Cite as: Lucchese, L.: Implementation of non-reflecting boundary conditions in OpenFOAM. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2022

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of non-reflecting boundary conditions in OpenFOAM

Developed for OpenFOAM-v2112

Author:
Leandro Lucchese
Sapienza University of Rome
leandro.lucchese@uniroma1.it

Peer reviewed by:
Saeed Salehi
Pietro Paolo Ciottoli
Iason Tsiapkinis

January 13, 2023

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- how and when to use OpenFOAM's native non-reflecting boundary conditions.
- the difference between OpenFOAM's native `advective` and `waveTransmissive` boundary conditions, when and for what flow variables to use them.
- how to use the new `LODI` boundary conditions when running a case.

The theory of it:

- the theory on which characteristic based boundary conditions are based.
- the difference between LODI equations for inviscid flow and the general characteristics equations.
- the limitations of current OpenFOAM non-reflecting boundary conditions.

How it is implemented:

- an in depth description of how non reflecting boundary conditions are currently implemented in OpenFOAM in the form of `advective` and `waveTransmissive` boundary conditions.
- how these boundary conditions are used as a reference to build a new class of boundary conditions based on the characteristic analysis of the N-S equations.

How to modify it:

- how to modify these and other boundary conditions for various cases of interest

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- Comprehensive understanding of the Navier-Stokes equations and general fluid dynamics.
- General knowledge of computational fluid dynamics (CFD) and numerical methods for discretizing the equations.
- Basic understanding of C++ programming.
- How to compile a top-level OpenFOAM application.

Contents

Nomenclature

Acronyms

CFD	Computational fluid dynamics
FVM	Finite volume method
LODI	Local one dimensional inviscid
N-S	Navier-Stokes
PDE	Partial differential equation

English symbols

$\mathscr{L}_i$	Amplitude of the $i-th$ wave	
c	Speed of sound	m/s
C_p	Specific heat capacity	$\mathrm{J \cdot K/kg}$
E	Specific total energy	J/kg
m_i	$i-th$ direction momentum density	$\mathrm{kg/m^2s}$
Pr	Prandtl number	
q_i	Heat flux along $i-th$ direction	$\mathrm{W/m^2}$
U	Cartesian velocity vector	m/s
u_i	$i-th$ Component of the cartesian velocity vector	m/s
u_n	Component of the cartesian velocity vector normal to the patch surface	m/s
u_t	Component of the cartesian velocity vector tangent to the patch surface	m/s

Greek symbols

δ_{ij}	Kronercker's delta	
γ	Specific heat ratio	
λ_i	Characteristic velocity of wave i	
μ	Fluid dynamic viscosity	$\mathrm{kg \cdot s/m}$
ν	Fluid kinematic viscosity	$\mathrm{m^2/s}$
ϕ	Generic field variable	
ρ	Fluid density	$\mathrm{kg/m^3}$
τ	Stress tensor	Pa

Superscripts

n	previous time-step
n+1	current time-step

Subscripts

c	cell center
f	face center

Introduction

When performing a CFD analysis of any of flow and geometry there will have to be, at some point, boundaries. These boundaries can be physical, given by the geometry that is being studied, like walls, or numerical, e.g an outlet from which the flow exits the domain of interest. When performing simulations of compressible flows in closed geometries, dealing with these outlets can be particularly problematic when the goal is to have no reflection of waves at the boundary. This is especially true when dealing with the turbulent reacting flows in combustion chambers, in which a large amount of experimental evidence shows that there is a strong coupling between acoustic waves and other mechanisms of the flow. There are many ways to solve this issue, like considering a larger domain and also simulating the external ambient hence not having to deal with the outlet of the chamber, or considering a *sponge* acting on a certain non-physical region in order to damp the flow variables to a known reference solution, as discussed extensively by Mani in [1]. These approaches, however, usually increase the computational cost significantly while adding no benefits since the equations have to be solved in a part of the domain that is of no interest to the study.

For these reasons, one possible solution is introducing non-reflecting or partially-non reflecting boundary conditions, which are obtained by exploiting the hyperbolic nature of the Navier-Stokes equations in order to obtain a formulation in terms of waves entering and exiting the domain that can be used to write a set of conditions on the primitive variables that ensures a high level of accuracy.

Chapter 1

Characteristic based boundary conditions for Navier-Stokes equations

1.1 Characteristic analysis of the Navier-Stokes equations

What follows is a description of the characteristic analysis of the Navier-Stokes equations as described by Poinsot and Lele [2], Lodato et al. [3] and Valorani and Favini [4].
For a compressible viscous flow the fluid dynamics equations in Cartesian coordinates are given by

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(m_i) = 0, \tag{1.1}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial}{\partial x_i}[(\rho E + p)u_i] = \frac{\partial}{\partial x_i}(u_j \tau_{ij}) - \frac{\partial q_i}{\partial x_i}, \tag{1.2}$$

$$\frac{\partial m_i}{\partial t} + \frac{\partial}{\partial x_j}(m_i u_j) + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j}, \tag{1.3}$$

with

$$\begin{aligned} \rho E &= \frac{1}{2}\rho u_k u_k + \frac{p}{\gamma - 1}, \\ m_i &= \rho u_i, \\ \tau_{ij} &= \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_k}{\partial x_k} \right), \end{aligned}$$

where p is the thermodynamic pressure, m_i is the momentum per unit volume in the x_i direction, ρE is the total energy per unit volume (kinetic + thermal). The heat flux q_i along x_i is given by

$$q_i = -\lambda \frac{\partial T}{\partial x_i},$$

where λ is the thermal conductivity and is obtained from the viscosity coefficient μ through

$$\lambda = \mu C_p / Pr,$$

where Pr is the Prandtl number.
Equations (1.1)-(1.3) can be conveniently written in vector form as

$$\frac{\partial \widetilde{\boldsymbol{U}}}{\partial t} + \frac{\partial \widetilde{\boldsymbol{F}}^i}{\partial x_i} + \frac{\partial \widetilde{\boldsymbol{D}}^i}{\partial x_i} = \mathbf{0}, \tag{1.4}$$

where $\widetilde{\boldsymbol{U}} = |\rho \quad \rho u_1 \quad \rho u_2 \quad \rho u_3 \quad \rho E|^{\mathrm{T}}$ is the vector of conservative variables, $\widetilde{\boldsymbol{F}}^k$ is the flux vector of the conservative variables along the x_k direction and the vectors $\widetilde{\boldsymbol{D}}^k$ contain the viscous and diffusive terms, these two can be explicitly written as

$$\widetilde{\boldsymbol{F}}^k = \begin{pmatrix} \rho u_k \\ m_1 u_k + \delta_{1k} p \\ m_2 u_k + \delta_{2k} p \\ m_3 u_k + \delta_{3k} p \\ (\rho E + p) u_k \end{pmatrix}, \quad \widetilde{\boldsymbol{D}}^k = \begin{pmatrix} 0 \\ -2\mu A_{1k} \\ -2\mu A_{2k} \\ -2\mu A_{3k} \\ -2\mu u_j A_{kj} + q_k \end{pmatrix},$$

where δ_{ij} is Kronecker's delta and $A_{ij} = \tau_{ij}/2\mu$.

If one defines the vector of primitive variables as $\boldsymbol{U} = \left| \begin{matrix} \rho & u_1 & u_2 & u_3 & p \end{matrix} \right|^{\mathrm{T}}$, by following the procedure described by Thompson [5], equation (1.1) can be rewritten in terms of primitive variables as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{F}^i \frac{\partial \boldsymbol{U}}{\partial x_i} + \boldsymbol{D} = \boldsymbol{0}, \tag{1.5}$$

where $\boldsymbol{D} = \boldsymbol{P}^{-1} \partial \widetilde{\boldsymbol{D}}^i / \partial x_i$ is a vector that includes all the viscous and diffusive terms and $\boldsymbol{F}^k$ represents the non-conservative Jacobian matrix related to the k th direction which in this case can be expressed as

$$\boldsymbol{F}^k = \begin{pmatrix} u_k & \delta_{1k}\rho & \delta_{2k}\rho & \delta_{3k}\rho & 0 \\ 0 & u_k & 0 & 0 & \delta_{1k}/\rho \\ 0 & 0 & u_k & 0 & \delta_{2k}/\rho \\ 0 & 0 & 0 & u_k & \delta_{3k}/\rho \\ 0 & \delta_{1k}\gamma p & \delta_{2k}\gamma p & \delta_{3k}\gamma p & u_k \end{pmatrix}. \tag{1.6}$$

The matrix $\boldsymbol{P} = \partial \widetilde{\boldsymbol{U}} / \partial \boldsymbol{U}$ is the Jacobian matrix that allows to change coordinates between primitive and conservative variables and is given by

$$\boldsymbol{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ u_1 & \rho & 0 & 0 & 0 \\ u_2 & 0 & \rho & 0 & 0 \\ u_3 & 0 & 0 & \rho & 0 \\ \frac{1}{2} u_k u_k & \rho u_1 & \rho u_2 & \rho u_3 & \frac{1}{\gamma - 1} \end{pmatrix}. \tag{1.7}$$

Each non-conservative Jacobian matrix $\boldsymbol{F}^k$ related to every direction k can be diagonalized through

$$\boldsymbol{S}_k^{-1} \boldsymbol{F}^k \boldsymbol{S}_k = \boldsymbol{\Lambda}^k, \tag{1.8}$$

and the eigenvalues are given by

$$\begin{aligned} \lambda_1^k &= u_k - c, \\ \lambda_{2,3,4}^k &= u_k, \\ \lambda_5^k &= u_k + c, \end{aligned}$$

where $c = \sqrt{\frac{\gamma p}{\rho}}$ is the speed of sound and the two matrixes $\boldsymbol{S}_k$ and $\boldsymbol{S}_k^{-1}$ can be explicitly written as

$$\boldsymbol{S}_k = \begin{pmatrix} \frac{1}{2c^2} & \frac{\delta_{1k}}{c^2} & \frac{\delta_{2k}}{c^2} & \frac{\delta_{3k}}{c^2} & \frac{1}{2c^2} \\ -\frac{\delta_{1k}}{2\rho c} & 1 - \delta_{1k} & 0 & 0 & \frac{\delta_{1k}}{2\rho c} \\ -\frac{\delta_{2k}}{2\rho c} & 0 & 1 - \delta_{2k} & 0 & \frac{\delta_{2k}}{2\rho c} \\ -\frac{\delta_{3k}}{2\rho c} & 0 & 0 & 1 - \delta_{3k} & \frac{\delta_{3k}}{2\rho c} \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}, \tag{1.9}$$

$$\boldsymbol{S}_k^{-1} = \begin{pmatrix} 0 & -\delta_{1k}\rho c & -\delta_{2k}\rho c & -\delta_{3k}\rho c & 1 \\ \delta_{1k}c^2 & 1-\delta_{1k} & 0 & 0 & -\delta_{1k} \\ \delta_{2k}c^2 & 0 & 1-\delta_{2k} & 0 & -\delta_{2k} \\ \delta_{3k}c^2 & 0 & 0 & 1-\delta_{3k} & -\delta_{3k} \\ 0 & \delta_{1k}\rho c & \delta_{2k}\rho c & \delta_{3k}\rho c & 1 \end{pmatrix}. \tag{1.10}$$

With this theoretical foundation laid down, depending on the type of boundary that is being considered (face, corner, etc.) a wide variety of different boundary conditions can be considered by taking into account different numbers of characteristic directions.

