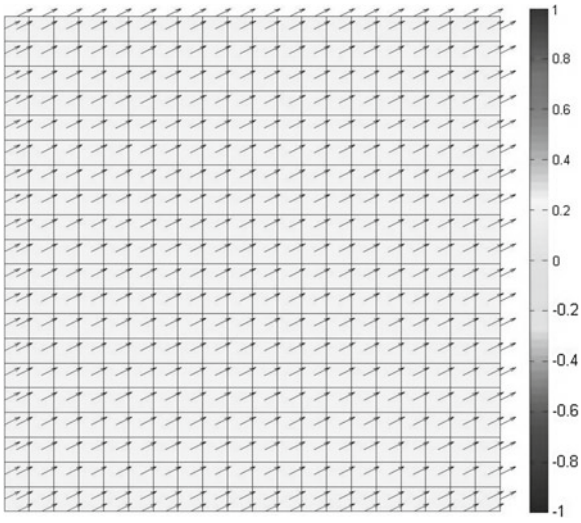


Fig. A.6 A plot of the velocity field over the domain

A.8 Interpolation Schemes

Several interpolation functions are included in uFVM serving different purposes. A summary of these interpolation functions is given below.

1. **cfdInterpolateFromElementsToNodes** function is used to compute the gradient according to the node-based method and in the **cfdPlotField** function. A loop over all nodes is performed and for each node an array is created to store the indices of the elements sharing the node. The value of the variable at the node is computed by applying Eq. (9.18).
2. **cfdInterpolateFromElementsToFaces** function is used in assembling the stress, diffusion, and 'mdot' terms as well as in the initialization of fields. Three interpolation schemes (Hyperbolic, Upwind, and Average) are implemented in this function from which the user can choose when computing face values from element values.
3. **cfdInterpolateFromNodesTofaces** function is used in computing the gradient according to the node-based method. For a single face, the value at the face centroid is computed from the values at the face nodes using Eq. (9.18).
4. **cfdInterpolateGradientsFromElementsToInteriorFaces** function is used to interpolate the gradients from elements to interior faces according to the selected interpolation scheme. Four options are available for this function (Average, Upwind, Downwind and Corrected Average). The Average scheme depends on the weighting geometric factor and includes the owner and neighbor of the

interior face. The Upwind and Downwind schemes use the value of the upwind and downwind element, respectively, depending on the direction of the 'mdot' vector at the specified interior face. The corrected Average scheme resembles the Average scheme but with the introduction of a correction to the interpolated gradient according to Eq. (9.35).

A.9 Test Cases

uFVM comes with a set of basic test cases that can be used to learn how to setup and run problems using the code. They are also useful as initial templates to setup new problems. The name of files for all test cases start with 'test', some of these test cases are listed below.

testAdvection.m
testDiffusion.m
testDiffusion01.m
testDiffusion02.m
testDiffusion03.m
testDiffusion04.m
testFlow01.m
testFlow02.m
testGradients.m
testShearRate.m
testSmithHutton.m
testStepProfile.m
testStepProfile2.m
testTemperature.m
testTransient.m
testTurbulence.m

Any test case can be run simply by calling it from within Matlab® as a script with no input. More information on uFVM can be obtained throughout the book in the various Computational Pointers sections.

A.10 Closing Remarks

uFVM is mainly used as a teaching tool. It is easy to read but it may take some time to get used to, especially the setting up of cases. However it provides a clear implementation of many of the numerics currently used in industrial type CFD codes. The uFVM code was found to provide good and practical insight to students. It is our hope that you will find it as useful in your classroom as we have in ours.