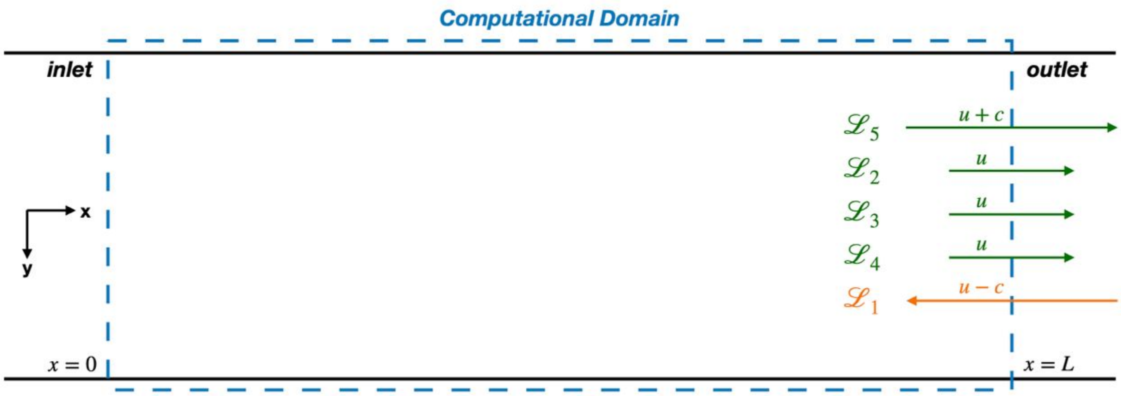


Figure 1.1: 2D domain with waves leaving and entering the domain

When considering a boundary with normal vector parallel to x_1 as in Fig. 1.1, the characteristic waves considered will be those traveling along the x_1 direction and therefore only $\boldsymbol{F}^1$ needs to be diagonalized, Equation (1.5) can be hence written as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{S}_1\boldsymbol{\Lambda}^1\boldsymbol{S}_1^{-1}\frac{\partial \boldsymbol{U}}{\partial x_1} + \boldsymbol{F}^2\frac{\partial \boldsymbol{U}}{\partial x_2} + \boldsymbol{F}^3\frac{\partial \boldsymbol{U}}{\partial x_3} + \boldsymbol{D} = \boldsymbol{0}, \tag{1.11}$$

and a vector $\mathscr{L}$ whose components $\mathscr{L}_i$ represent the amplitude time variations of the characteristic waves can be defined as

$$\mathscr{L} = \boldsymbol{\Lambda}^1\boldsymbol{S}_1^{-1}\frac{\partial \boldsymbol{U}}{\partial x_1} = \begin{pmatrix} \lambda_1\left(\frac{\partial p}{\partial x_1} - \rho c\frac{\partial u_1}{\partial x_1}\right) \\ \lambda_2\left(c^2\frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1}\right) \\ \lambda_3\frac{\partial u_2}{\partial x_1} \\ \lambda_4\frac{\partial u_3}{\partial x_1} \\ \lambda_5\left(\frac{\partial p}{\partial x_1} + \rho c\frac{\partial u_1}{\partial x_1}\right) \end{pmatrix}. \tag{1.12}$$

Equation (1.11) can finally be written as a function of the wave amplitude variations obtaining

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{d} + \boldsymbol{F}^2\frac{\partial \boldsymbol{U}}{\partial x_2} + \boldsymbol{F}^3\frac{\partial \boldsymbol{U}}{\partial x_3} + \boldsymbol{D} = \boldsymbol{0}, \tag{1.13}$$

where

$$\mathbf{d} = \boldsymbol{S}_1\mathscr{L} = \begin{bmatrix} \frac{\partial m_1}{\partial x_1} \\ \frac{\partial(c^2 m_1)}{\partial x_1} + (1-\gamma)\mu\frac{\partial p}{\partial x_1} \\ u_1\frac{\partial u_1}{\partial x_1} + \frac{1}{\rho}\frac{\partial p}{\partial x_1} \\ u_1\frac{\partial u_2}{\partial x_1} \\ u_1 \end{bmatrix} = \begin{bmatrix} \frac{1}{c^2}\left[\mathscr{L}_2 + \frac{1}{2}\left(\mathscr{L}_5 + \mathscr{L}_1\right)\right) \\ \frac{1}{2}\left(\mathscr{L}_5 + \mathscr{L}_1\right) \\ \frac{1}{2\rho c}\left(\mathscr{L}_5 - \mathscr{L}_1\right) \\ \mathscr{L}_3 \\ \mathscr{L}_4 \end{bmatrix}, \tag{1.14}$$

By explicitly writing all the terms, the full system becomes

$$\frac{\partial \rho}{\partial t} + d_1 + \frac{\partial}{\partial x_2}(m_2) + \frac{\partial}{\partial x_3}(m_3) = 0, \tag{1.15}$$

$$\begin{aligned}\frac{\partial \rho E}{\partial t} + \frac{1}{2}(u_k u_k)\, d_1 + \frac{d_2}{\gamma - 1} + m_1 d_3 + m_2 d_4 + m_3 d_5 \\ + \frac{\partial}{\partial x_2}[(\rho E + p)u_2] + \frac{\partial}{\partial x_3}[(\rho E + p)u_3] = \frac{\partial}{\partial x_i}(u_j \tau_{ij}) - \frac{\partial q_i}{\partial x_i},\end{aligned} \tag{1.16}$$

$$\frac{\partial m_1}{\partial t} + u_1 d_1 + \rho d_3 + \frac{\partial}{\partial x_2}(m_1 u_2) + \frac{\partial}{\partial x_3}(m_1 u_3) = \frac{\partial \tau_{1j}}{\partial x_j}, \tag{1.17}$$

$$\frac{\partial m_2}{\partial t} + u_2 d_1 + \rho d_4 + \frac{\partial}{\partial x_2}(m_2 u_2) + \frac{\partial}{\partial x_3}(m_2 u_3) + \frac{\partial p}{\partial x_2} = \frac{\partial \tau_{2j}}{\partial x_j}, \tag{1.18}$$

$$\frac{\partial m_3}{\partial t} + u_3 d_1 + \rho d_5 + \frac{\partial}{\partial x_2}(m_3 u_2) + \frac{\partial}{\partial x_3}(m_3 u_3) + \frac{\partial p}{\partial x_3} = \frac{\partial \tau_{3j}}{\partial x_j}, \tag{1.19}$$

Here the explicit contribution of the waves becomes clear. The $\mathscr{L}_i$'s are the amplitudes of the waves entering and exiting the domain and for each wave is associated a characteristic velocity λ_i given by

$$\lambda_1 = u_1 - c, \tag{1.20}$$

$$\lambda_2 = \lambda_3 = \lambda_4 = u_1, \tag{1.21}$$

$$\lambda_5 = u_1 + c, \tag{1.22}$$

λ_1 and λ_5 are the velocities of the waves moving in the x_1 direction (positive and negative) and λ_3 and λ_4 are the velocities at which the x_2 and x_3 components of the velocity are advected in the x_1 direction (this will be very important later). The expressions of the $\mathscr{L}_i$'s are given by

$$\mathscr{L}_1 = \lambda_1 \left(\frac{\partial p}{\partial x_1} - \rho c \frac{\partial u_1}{\partial x_1} \right), \tag{1.23}$$

$$\mathscr{L}_2 = \lambda_2 \left(c^2 \frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1} \right), \tag{1.24}$$

$$\mathscr{L}_3 = \lambda_3 \frac{\partial u_2}{\partial x_1}, \tag{1.25}$$

$$\mathscr{L}_4 = \lambda_4 \frac{\partial u_3}{\partial x_1}, \tag{1.26}$$

$$\mathscr{L}_5 = \lambda_5 \left(\frac{\partial p}{\partial x_1} + \rho c \frac{\partial u_1}{\partial x_1} \right), \tag{1.27}$$

Equations (1.15) - (1.19) are the conservation equations written in terms of characteristic variables and wave amplitude variations (in the standard reference frame) and are extremely useful since they make it extremely easy to impose different types of boundary conditions in terms of the values that are assigned to the $\mathscr{L}_i$'s.

1.2 Local One-Dimensional Inviscid Relations (LODI)

When a 1D inviscid flow (Euler flow) is considered, Equation (1.13) becomes much easier and the system arising from this description is called the LODI system which in terms of primitive variables becomes

$$\frac{\partial \rho}{\partial t} + \frac{1}{c^2}\left[\mathscr{L}_2 + \frac{1}{2}\left(\mathscr{L}_5 + \mathscr{L}_1\right)\right] = 0, \tag{1.28}$$

$$\frac{\partial p}{\partial t} + \frac{1}{2}\left(\mathscr{L}_5 + \mathscr{L}_1\right) = 0, \tag{1.29}$$

$$\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}\left(\mathscr{L}_5 - \mathscr{L}_1\right) = 0, \tag{1.30}$$

$$\frac{\partial u_2}{\partial t} + \mathscr{L}_3 = 0, \tag{1.31}$$

$$\frac{\partial u_3}{\partial t} + \mathscr{L}_4 = 0, \tag{1.32}$$

These equations can be also written in another extremely useful form in terms of gradients normal to the boundary, as

$$\frac{\partial \rho}{\partial x_1} = \frac{1}{c^2}\left[\frac{\mathscr{L}_2}{u_1} + \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_1+c} + \frac{\mathscr{L}_1}{u_1-c}\right)\right], \tag{1.33}$$

$$\frac{\partial p}{\partial x_1} = \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_1+c} + \frac{\mathscr{L}_1}{u_1-c}\right), \tag{1.34}$$

$$\frac{\partial u_1}{\partial x_1} = \frac{1}{2\rho c}\left(\frac{\mathscr{L}_5}{u_1+c} - \frac{\mathscr{L}_1}{u_1-c}\right), \tag{1.35}$$

$$\frac{\partial T}{\partial x_1} = \frac{T}{\rho c^2}\left[-\frac{\mathscr{L}_2}{u_1} + \frac{1}{2}(\gamma-1)\left(\frac{\mathscr{L}_5}{u_1+c} + \frac{\mathscr{L}_1}{u_1-c}\right)\right], \tag{1.36}$$

By imposing different conditions to the amplitudes of characteristic waves $\mathscr{L}_i$'s a series of physically meaningful boundary conditions can be imposed, a fixed pressure boundary condition for example can be obtained by setting $\mathscr{L}_5 = -\mathscr{L}_1$ to fix the amplitude variation of the wave entering the domain, or **for a perfectly non-reflecting boundary condition** the incoming wave amplitude has to be set to 0 $\mathscr{L}_1 = 0$.

When solving a compressible, viscous flow using a finite difference method or a node-centered finite volume method for discretizing the N-S equations, the characteristic-based conservation Equations (1.15) - (1.19) have to be used by imposing the desired conditions on the wave amplitudes since the equations will actually be solved at the node that is being considered. When using a cell centered finite volume method though, the equations are not solved at the boundary and instead the contribution of every *boundary face* enters the linear system in the form of either a coefficient that sums the diagonal part of the coefficient matrix or the source term (or both). For this reason, it could be said that utilizing inviscid equations like the LODI system to determine the values of the primitive variables at the boundary implies a smaller error when utilizing a cell centered FVM since the contribution of the boundary face is only one of many faces.

1.2.1 Non-reflecting and partially non-reflecting boundary conditions

Supposing a subsonic flow in a simple domain like the one shown in Fig. 1.1 4 waves will be exiting the domain $\mathscr{L}_2$, $\mathscr{L}_3$, $\mathscr{L}_4$ and $\mathscr{L}_5$, and only one will enter the domain $\mathscr{L}_1$. If a perfectly non-reflecting boundary condition is to be implemented, the upcoming wave amplitude has to be set to zero $\mathscr{L}_1 = 0$, this condition should in theory be applied to the full N-S Equations (1.15) - (1.19) to find an equation for the primitive variables at the boundary, but if one considers a simplified boundary non-viscous boundary and applies the condition to the LODI Equations (1.28) - (1.32), the pressure and velocity Equations (1.29) and (1.31) become simply

$$\frac{\partial p}{\partial t} + \frac{1}{2}\left(\mathscr{L}_5\right) = 0, \tag{1.37}$$

$$\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}\left(\mathscr{L}_5\right) = 0, \tag{1.38}$$

which can be combined with Equations (1.34) and (1.35) and, using $\lambda_5 = u_1 + c$, become

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x} = 0, \tag{1.39}$$

$$\frac{\partial u_1}{\partial t} + (u_1 + c)\frac{\partial u_1}{\partial x} = 0, \tag{1.40}$$

Showing that when considering a 1-D inviscid flow and perfectly non-reflecting boundary conditions, the equations for pressure and velocity at the boundary are very simple and imply that these quantities are simply transported with a speed given by a velocity equal to λ_5. These equations are actually the ones implemented in OpenFOAM in the `waveTransmissive` boundary conditions, while the `advective` boundary conditions are identical but transporting the variables with u, as will be further discussed and shown in Sections 2.2.1 and 2.2.2.
Using perfectly non-reflecting conditions for Navier-Stokes equations is dangerous and often non recommendable since it may lead to an ill-posed problem as also explained by Rudy [6] and Poinsot [2]. One can easily understand why when imagining a domain where the mass flow rate is assigned at inlet and a perfectly non-reflecting boundary condition is employed at the outlet, the result will be that there is no way for the flow to determine what the pressure will be, and as result it will drift randomly form the initial value that one assigns.
In order to solve this problem and add some physical information on the mean static pressure, *partially non-reflective* boundary conditions can be applied as shown by Rudy and Strikwerda [7]. This is equivalent to imagining an outlet at a certain distance from the domain with pressure p_∞ that sends waves into the domain whose intensity depends on how different the pressure in the domain is with respect to p_∞, in order to do so the upcoming wave $\mathscr{L}_1$ is set as

$$\mathscr{L}_1 = K(p - p_\infty), \tag{1.41}$$

When this is applied, by following the same procedure as before one can find expressions for the primitive variables, for example the pressure reads

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x} + K(p - p_\infty) = 0, \tag{1.42}$$

If one were to discretize Equation (1.40) written for a general flow variable ϕ with an Euler time scheme, the results would be

$$\frac{\phi_f^{n+1} - \phi_f^n}{\delta t} + U_n\frac{\phi_f^{n+1} - \phi_c^{n+1}}{\delta x} = 0, \tag{1.43}$$

That, when manipulated becomes

$$\phi_f^{n+1} = \phi_f^n\frac{1}{1+\alpha} + \frac{\alpha}{1+\alpha}\phi_c^{n+1}, \tag{1.44}$$

With $\alpha = \delta t U_n / \mathbf{d}$.
Applying the same procedure to Equation (1.42) and introducing the α parameter as done before, after some manipulations the result becomes

$$\phi_f^{n+1} = (\phi_f^n + k\phi^\infty)\frac{1}{1+\alpha+k} + \frac{\alpha}{1+\alpha+k}\phi_c^{n+1}, \tag{1.45}$$

These two formulas are extremely important since it will be shown that they correspond exactly to the way OpenFOAM defines its boundary conditions.

Chapter 2

Non-reflecting boundary conditions in OpenFOAM

In this chapter, a general description of how boundary conditions are implemented in OpenFOAM is presented by showing the points in the code where the functions of these classes are called, and by discussing the implementation of the `mixed` boundary conditions. Then, a detailed description of non-reflecting boundary conditions in OpenFOAM is presented, with a major focus on the two classes `advective` and `waveTransmissive`. The important parts of the codes of these calsses are discussed and compared to the theory, and the correct way of utilizing these boundary conditions is investigated through two simple test-cases.

2.1 General boundary conditions in OpenFOAM

The two main functions used for discretizing the PDE's in OpenFOAM are the `fvm::div` and `fvm::laplacian` functions that can be seen in every top level solver. What this functions do is, for every primitive variable that is being solved, creating the linear system corresponding to the discretized PDE in terms of coefficient matrix and source term. In this context, as explained in detail in the *CFD with open source software* course slides, the boundary conditions contribute either to the diagonal coefficients of the coefficient matrix, to the source term, or both. This can be clearly seen in the `fvmDiv` function of the `gaussConvectionScheme`:

fvmDiv function of the `gaussConvectionScheme` class

```
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.ref();

    fvm.lower() = -weights.primitiveField()*faceFlux.primitiveField();
```

```
138     fvm.upper() = fvm.lower() + faceFlux.primitiveField();
139     fvm.negSumDiag();
140
141     forAll(vf.boundaryField(), patchi)
142     {
143         const fvPatchField<Type>& psf = vf.boundaryField()[patchi];
144         const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchi];
145         const fvsPatchScalarField& pw = weights.boundaryField()[patchi];
146
147         fvm.internalCoeffs()[patchi] = patchFlux*psf.valueInternalCoeffs(pw);
148         fvm.boundaryCoeffs()[patchi] = -patchFlux*psf.valueBoundaryCoeffs(pw);
149     }
150
151     if (tinterpScheme_().corrected())
152     {
153         fvm += fvc::surfaceIntegrate(faceFlux*tinterpScheme_().correction(vf));
154     }
155
156     return tfvm;
157 }
```

In line 143, the `psf` object of the `fvPatchField<Type>` class is introduced and two functions of that class are used to fill the diagonal coefficients and the source terms of the coefficient matrix: `valueInternalCoeffs()` and `valueBoundaryCoeffs()` (lines 147-148). These two functions are two of the main functions of any boundary condition in OpenFOAM. The same can be said for the laplacian term, where the contribution of the boundary conditions to the diagonal part of the coefficient matrix and the source term are provided by the functions `gradientInternalCoeffs()` and `gradientBoundaryCoeffs()`.

2.2 Mixed boundary conditions in OpenFOAM

Every boundary condition in OpenFOAM is either a `fixedValue`, a `fixedGradient`, or a mixture of the two, namely `mixed` boundary condition (otherwise referred to as Robin condition). Boundary conditions in OpenFOAM can be found in the `/src/finiteVolume/fields/fvPatchFields` folder, and the boundary conditions we are interested in are the `advective` and `waveTranmissive` that can be found in the `fvPatchFields/derived` folder and are sub-classes of the `mixed` that is instead found in the `fvPatchFields/basic` folder.
The way mixed boundary conditions work in OpenFOAM is by defining the value of the field at the boundary face as

$$\phi_f = w\phi_{\text{ref}} + (1-w)(\phi_c + \mathbf{d}\nabla(\phi_{\text{ref}})), \tag{2.1}$$

where:

- ϕ_f is the boundary face value,
- ϕ_c is the boundary cell value,
- ϕ_{ref} is a reference value,
- $\mathbf{d}$ is the face-to-cell distance,
- w is the "value fraction".

When applying this boundary condition directly these quantities have to be specified by the user, but the main scope of this boundary condition is to function as a base-class for other classes which will define their own values for these quantities. The declaration file for the mixed boundary conditions is `mixedFvPatchField.H`:

member data of the `mixedFvPatchField` class

```
template<class Type>
class mixedFvPatchField
:
    public fvPatchField<Type>
{
    // Private data

        //- Value field
        Field<Type> refValue_;

        //- Normal gradient field
        Field<Type> refGrad_;

        //- Fraction (0-1) of value used for boundary condition
        scalarField valueFraction_;

        //- Source field
        Field<Type> source_;
```

Here can be seen that the mixed boundary condition is a templated class, meaning that the definition is described only once through this file but at compilation it will be defined for many types of data (scalars, vectors, etc), and the member data of the class are those just described. The main functions of the class are:

- `evaluate`: Evaluates the patch Field.
- `snGrad`: Returns the patch normal gradient.
- `valueInternalCoeffs`: Returns the contribution to the coefficient matrix of the linear system of the divergence term at boundary patch.
- `valueBoundaryCoeffs`: Returns the contribution to the source term of the linear system of the divergence term at boundary patch.
- `gradientInternalCoeffs`: Returns the contribution to the coefficient matrix of the linear system of the laplacian term at boundary patch.
- `gradientBoundaryCoeffs`: Returns the contribution to the source term of the linear system of the laplacian term at boundary patch.

The definition of these functions is in the `mixedFvPatchField.C`:

main functions of the `mixedFvPatchField` class

```
template<class Type>
void Foam::mixedFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{

    if (!this->updated())
    {
        this->updateCoeffs();
    }

    Field<Type>::operator=
    (
        valueFraction_*refValue_
      + (1.0 - valueFraction_)
       *(
            this->patchInternalField()
          + refGrad_/this->patch().deltaCoeffs()
        )
    );

    fvPatchField<Type>::evaluate();
```

```
177 }
178
179
180 template<class Type>
181 Foam::tmp<Foam::Field<Type>>
182 Foam::mixedFvPatchField<Type>::snGrad() const
183 {
184     return
185         valueFraction_
186        *(refValue_ - this->patchInternalField())
187        *this->patch().deltaCoeffs()
188       + (1.0 - valueFraction_)*refGrad_;
189 }
190
191
192 template<class Type>
193 Foam::tmp<Foam::Field<Type>>
194 Foam::mixedFvPatchField<Type>::valueInternalCoeffs
195 (
196     const tmp<scalarField>&
197 ) const
198 {
199     return Type(pTraits<Type>::one)*(1.0 - valueFraction_);
200 }
201
202
203 template<class Type>
204 Foam::tmp<Foam::Field<Type>>
205 Foam::mixedFvPatchField<Type>::valueBoundaryCoeffs
206 (
207     const tmp<scalarField>&
208 ) const
209 {
210     return
211         valueFraction_*refValue_
212       + (1.0 - valueFraction_)*refGrad_/this->patch().deltaCoeffs();
213 }
214
215
216 template<class Type>
217 Foam::tmp<Foam::Field<Type>>
218 Foam::mixedFvPatchField<Type>::gradientInternalCoeffs() const
219 {
220     return -Type(pTraits<Type>::one)*valueFraction_*this->patch().deltaCoeffs();
221 }
222
223
224 template<class Type>
225 Foam::tmp<Foam::Field<Type>>
226 Foam::mixedFvPatchField<Type>::gradientBoundaryCoeffs() const
227 {
228     return
229         valueFraction_*this->patch().deltaCoeffs()*refValue_
230       + (1.0 - valueFraction_)*refGrad_;
231 }
```

As shown in lines 192 – 213, in OpenFOAM mixed boundary conditions are written in a standardized form that depends on these member data. If for simplicity we call the `valueFraction_` member data f the expression of the boundary condition can be written as

`valueInternalCoeffs = 1-f`,

`valueBoundaryCoeffs = f*refValue + (1-f)*refGrad*d`,

Remembering that the `valueInternalCoeffs()` function corresponds to the diagonal term and the `valueBoundaryCoeffs()` to the source term, if the two expressions are merged in order to obtain a single formula for the value that this boundary condition imposes to the field at the boundary patch, we obtain

$$\phi_f^{n+1} = f * \text{refValue} + (1-f)(\phi_c^{n+1} + \text{refGrad} * \mathbf{d}), \tag{2.2}$$

ϕ_c^{n+1} is the value of the field that is being solved at the current time step at the cell center (the implicit term), ϕ_f^{n+1} is the value of the field at the boundary face at the current time step (our unknown) and ϕ_f^n is the value of the field at the boundary patch at the previous time step (for the first time step it is provided by the user). This same formula is also written explicitly in the `evaluate()` function (lines 166-176) which evaluates the value of the patch field at the boundary face.
In general, when a mixed boundary condition has to be defined in OpenFOAM, the expression will have to be manipulated in order to get to a formula similar to that in Equation (2.2), where `valueFraction_`, `refValue` and `refGrad` have to be defined for each case.

2.2.1 Advective boundary conditions in OpenFOAM

The advective boundary condition in OpenFOAM is located in `fvPatchFields/derived` and the header file `advectiveFvPatchField.H` reads:

main functions of the `mixedFvPatchField` class

```
template<class Type>
class advectiveFvPatchField
:
    public mixedFvPatchField<Type>
{
protected:

    // Private data

        //- Name of the flux transporting the field
        word phiName_;

        //- Name of the density field used to normalise the mass flux
        //- if necessary
        word rhoName_;

        //- Field value of the far-field
        Type fieldInf_;

        //- Relaxation length-scale
        scalar lInf_;
```

Hence, the advectiveFvPatchField *is a* mixedFvPatchField (it inherits from the mixed class) and beyond the member data of the mixed class it has four added ones.
The two main functions of the `advectiveFvPatchField` class are:

- `advectionSpeed()`: calculates and returns the value of the advection speed at the boundary.
- `updateCoeffs()`: updates the coefficients associated to the patch field, i.e the `refValue`, `valueFraction` and `refGrad` used in the `mixedFvPatchField` boundary condition.

The definition of the `advectionSpeed()` function is here shown:

advectionSpeed function of the `mixedFvPatchField` class

```
template<class Type>
Foam::tmp<Foam::scalarField>
Foam::advectiveFvPatchField<Type>::advectionSpeed() const
{
    const surfaceScalarField& phi =
        this->db().objectRegistry::template lookupObject<surfaceScalarField>
```

```
        (phiName_);

    fvsPatchField<scalar> phip =
        this->patch().template lookupPatchField<surfaceScalarField, scalar>
        (
            phiName_
        );

    if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchScalarField& rhop =
            this->patch().template lookupPatchField<volScalarField, scalar>
            (
                rhoName_
            );

        return phip/(rhop*this->patch().magSf());
    }
    else
    {
        return phip/this->patch().magSf();
    }
}
```

The function can take as input either a massflow rate or a velocity, and in both cases it uses the flux and simply returns the velocity at the boundary face.
The `updateCoeffs()` function is way more complex and long since it has to return the output of the procedure shown in Equations (1.44) and (1.45). This changes based on whether one wants a perfectly or partially non-reflecting condition, and also based on the time discretization scheme that is being applied. For the sake of brevity, only the parts of the function concerning the Euler time discretization are reported:

main functions of the `mixedFvPatchField` class

```
template<class Type>
void Foam::advectiveFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const fvMesh& mesh = this->internalField().mesh();

    word ddtScheme
    (
        mesh.ddtScheme(this->internalField().name())
    );
    scalar deltaT = this->db().time().deltaTValue();

    const GeometricField<Type, fvPatchField, volMesh>& field =
        this->db().objectRegistry::template
        lookupObject<GeometricField<Type, fvPatchField, volMesh>>
        (
            this->internalField().name()
        );

    // Calculate the advection speed of the field wave
    // If the wave is incoming set the speed to 0.
    const scalarField w(Foam::max(advectionSpeed(), scalar(0)));

    // Calculate the field wave coefficient alpha (See notes)
    const scalarField alpha(w*deltaT*this->patch().deltaCoeffs());

    label patchi = this->patch().index();

    // Non-reflecting outflow boundary
```

```
    // If lInf_ defined setup relaxation to the value fieldInf_.
    if (lInf_ > 0)
    {
        // Calculate the field relaxation coefficient k (See notes)
        const scalarField k(w*deltaT/lInf_);

        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValue() =
            (
                field.oldTime().boundaryField()[patchi] + k*fieldInf_
            )/(1.0 + k);

            this->valueFraction() = (1.0 + k)/(1.0 + alpha + k);
        }
        else if (ddtScheme == fv::backwardDdtScheme<scalar>::typeName)
        {
            .
            .
            .
        }
        else if
        (
            ddtScheme == fv::localEulerDdtScheme<scalar>::typeName
        )
        {
            .
            .
            .
        }
        .
        .
        .
    }
    else
    {
        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValue() = field.oldTime().boundaryField()[patchi];

            this->valueFraction() = 1.0/(1.0 + alpha);
        }
        else if (ddtScheme == fv::backwardDdtScheme<scalar>::typeName)
        {
            this->refValue() =
            (
                2.0*field.oldTime().boundaryField()[patchi]
              - 0.5*field.oldTime().oldTime().boundaryField()[patchi]
            )/1.5;

            this->valueFraction() = 1.5/(1.5 + alpha);
        }
        else if
        .
        .
        .
    }

    mixedFvPatchField<Type>::updateCoeffs();
}
```

In line *211*, the advection speed is defined through the `advectionSpeed()` function of the same class and is then used to define the parameter `alpha` (line 214) as

$$\alpha = \frac{w\Delta t}{\mathbf{d}}, \tag{2.3}$$

which is exactly the one introduced in Equation (1.44). Then, an `if` statement is introduced to separate the perfectly or partially non-reflecting cases, according to whether the `lInf` parameter is defined or not.
In the case of `lInf_` > 0 (partially non-reflecting), the function first defines parameter `k` as

$$k = \frac{w\Delta t}{\mathrm{lInf}}, \tag{2.4}$$

and then changes the values of `refValue` and `valueFraction` which are member data inherited from the base class `mixedFvPatchField` as

$$\mathrm{refValue} = (\phi_f + k\phi^{\infty})\frac{1}{1+k} \tag{2.5}$$

$$\mathrm{valueFraction} = \frac{1+k}{1+\alpha+k} \tag{2.6}$$

It is easily verified that when inserting this values in Equation (2.2) the result is exactly that obtained in Equation (1.45) which shows that the way the advective boundary conditions operate is consistent with the theory. By applying the same procedure to the other *if* statement it is easy to verify that the same can be said for the case with `lInf` $= 0$ (perfectly non-reflecting case).

2.2.2 WaveTransmissive boundary conditions in OpenFOAM

The `waveTransmissive` boundary condition is also found in the `fvPatchFields/derived` directory and it inherits directly from the `advective` boundary condition. The first rows of definition file are here presented together with the class' member data:

member data of the `waveTransmissiveFvPatchField` class

```
template<class Type>
class waveTransmissiveFvPatchField
:
    public advectiveFvPatchField<Type>
{

    // Private data

        //- Name of the compressibility field used to calculate the wave speed
        word psiName_;

        //- Heat capacity ratio
        scalar gamma_;
```

The class only has one function called `advectionSpeed()` which overrides the same function of its base class `advectiveFvPatchField`. The definition of this function is:

`advectionSpeed` function of the `waveTransmissiveFvPatchField` class

```
template<class Type>
Foam::tmp<Foam::scalarField>
Foam::waveTransmissiveFvPatchField<Type>::advectionSpeed() const
{
    // Lookup the velocity and compressibility of the patch
    const fvPatchField<scalar>& psip =
        this->patch().template
            lookupPatchField<volScalarField, scalar>(psiName_);

```

```
    const surfaceScalarField& phi =
        this->db().template lookupObject<surfaceScalarField>(this->phiName_);

    fvsPatchField<scalar> phip =
        this->patch().template
            lookupPatchField<surfaceScalarField, scalar>(this->phiName_);

    if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchScalarField& rhop =
            this->patch().template
                lookupPatchField<volScalarField, scalar>(this->rhoName_);

        phip /= rhop;
    }

    // Calculate the speed of the field wave w
    // by summing the component of the velocity normal to the boundary
    // and the speed of sound (sqrt(gamma_/psi)).
    return phip/this->patch().magSf() + sqrt(gamma_/psip);
}
```

since `phip` is the flux at the face, when divided by the area of the face gives the velocity **parallel to the patch** (the flux in OpenFOAM is calculated as $\mathbf{U} \cdot \mathbf{S}_f$). The advection speed is hence obtained as

$$w = U + \sqrt{\frac{\gamma}{\psi}},$$

where ψ is the compressibility and in general is simply given by $\psi = p/\rho$, resulting in an advection speed which is $w = U + c$.
Since the `waveTransmissive` boundary condition does not override any other function, it provides the same implementation of the `advective` boundary condition but with a different advection speed.

2.3 Usage of non-reflecting boundary conditions in OpenFOAM

It has been shown that the way non-reflecting boundary conditions work in OpenFOAM is by applying the LODI relations, and it is up to the user to apply the `advective` or the `waveTransmissive` boundary conditions to each variable.
According to the theory discussed in Section 1.2.1, the way these boundary conditions should be used is by applying an `advective` boundary condition to the variables that travel with an advection speed equal to u and a `waveTransmissive` boundary conditions to those that travel with $u+c$. When reviewing the tutorials available in the OpenFOAM-v2112 version though, the `waveTransmissive` boundary condition is used in 10 tutorials and is always only applied to the pressure while for velocity, temperature and other variables the `inletOutlet` boundary condition is used.
In order to understand why this choice has been made in the tutorials and how to properly use this boundary conditions, two test cases have been studied:

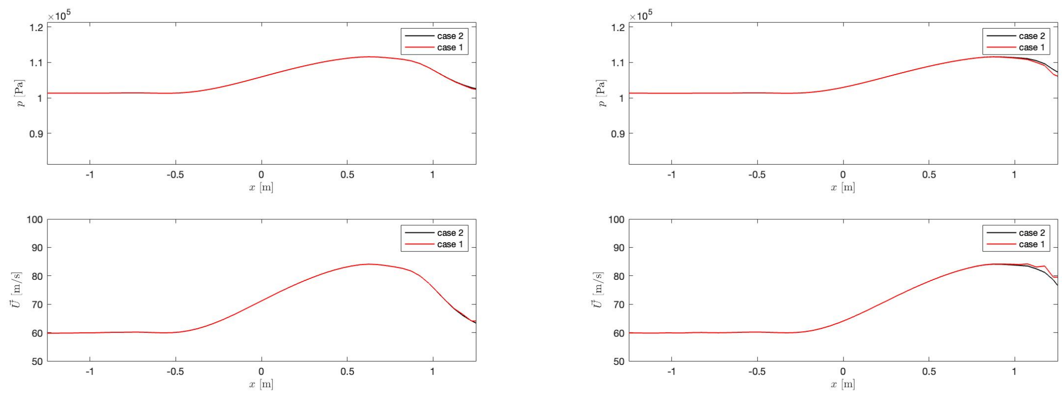
- A 1-D duct with a sinusoidal pressure inlet, for which a simple custom boundary condition has been implemented in order to have a sinusoidal input with a single period of oscillation (source code in appendix).
- A 2-D square domain with a fixed temperature constraint at the center of 3000 K (a so-called "spark") acting for some instants, in order to cause a pressure wave traveling towards the output.

For both test cases, the `rhoPimpleFoam` unsteady compressible solver is used and the simulations are carried out with two different sets of boundary conditions for the outflow shown in Table 2.1

Table 2.1: Outflow conditions

	Case 1	**Case 2**
p	waveTransmissive	waveTransmissive
U	pressureInletOutletVelocity	waveTransmissive
T	inletOutlet	advective

Case 1 sets the outflow conditions as is done in the OpenFOAM tutorials, while Case 2 are the conditions that should be applied according to theory. The `lInf` parameter is set to the length of the duct for both cases and for both `waveTransmissive` and `advective` boundary conditions. The results of the first simulation in terms of pressure at the center of the duct are shown for two successive time steps in fig. 2.1.



(a) t_0 (b) t_1

Figure 2.1: Pressure and velocity oscillation in a 1-D duct with two different sets of outlet boundary conditions

The results show that for Case 1, the behavior in terms of reflection at the output is far worse than that of Case 2, with both pressure and velocity showing some clear disturbances in the shape of the bell at the outflow, which results in a reflection of the wave into the computational domain. It is worth pointing out that this discrepancy between the two solutions tends to decrease the more the mesh is refined, with *case 1* boundary conditions providing a solution almost as good as *case 2* when the number of cells is quadrupled with respect to the mesh presented here which has 50 cells in the longitudinal direction.

This difference in behavior between the two cases is even more visible in the second, two-dimensional test case for which the results reported in fig. 2.2 show almost no reflection of the pressure wave for *case 2*, and a very prominent reflection at all four boundaries for Case 1.
As a result, it is clear that, for compressible simulations, the correct way to utilize the currently implemented non-reflecting boundary conditions in OpenFOAM is to apply a `waveTransmissive` boundary condition to pressure and velocity, and an `advective` boundary condition to the other variables.

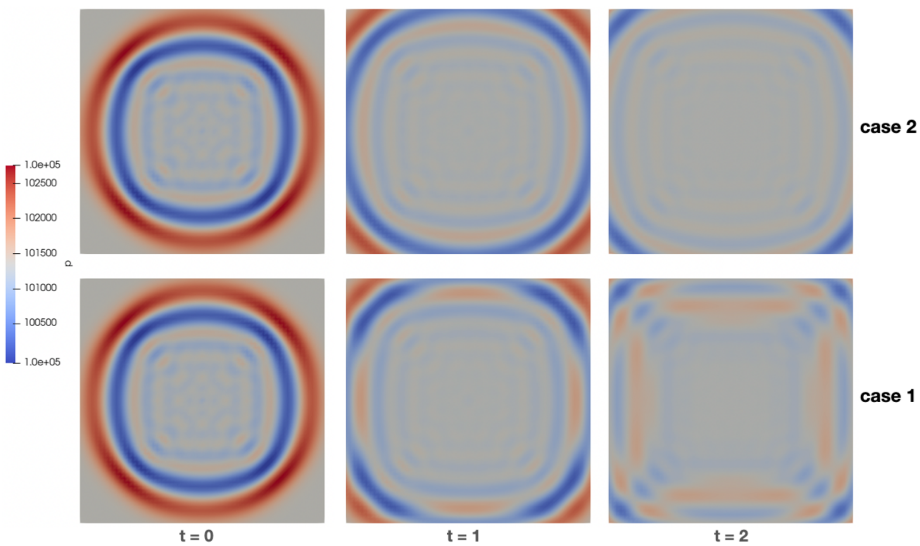


Figure 2.2: Propagation of pressure waves in the domain in the two cases

Chapter 3

Implementation of the custom non-reflecting boundary conditions

The way to properly apply non-reflecting boundary conditions in OpenFOAM has been discussed in Section 2.3, which is to use `waveTransmissive` for velocity and pressure and `advective` for the other variables (T, species Y_i etc.). The problem with this approach is that according to the LODI theory and as shown in Equations (1.30), (1.31) and (1.32), the only velocity component that should travel with an advection speed $w = U + c$, and therefore be treated with a `waveTransmissive` boundary condition, is the one normal to the outlet patch. Instead, the components of the velocity orthogonal to the outlet patch should travel with $w = U$ and hence an `advective` boundary condition.
One possible solution to fix this problem is modifying the `mixed` boundary condition so that instead of simply solving the transport equation with a given advection speed for the whole velocity vector, it first projects the velocity on a reference frame normal to the patch, transports component of the velocity normal to the patch u_n with a $U + c$ advection speed and the tangential components with u, and finally projects the results back to the Cartesian reference frame. This approach is indeed the one that has been implemented in the present work. The "improved" *LODI* boundary conditions have been implemented for a two-dimensional case in order to preliminarily study the behavior and consistency of this theory, with the intention to eventually extend the application to a general three-dimensional case.

3.1 Necessary modifications to the OpenFOAM approach

What currently happens in OpenFOAM is that the velocity vector is taken into account as a whole, and for a 2-D case it is hence transported at the boundary following

$$\begin{bmatrix} u \\ v \end{bmatrix}_f^{n+1} = \begin{bmatrix} u \\ v \end{bmatrix}_f^{n} \frac{1}{1+\alpha} + \begin{bmatrix} u \\ v \end{bmatrix}_c^{n+1} \frac{\alpha}{1+\alpha}, \tag{3.1}$$

with $\alpha = \delta t U_n/\mathbf{d}$ if the boundary condition is `advective` and $\alpha = \delta t(U_n + c)/\mathbf{d}$ if the boundary condition is `waveTransmissive`.
What we want to do is instead transporting the velocity components normal and tangential to every boundary face separately, with different advection speeds. This can be done by identifying the patch normal vector $\mathbf{n}$ and projecting the velocity onto it. Figure 3.1 shows that the components of the patch normal and tangential vectors $\mathbf{n}$ and $\mathbf{t}$ in the Cartesian reference frame are

$$\mathbf{n} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}, \quad \mathbf{t} = \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}, \tag{3.2}$$

These can be used to project the velocity vector and obtain u_n and u_t as

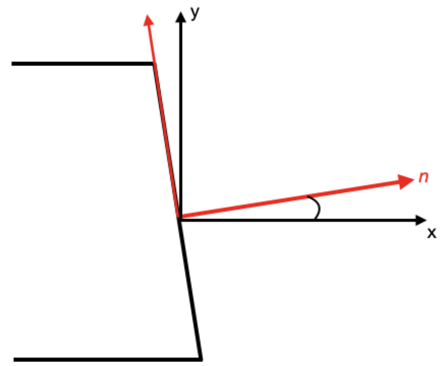


Figure 3.1: Patch normal vector with respect to the Cartesian reference frame

$$\begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}, \tag{3.3}$$

Once these two velocity components are calculated, they have to be transported at the boundary face according to the LODI relations (1.30), (1.31), which for a fully non-reflecting case are

$$\frac{\partial u_n}{\partial t} + (u_n + c)\frac{\partial u_n}{\partial n} = 0, \tag{3.4}$$

When discretized through an Euler time discretization this yields

$$(u_n)_f^{n+1} = (u_n)_f^n \frac{1}{1+\alpha_{uc}} + \frac{\alpha_{uc}}{1+\alpha_{uc}}(u_n)_c^{n+1}, \tag{3.5}$$

$$(u_t)_f^{n+1} = (u_t)_f^n \frac{1}{1+\alpha_u} + \frac{\alpha_u}{1+\alpha_u}(u_t)_c^{n+1}, \tag{3.6}$$

with $\alpha_u = \frac{\delta t u_n}{\mathbf{d}}$ and $\alpha_{uc} = \frac{\delta t(u_n+c)}{\mathbf{d}}$. Once the velocity components are transported, it's necessary to transform them back the Cartesian reference frame in order to finally obtain the values of the velocity at the boundary face

$$\begin{bmatrix} u \\ v \end{bmatrix}_f^{n+1} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u_n \\ u_t \end{bmatrix}_f^{n+1} = (u_n)_f^{n+1} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + (u_t)_f^{n+1} \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}, \tag{3.7}$$

This procedure has been implemented in the custom non-reflecting boundary conditions, namely in the two classes "`basic/mixedV2D`" and "`derived/LODI2D`" which have been created by modifying, adding functionalities and merging with the `mixed`, `advective` and `waveTransmissive` boundary conditions.

3.2 Implementation of the modified `mixedFvPatchField` boundary conditions

The mixed boundary condition is a templated class, meaning that it is compiled through a series of macros that allows for the boundary condition to be applicable to different data types. The template parameter "Type" serves as a keyword that can change its meaning (scalar, vector, etc.) in order to

make the implementation more dynamic and flexible.
The new `mixedV2D` boundary condition does not need this flexibility since it is supposed to be applied only to the velocity, hence the "Type" parameter has to be "vector". In order to make the new boundary condition applicable only for vector fields, the class from which it inherits has to be changed from the templated class `fvPatchField<Type>` to `fvPatchVectorField`. The definition of the member data of the class and the output of its functions also have to change, in particular the five functions: `valueInternalCoeffs`, `valueBoundaryCoeffs`, `gradientInternalCoeffs`, `gradientBoundaryCoeffs` and `snGrad`, are all of type `virtual tmp<Field<Type>>` since they have to provide an output that is coherent with the type of field that is being considered. These functions all have to be changed into type `virtual tmp<vectorField>` (where `tmp` indicates a smart pointer and `virtual` indicates that these functions are dynamically binded).
As shown in Sections 2.2, and 2.2.1, in the case of a vector type of field, the functions currently use the member data `refValue_`, `refGrad_` and `valueFraction_` to apply the transport equation to the field at the boundary with *one advection speed* for all components of the vector. This has to be changed in order to apply a different advection speed to every component, hence these functions have to be defined twice, once for the $u + c$ advection speed and once for u.
The declaration of the member data of the new "`mixedV2DFvPatchVectorField`" is here presented:

Member data declaration of the `mixedV2DFvPatchVectorField` class

```
class mixedV2DFvPatchVectorField
:
    public fvPatchVectorField
{
    // Private data

        //- Velocity normal to the patches
        scalarField Un_;

        //- Velocity tangential to the patches
        scalarField Ut_;

        //- Patch normal vector
        vectorField n_;

        //- Value field for the quantities that travel with U
        scalarField refValueU_;

        //- Value field for the quantities that travel with U +- C
        scalarField refValueUC_;

        //- Normal gradient field
        vectorField refGrad_;

        //- valueFraction calculated with velocity U
        scalarField valueFractionU_;

        //- valueFraction calculated with velocity U +- C
        scalarField valueFractionUC_;

        //- Source field
        vectorField source_;

        //- First vector for coordinate change
        vectorField vector1_;

        //- Second vector for coordinate change
        vectorField vector2_;

        //- Third vector for coordinate change
        vectorField vector3_;
```

Where

- `Un_` and `Ut_` are the velocities normal and tangential to the patch.

- `n_` is the patch normal vector.
- `refValueU_` and `refValueUC_` are the value fields necessary to build the transport equations of the normal and tangential velocity components with different advection speeds (note that the type of these data is now `scalarField` since they have to be applied to velocity components not vectors).
- `valueFractionU_` and `valueFractionUC_` are the weights necessary to build the transport equations of the normal and tangential velocity components.
- `vector1_`, `vector2_` and `vector3_` are the three vectors that define the rotation matrix to go back to Cartesian coordinates (the functions of the class provide outputs that are directly applied to the velocity components in the Cartesian coordinates).

The definitions of the constructors in the `*.C` file have to be modified for initializing the newly added variables, one of the modified constructors from the file `mixedV2DFvPatchVectorField.C` is reported here:

Definition of the first constructor in the file `mixedV2DFvPatchVectorField.C` class

```
Foam::mixedV2DFvPatchVectorField::mixedV2DFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fvPatchVectorField(p, iF),
    Un_(p.size()),
    Ut_(p.size()),
    n_(p.size()),
    refValueU_(p.size()),
    refValueUC_(p.size()),
    refGrad_(p.size()),
    valueFractionU_(p.size()),
    valueFractionUC_(p.size()),
    source_(p.size(), Zero),
    vector1_(p.size()),
    vector2_(p.size()),
    vector3_(p.size())
{
    n_ = this->patch().nf();
    forAll(vector1_, i)
    {
        vector1_[i][0] = n_[i][0];
        vector1_[i][1] = n_[i][1];
        vector1_[i][2] = n_[i][2];
    }
    forAll(vector2_, i)
    {
        vector2_[i][0] = -n_[i][1];
        vector2_[i][1] = n_[i][0];
        vector2_[i][2] = n_[i][2];
    }
    forAll(vector3_, i)
    {
        vector3_[i][0] = 0.0;
        vector3_[i][1] = 0.0;
        vector3_[i][2] = 1.0;
    }
}
```

Where from the initialization of the member data it can be seen that `vector1_`, `vector2_` and `vector3_` are the vectors obtained in Equation (3.7) that allow to go back from the patch to the Cartesian reference frame (`vector3_` being simply $[0, 0, 1]^T$ since this implementation is for a 2-D case).

All of the six member functions of the class have also been modified in order to perform the operations described in Section 3.1 and are shown here:

Definition of the `evaluate` function of the `mixedV2DFvPatchVectorField` class

```
void Foam::mixedV2DFvPatchVectorField::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    vectorField n = this->patch().nf();
    vectorField U = this->patchInternalField();
    scalarField Un = Un_;
    scalarField Ut = Ut_;
    forAll(U, i)
    {
        Un[i] = U[i][0]*n[i][0] + U[i][1]*n[i][1]; //ucos+vsin
        Ut[i] = -U[i][0]*n[i][1] + U[i][1]*n[i][0]; //-usin+vcos
    }

    Foam::scalarField valueU =
    (
        valueFractionU_*refValueU_
      + (1.0 - valueFractionU_)
       *(
            Ut
        )
    );

    Foam::scalarField valueUC =
    (
        valueFractionUC_*refValueUC_
      + (1.0 - valueFractionUC_)
       *(
            Un
        )
    );

    vectorField::operator=
    (
    vector1_ * valueUC + vector2_ * valueU
    );

    fvPatchVectorField::evaluate();
}
```

In the `evaluate` function, first of all the normal and tangential velocity components are obtained as in Equation (3.3), then two scalar fields `valueU` and `valueUC` representing the right hand sides of Equations (3.5) and (3.6) are created, and finally the output of the function is obtained multiplying these values for the two vectors, as in Equation (3.7).

Definition of the `snGrad` function of the `mixedV2DFvPatchVectorField`

```
Foam::tmp<Foam::vectorField>
Foam::mixedV2DFvPatchVectorField::snGrad() const
{
    vectorField n = this->patch().nf();
    vectorField U = this->patchInternalField();
    scalarField Un = Un_;
    scalarField Ut = Ut_;
    forAll(U, i)
    {
        Un[i] = U[i][0]*n[i][0] + U[i][1]*n[i][1]; //ucos+vsin
        Ut[i] = -U[i][0]*n[i][1] + U[i][1]*n[i][0]; //-usin+vcos
    }

```

```
    Foam::scalarField valueU =
        (valueFractionU_
       *(refValueU_ - Ut)
       *this->patch().deltaCoeffs());

    Foam::scalarField valueUC =
        (valueFractionUC_
       *(refValueUC_ - Un)
       *this->patch().deltaCoeffs());

    return
        vector1_ * valueUC + vector2_ * valueU;
}
```

The `snGrad` function, which calculates the patch normal gradient at the face, is modified similarly to the `evaluate` function, through the usage of the normal and tangential velocity components.

Definition of the other member functions of the `mixedV2DFvPatchVectorField` class

```
Foam::tmp<Foam::vectorField>
Foam::mixedV2DFvPatchVectorField::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    scalarField valueU =
    (1.0 - valueFractionU_);

    scalarField valueUC =
    (1.0 - valueFractionUC_);

    return vector1_ * valueUC + vector2_ * valueU;
}


Foam::tmp<Foam::vectorField>
Foam::mixedV2DFvPatchVectorField::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    scalarField valueU =
        valueFractionU_*refValueU_;

    scalarField valueUC =
        valueFractionUC_*refValueUC_;

    return vector1_ * valueUC + vector2_ * valueU;
}


Foam::tmp<Foam::vectorField>
Foam::mixedV2DFvPatchVectorField::gradientInternalCoeffs() const
{
    scalarField valueU =
         -valueFractionU_*this->patch().deltaCoeffs();

    scalarField valueUC =
         -valueFractionUC_*this->patch().deltaCoeffs();

    return vector1_ * valueUC + vector2_ * valueU;
}


Foam::tmp<Foam::vectorField>
Foam::mixedV2DFvPatchVectorField::gradientBoundaryCoeffs() const
{
    scalarField valueU =
```

```
        valueFractionU_*this->patch().deltaCoeffs()*refValueU_;

    scalarField valueUC =
        valueFractionUC_*this->patch().deltaCoeffs()*refValueUC_;

    return vector1_ * valueUC + vector2_ * valueU;
}
```

The remaining functions are also modified for projecting and dividing the contributions of the different components of the velocity.

3.3 Implementation of the LODI2D boundary condition

The member variables `valueFractionU_`, `valueFractionUC_`, `refValueU_` and `refValueUC_` that are used inside the member functions of `mixedV2D` are not defined inside the class itself but are instead defined inside a second class that inherits from `mixedV2D` called `LODI2D`. This is done in order to keep the same structure adopted by OpenFOAM when dealing with non-reflecting boundary conditions. `advective` and `waveTransmissive` boundary conditions are also templated classes, while the `LODI2D` class which merges these two and adds a series of functionalities is supposed to be used only for vectorField type of inputs, meaning that the declaration and definition of all member data and member functions of the `advective` class have to be modified. Moreover, as discussed previously, the class has to take into account both a transport of the velocity component with the two different advection speeds.
The declaration of the member data of the new `LODI2D` class is reported here:

Member data declaration of the `LODI2DFvPatchVectorField` class

```
class LODI2DFvPatchVectorField
:
    public mixedV2DFvPatchVectorField
{
protected:

    // Private data

        //- Normal velocity vector at old time
        scalarField Unold_;

        //- Normal velocity vector at oold time
        scalarField Unoold_;

        //- Tangential velocity vector at old time
        scalarField Utold_;

        //- Tangential velocity vector at oold time
        scalarField Utoold_;

        //- Normal Velocity at infinite for every patch
        scalarField UnInf_;

        //- Name of the flux transporting the field
        word phiName_;

        //- Name of the density field used to normalise the mass flux
        //- if necessary
        word rhoName_;

        //- Field value of the far-field
        vector fieldInf_;

        //- Relaxation length-scale
        scalar lInf_;

        //- waveTransmissive member data ------------------------------------
```

```

        //- Name of the compressibility field used to calculate the wave speed
        word psiName_;

        //- Heat capacity ratio
        scalar gamma_;
```

Where

- `Unold_`, `Utold_`, `Unoold_` and `Utoold_` are the normal and tangential velocity vectors at the previous and even previous times.
- `fieldInf_` is the field (the velocity in this case) at infinity.
- `UnInf_` is the component of the velocity at infinity normal to the boundary patch.
- `lInf_` is the relaxation length used to calculate the strength of the reflecting wave when considering partially non-reflecting boundary conditions.

Once again, the definitions of the constructors have been modified to take into account the new member data:

Definition of a constructor in the file `LODI2DFvPatchVectorField.C` class

```
Foam::LODI2DFvPatchVectorField::LODI2DFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    mixedV2DFvPatchVectorField(p, iF),
    Unold_(p.size()),
    Unoold_(p.size()),
    Utold_(p.size()),
    Utoold_(p.size()),
    UnInf_(p.size()),
    phiName_(dict.getOrDefault<word>("phi", "phi")),
    rhoName_(dict.getOrDefault<word>("rho", "rho")),
    fieldInf_(Zero),
    lInf_(-GREAT),
    psiName_(dict.getOrDefault<word>("psi", "thermo:psi")),
    gamma_(dict.get<scalar>("gamma"))
{
    if (dict.found("value"))
    {
        fvPatchVectorField::operator=
        (
            vectorField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchVectorField::operator=(this->patchInternalField());
    }
    vectorField U = this->patchInternalField();
    scalarField Un = Unold_; //initialize them
    scalarField Ut = Utold_;
    vectorField n = this->n();
    forAll(Un, i)
    {
    Un[i] = U[i][0]*n[i][0] + U[i][1]*n[i][1];
    Ut[i] = -U[i][0]*n[i][1] + U[i][1]*n[i][0];
    }

    this->refValueU() = Un;
    this->refValueUC() = Ut;
```

```
    this->refGrad() = Zero;
    this->valueFractionU() = 0.0;
    this->valueFractionUC() = 0.0;

    if (dict.readIfPresent("lInf", lInf_))
    {
        dict.readEntry("fieldInf", fieldInf_);

        if (lInf_ < 0.0)
        {
            FatalIOErrorInFunction(dict)
                << "unphysical lInf specified (lInf < 0)" << nl
                << "    on patch " << this->patch().name()
                << " of field " << this->internalField().name()
                << " in file " << this->internalField().objectPath()
                << exit(FatalIOError);
        }
    }
}
```

Where, once again, the construction of the scalar fields containing the components of the velocity normal and tangential to the boundary patch can be seen, together with an error message that occurs if the user defines a negative value for `lInf_`.
The `LODI2D` class has three member functions, two of them are the `advectionSpeed` functions of the `advective` and the `waveTransmissive` classes, the first of which returns a scalar field containing u_n, the second one containing $u_n + c$ for every boundary patch.

`advectionSpeed()` and `advectionSpeedWT` functions of the `LODI2DFvPatchVectorField` class

```
//- Adective advectionSpeed function ---------------------------------------
Foam::tmp<Foam::scalarField>
Foam::LODI2DFvPatchVectorField::advectionSpeed() const
{
    const surfaceScalarField& phi =
        this->db().objectRegistry::template lookupObject<surfaceScalarField>
        (phiName_);

    fvsPatchField<scalar> phip =
        this->patch().template lookupPatchField<surfaceScalarField, scalar>
        (
            phiName_
        );

    if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchScalarField& rhop =
            this->patch().template lookupPatchField<volScalarField, scalar>
            (
                rhoName_
            );

        return phip/(rhop*this->patch().magSf());
    }
    else
    {
        return phip/this->patch().magSf();
    }
}

//- WaveTransmissive advectionSpeed function ------------------------------
Foam::tmp<Foam::scalarField>
Foam::LODI2DFvPatchVectorField::advectionSpeedWT() const
{
    // Lookup the velocity and compressibility of the patch
    const fvPatchField<scalar>& psip =
        this->patch().template
            lookupPatchField<volScalarField, scalar>(psiName_);
```

```

    const surfaceScalarField& phi =
        this->db().template lookupObject<surfaceScalarField>(this->phiName_);

    fvsPatchField<scalar> phip =
        this->patch().template
            lookupPatchField<surfaceScalarField, scalar>(this->phiName_);

    if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchScalarField& rhop =
            this->patch().template
                lookupPatchField<volScalarField, scalar>(this->rhoName_);

        phip /= rhop;
    }

    // Calculate the speed of the field wave w
    // by summing the component of the velocity normal to the boundary
    // and the speed of sound (sqrt(gamma_/psi)).
    return phip/this->patch().magSf() + sqrt(gamma_/psip); // U + C
}
```

The last and most important function is the `updateCoeffs` function, responsable of updating the coefficients shown in the `mixedV2D` class that provide the output of the most important functions. The first part of the `updateCoeffs` function is shown here

First part of the `updateCoeffs` function of the `LODI2DFvPatchVectorField` class

```
void Foam::LODI2DFvPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const fvMesh& mesh = this->internalField().mesh();


    word ddtScheme
    (
        mesh.ddtScheme(this->internalField().name())
    );
    scalar deltaT = this->db().time().deltaTValue();

    const GeometricField<vector, fvPatchField, volMesh>& field =
        this->db().objectRegistry::template
        lookupObject<GeometricField<vector, fvPatchField, volMesh>>
        (
            this->internalField().name()
        );

    // Calculate the advection speed of the field wave
    // If the wave is incoming set the speed to 0.
    // advection speed U
    const scalarField wU(Foam::max(advectionSpeed(), scalar(0)));
    // advection speed U +- C
    const scalarField wUC(Foam::max(advectionSpeedWT(), scalar(0)));

    // Calculate the field wave coefficient alpha with U and U+-C(See notes)
    const scalarField alphaU(wU*deltaT*this->patch().deltaCoeffs());
    const scalarField alphaUC(wUC*deltaT*this->patch().deltaCoeffs());

    label patchi = this->patch().index();

    vectorField Uold = field.oldTime().boundaryField()[patchi];
    vectorField Uoold = field.oldTime().oldTime().boundaryField()[patchi];

```

```
    vectorField n = this->n();
    vectorField t = n;
    forAll(t, i)
    {
    t[i][0] = -n[i][1];
        t[i][1] = n[i][0];
    }

    forAll(Unold_, i)
    {
    Unold_[i] = Uold[i][0]*n[i][0] + Uold[i][1]*n[i][1];
    Utold_[i] = -Uold[i][0]*n[i][1] + Uold[i][1]*n[i][0];
    Unoold_[i] = Uoold[i][0]*n[i][0] + Uoold[i][1]*n[i][1];
    Utoold_[i] = -Uoold[i][0]*n[i][1] + Uoold[i][1]*n[i][0];
    }

    forAll(UnInf_, i)
    {
    UnInf_[i] = fieldInf_[0]*n[i][0] + fieldInf_[1]*n[i][1];
    }
```

Where the two advection speeds are introduced using the two `advectionSpeed` functions, then the two field wave coefficients `alphaU` and `alphaUC` are calculated, which correspond exactly to the α_u and α_{uc} of Equations (3.5) and (3.6). The scalar fields `Unold`, `Utold`, `Unoold`, `Utoold` and finally the velocity of the far field `UnInf` are calculated, the latter by using the `fieldInf_` variable provided by the user, which is the whole velocity vector in the Cartesian coordinates and must therefore also be projected.
As shown for the `advective` class, the core of the function corresponds to a series of `if` statements that fill the `valueFraction_` and `refValue_` scalar fields with the quantities corresponding to Equations (3.5) and (3.6), the first of which (the one corresponding to a partially non-reflecting condition with `lInf` $\neq 0$ and an Euler time discretization) is reported here:

First part of the `if` statement of the `updateCoeffs` function

```
    if (lInf_ > 0)
    {
        // Calculate the field relaxation coefficient k (See notes)
        // K calculated with the advection speed U +- C (not U)
        const scalarField k(wUC*deltaT/lInf_); // was calculated with wU initially

        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValueU() = Utold_;

            this->refValueUC() =
            (
                Unold_ + k*UnInf_
            )/(1.0 + k);

            this->valueFractionU() = 1.0/(1.0 + alphaU);

            this->valueFractionUC() = (1.0 + k)/(1.0 + alphaUC + k);
        }
```

This quantities, when inserted in the expression of the `evaluate` function of the `mixedV2D` class shown before, yield the following expressions for the normal and tangential velocity components

$$(u_n)_f^{n+1} = ((u_n)_f^n + k(u_n)^\infty)\frac{1}{1+\alpha_{uc}+k} + \frac{\alpha_{uc}}{1+\alpha_{uc}+k}(u_n)_c^{n+1}, \tag{3.8}$$

$$(u_t)_f^{n+1} = (u_t)_f^n \frac{1}{1+\alpha_u} + \frac{\alpha_u}{1+\alpha_u}(u_t)_c^{n+1}, \tag{3.9}$$

Meaning that the normal component of the velocity is being transported with $u_n + c$, and sees a partially non-reflecting boundary having assigned a value to the wave entering the domain as per Equation (1.41), while the tangential component is transported with u and is not influenced by the upcoming wave.
The case with `lInf` $= 0$ and an Euler time discretization is shown here:

Second part of the `if` statement of the `updateCoeffs` function

```
    else // if lInf = 0
    {
        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValueU() = Utold_;

            this->refValueUC() = Unold_;

            this->valueFractionU() = 1.0/(1.0 + alphaU);

            this->valueFractionUC() = 1.0/(1.0 + alphaUC);
        }
```

When these expressions are inserted in the `evaluate` function it yields the same results shown in Equations (3.5) and (3.6).

3.4 Compilation of the custom boundary conditions

As explained at the beginning of Section 3.2, OpenFOAM's `mixed` and `advective` boundary conditions are a templated classes, meaning that they need macros and typedefs to be compiled, which are contained in the files `*FvPatchFields.H`, `*FvPatchFields.C` and `*FvPatchFieldsFwd.H`, for example:

```
finiteVolume
├── Make
│   ├── files
│   └── pptions
└── fields
    └── fvPatchFields
        └── basic
            └── mixed
                ├── mixedFvPatchField.H
                ├── mixedFvPatchField.C
                ├── mixedFvPatchFields.H
                ├── mixedFvPatchFields.C
                └── mixedFvPatchFieldsFwd.H
```

The `mixedV2D` and `LODI2D` boundary conditions however are not templated and therefore do not need these additional files for compilation. In order to compile these custom boundary conditions is sufficient to place the `mixedV2D` and `LODI2D` folders with theirs `*.C` and `*.H` files inside the user's folder

`src/finiteVolume/fields/fvPatchFields/basic`

In order to create a directory structure as the following

```
finiteVolume
└── Make
    ├── files
    └── options
```

```
└── fields
    └── fvPatchFields
        ├── basic
        │   └── mixedV2D
        │       ├── mixedV2DFvPatchVectorField.H
        │       └── mixedV2DFvPatchVectorField.C
        └── derived
            └── LODI2D
                ├── LODI2D2DFvPatchVectorField.H
                └── LODI2D2DFvPatchVectorField.C
```

Where, just like in the main installation folder, there is one only `Make` folder for all fields, with the `Make/files` that must contain:

`src/finiteVolume/Make/files`

```
fvPatchFields = fields/fvPatchFields
derivedFvPatchFields = $(fvPatchFields)/derived
basicFvPatchFields = $(fvPatchFields)/basic

$(basicFvPatchFields)/mixedV2D/mixedV2DFvPatchVectorField.C
$(derivedFvPatchFields)/LODI2D/LODI2DFvPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
```

And the `Make/options` file must contain:

`src/finiteVolume/Make/files`

```
EXE_INC = \
    -I$(LIB_SRC)/fileFormats/lnInclude \
    -I$(LIB_SRC)/surfMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = \
    -lOpenFOAM \
    -lfileFormats \
    -lmeshTools \
    -lfiniteVolume
```

Once this directory structure have been set up, the custom boundary conditions can be compiled by simply executing the `wmake` command inside the `finiteVolume` directory.

Chapter 4

Test cases setup and results

This chapter presents the setup of the test cases and some preliminary findings and results obtained by executing these cases with both OpenFOAM native non-reflecting boundary conditions and the custom `LODI2D` conditions. It is important to keep in mind that although the difference in terms of implementation and theoretical model between the custom `LODI2D` and the native `waveTransmissive` boundary conditions is substantial, most of the problems become numerically extremely similar since in most practical cases the velocity is already normal to the outlet patches, and the tangential components tend to be very small. Moreover, the here presented boundary conditions up to this point have only been implemented for a 2-D case on the $x - y$ plane, whilst turbulence and other flow phenomena are intrinsically three-dimensional.
For these reasons, defining a test case on which performing a proper study on the performance of the newly implemented boundary conditions is a procedure that certainly requires more time and focus, and will be part of future work, together with an implementation of the full three-dimensional implementation of the LODI relations.

4.1 2-D circle simulation

The first test case that has been studied is a simple two-dimensional circle with a diameter of 2 m and a temperature spark in the center that causes a series of pressure waves to travel towards the boundaries. The simulation has been performed with two different setups, once with the OpenFOAM native boundary conditions and once with the new `LODI2D` boundary conditions for the velocity as reported in Table 4.1.

Table 4.1: Boundary conditions of 2D-circle test-case

	Case 1	**Case 2**
p	waveTransmissive	waveTransmissive
U	waveTransmissive	LODI2D
T	advective	advective
lInf	10	10

The mesh, shown in figure 4.1 has been created with blockMesh and is composed of 12 blocks.

4.1.1 simulation setup

In this section the setup of the simulations will be presented, for the sake of brevity, only the most important and characteristic files will be shown explicitly, while the rest can be found and read by the reader in the accompanying files.
The simulation consists of a single-phase laminar flow. The cells on which to apply the circular fixed temperature constraint are identified through the `topoSet` utility, controlled through the

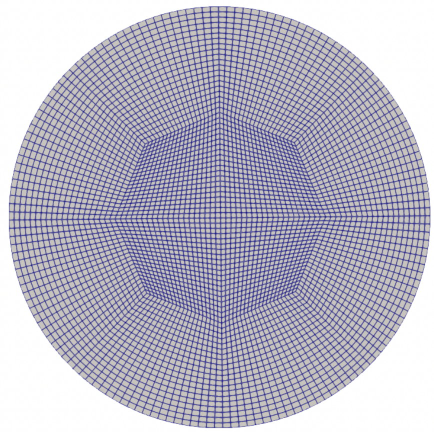


Figure 4.1: Mesh of 2D-circle test case

`system/topoSetDict` file as shown here:

`system/topoSetDict` file of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      topoSetDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //


actions
(
{
        name spark1;
        type cellSet;
        action new;

        source sphereToCell;
        sourceInfo
        {
            centre          (0.0  0.0  0.0);
            radius          0.05;
        }
}
);

// ************************************************************************* //
```

The constraint itself is assigned in the `constant/fvOptions` file, which in this case sets a fixed temperature constraint of 3000 K for a duration of 0.0002 seconds.

`constant/fvOptions` file of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
```

```
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      fvOptions;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

source1
{
    type            fixedTemperatureConstraint;

    timeStart       0;
    duration        0.0002;
    selectionMode   cellSet;
    cellSet         spark1;
    mode            uniform;
    temperature     3000;
}

// ************************************************************************* //
```

The boundary conditions are assigned in the files inside the `0` folder. For Case 2, the velocity conditions are assigned as follows:

`0/U` file of Case 2 of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    outlet
    {
        type            LODI2D;
    value       $internalField;
    field       U;
    gamma       1.4;
    rho     rho;
    lInf        10;
    fieldInf    (0 0 0);
    }

```

```
    frontAndBack
```

For Case 1 the velocity boundary conditions are:

`0/U` file of Case 1 of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    outlet
    {
        type            waveTransmissive;
    gamma       1.4;
    fieldInf    (0 0 0);
    lInf        10;
    value       $internalField;
    }

    frontAndBack
    {
        type            empty;
```

The temperature and pressure boundary conditions are identical for the two test-cases, with the pressure at the outlet being `waveTransmissive` and the temperature being `advective`, and are reported here:

`0/p` file of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 101325;

boundaryField
```

```
{
    outlet
    {
        type            waveTransmissive;
        gamma           1.4;
        fieldInf        101325;
        lInf            10;
        value           $internalField;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

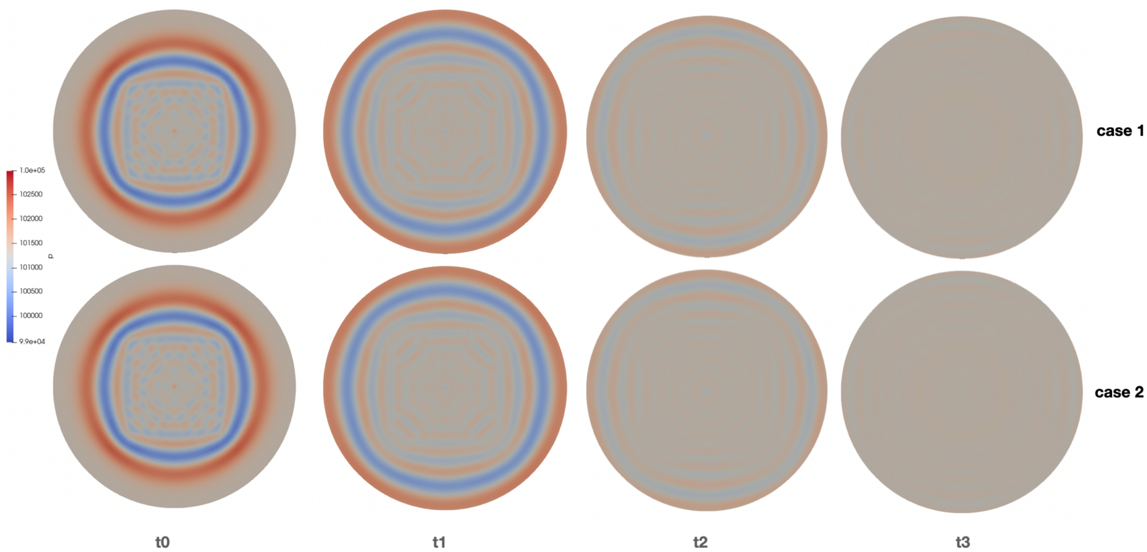
`0/T` file of the 2-D circle simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 300;

boundaryField
{
    outlet
    {
        type            advective;
    fieldInf    300;
    lInf        10;
    value       $internalField;
    }

    frontAndBack
    {
        type            empty;
    }
}


// ************************************************************************* //
```

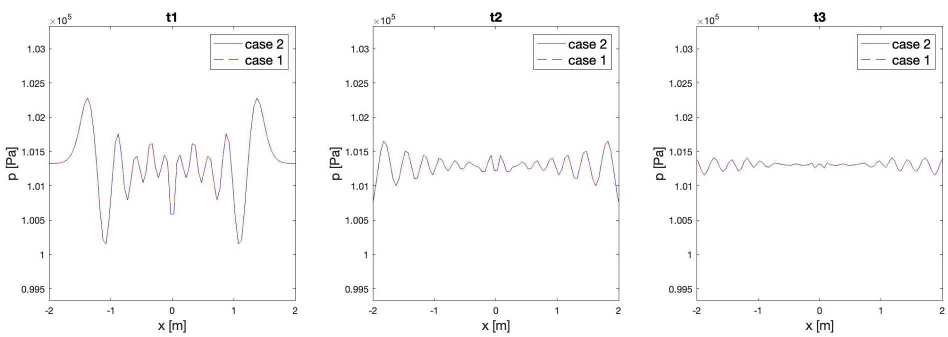
The `lInf` parameter regulates the extent of the reflected wave, ideally representing the distance after which the field will reach the **`fieldInf`** value assigned by the user. This implies that the larger `lInf` the lower the reflection at the boundaries will be. However, an extremely high value of `lInf` is not recommendable since, as explained in section 1.2.1, the complete lack of a constraint on the value of the pressure would lead to an ill-posed problem and, therefore, to a drift in the value of the pressure inside the computational domain.

4.1.2 Results of the 2-D circle simulation

The simulation has been performed with the `rhoPimpleFoam` solver for a time interval of $t = 0.01s$ and a $\Delta t = 2e-6$.
The results in terms of pressure propagating in the circular domain for four consecutive time steps are shown in Fig. 4.2a, while Fig. 4.2b shows the evolution of the waves along the horizontal axis.



(a) Propagation of pressure waves in the domain.



(b) Propagation of pressure waves along the horizontal axis.

Figure 4.2: 2D circle results for Case 1 and Case 2.

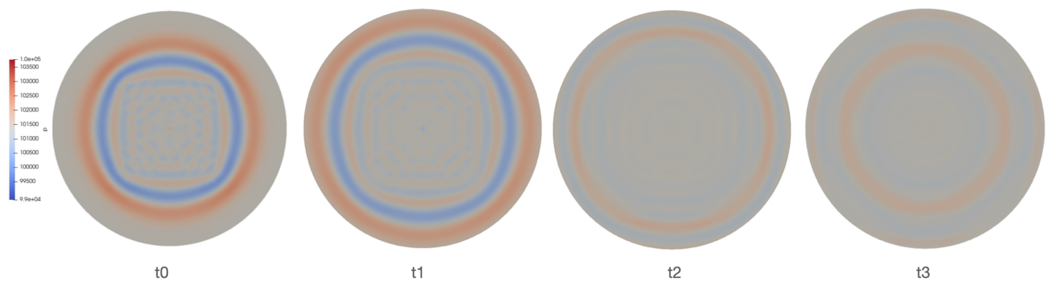
The pressure waves created by the spark travel towards the boundary and, for both cases, exit the domain with no visible reflection.
The two solutions are almost identical, which can be explained by the fact that because of the geometry of the domain, the velocity is always perfectly parallel to the outlet patches, implying that transporting both components of the velocity with an advection speed $w = u + a$ becomes equivalent to what is obtained by rotating the velocity and transporting the normal component with $u + a$ and going back to the Cartesian reference frame, since the velocity component tangential to the patch is always null and therefore gives no contribution.
In order to further investigate the correct usage of OpenFOAM's native set of non-reflecting boundary conditions, the same simulation has been also performed with the setup implemented in all the tutorials, shown in Table 4.2

The resulting pressure fields are shown in Fig. 4.3a and 4.3b where, once the pressure waves hit the boundary, a clear reflection can be seen with waves traveling back towards the center of the

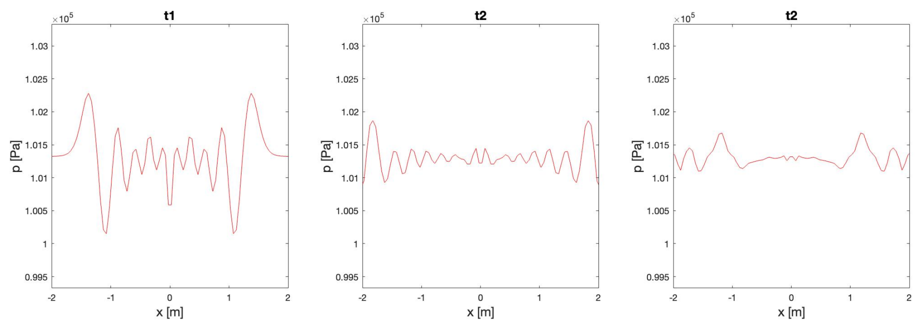
Table 4.2: Boundary conditions used by OpenFOAM tutorials

p	waveTransmissive
U	pressureInletOutletVelocity
T	inletOutlet
lInf	10

domain. This shows again that the correct way to apply OpenFOAM's native set of non-reflecting boundary conditions is that shown in Case 1 of table 4.1, namely using `waveTransmissive` for velocity and pressure, and `advective` for temperature.



(a) Propagation of pressure waves in the domain



(b) Propagation of pressure waves on the x axis.

Figure 4.3: 2D circle results for the improper boundary conditions used in OpenFOAM tutorials.

Ultimately, although this simulation gives no significant information about the performance of the custom boundary conditions with respect to OpenFOAM's `waveTransmissive` it serves as a proof of the fact that the theory is consistent, and that the newly implemented `LODI2D` conditions are indeed suited for simulating non-reflecting, non planar boundaries.

4.2 2-D turbulent flow around bluff body

The second test case that has been studied is inspired by the work done by Lysenko in 2011 [8] and Pirozzoli in 2012 [9] and consists of a bluff body inside a turbulent free-stream flow. The mesh is shown in figure 4.4.

The characteristic length of the body is $L = 0.1\ m$ while the length of the domain is $L_D = 115L$. The overall computational cost of the simulation is kept small by applying a strong mesh refinement in correspondence of the body and its wake, which limits the overall cell count to 38200 cells. The simulation is performed with a free stream Mach number of $M_\infty = 0.34$, and turbulence has been modeled through Menter's $k - \omega$ SST model [10]. Under these flow conditions, vortices are periodically shed in the bluff body wake and preventing spurious reflection of pressure waves from

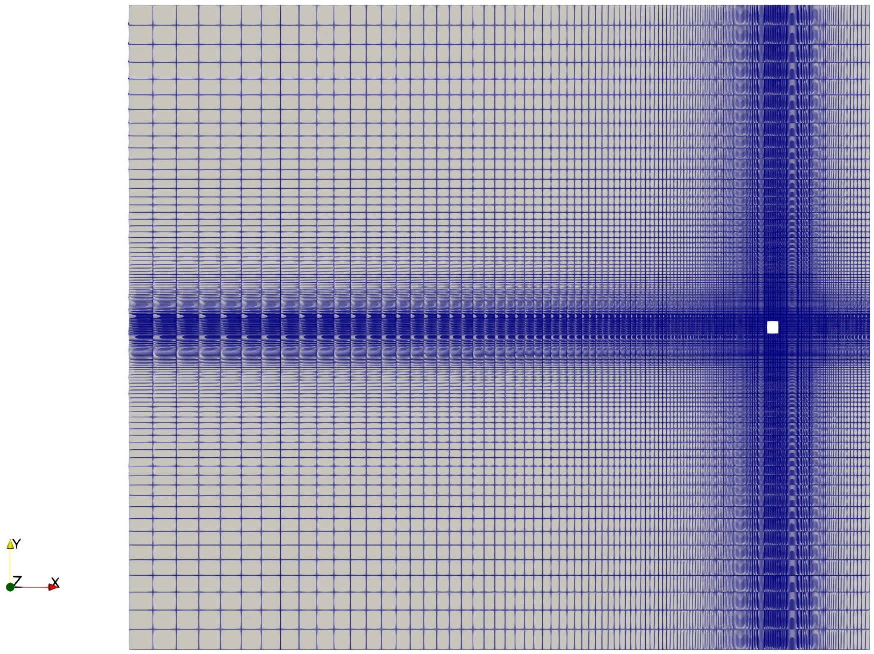


Figure 4.4: Mesh of 2-D turbulent flow around bluff body case

the outflow is challenging for numerical boundary conditions.
The patches composing the domain or the body are:

- inlet: the left boundary of the domain.
- outlet: the right boundary of the domain.
- upperAndLower: the upper and lower boundaries of the domain.
- obstacle: the walls of the bluff body.

The outflow boundary conditions for velocity, pressure and temperature are the same as those reported in Table 4.1. The full set of boundary conditions for velocity and pressure as written in the 0 folder are shown here:

0/U file of the 2-D bluff body simulation (LODI2D outlet case)

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

```

```
internalField   uniform (120 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
    value       uniform (120 0 0);
    }

    outlet
    {
        type            LODI2D;
    value       $internalField;
    field       U;
    gamma       1.4;
    rho     rho;
    lInf        10;
    fieldInf    (120 0 0);
    }

    upperAndLower
    {
        type            freestreamVelocity;
    freestreamValue $internalField;
    }

    obstacle
    {
        type            noSlip;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

For the velocity, the upper and lower walls are simulated through a `freestreamVelocity` which is an inlet-outlet condition that uses the velocity orientation to continuously blend between fixed value for normal inlet and zero gradient for normal outlet flow. This condition is designed to operate with the `freestreamPressure` condition, which has been applied to pressure as shown in the `0/p` file:

`0/p` file of the 2-D bluff body simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 101325;

boundaryField
```

```
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            waveTransmissive;
        gamma           1.4;
        fieldInf        101325;
        lInf            10;
        value           $internalField;
    }

    upperAndLower
    {
        type            freestreamPressure;
    freestreamValue $internalField;
    }

    obstacle
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}


// ************************************************************************* //
```

The complete set of boundary conditions can be found in the accompanying files.

4.2.1 Results of the 2-D turbulent flow around bluff body simulation

The simulation has been performed with the `rhoPimpleFoam` solver, with a maximum Courant number of $Co = 0.3$ and an initial timestep of $2e - 6$. The `system/controlDict` file is shown here:

`system/controlDict` file of the 2-D bluff body simulation

```
application     rhoPimpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         0.1;

deltaT          2e-6;

writeControl    runTime;

writeInterval   0.0005;

purgeWrite      0;      // was 10

writeFormat     ascii;

writePrecision  16;

writeCompression off;
```

```

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

adjustTimeStep  yes;

maxCo           0.3;

maxDeltaT       1;

functions
{
    fieldAverage
    {
        type            fieldAverage;
        libs            (fieldFunctionObjects);
        writeControl    writeTime;
        fields
        (
            U
            {
                mean        on;
                prime2Mean  on;
                base        time;
            }

            p
            {
                mean        on;
                prime2Mean  on;
                base        time;
            }

        );
    }
}

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

Where the `fieldAverage` function with the `prime2Mean on` flag provides the prime squared mean of the fields. The last line `libs ("libmyFiniteVolume.so");` is required for the solver to link to the custom boundary conditions and hence allows it to use them.

Figure 4.5 shows the evolution of the density field of the two cases during four different time steps. The overall structure of the flow is still very similar between the two cases, with two big vortices forming downstream of the body at $t0$. At $t1$, the vortices are fully detached and moving towards the outlet. At times $t3$ and $t4$ the process of the first vortex leaving the computational domain can be observed.
Other then some density waves being produced by the wake, no significant reflection phenomena can be observed at the outlet for both sets of boundary conditions.

Figure 4.6 shows the same phenomenon, but in terms of the y-component of the velocity vector U_y. It is worth noting that for this particular geometry the outlet patch is perfectly parallel to the Cartesian reference frame and the only difference between the two simulations is that in the `waveTransmissive` case, U_y at the boundary is transported with an advection speed $u + c$, while in the `LODI2D` case, it is transported with u. Regardless of this, the overall behavior of the two simulations is similar, with the vortexes leaving the domain and causing no significant reflecting waves.

Finally, the `pPrime2Mean` field, corresponding to the time-averaged square of the pressure oscillation $\overline{p'^2}$ is shown in figure 4.7. The field shows how the pressure oscillation is fully concentrated in the wave area and around the body. Once again, for both cases there is no significant contribution of the boundary to be seen for any of the two cases, confirming that the performance of both the

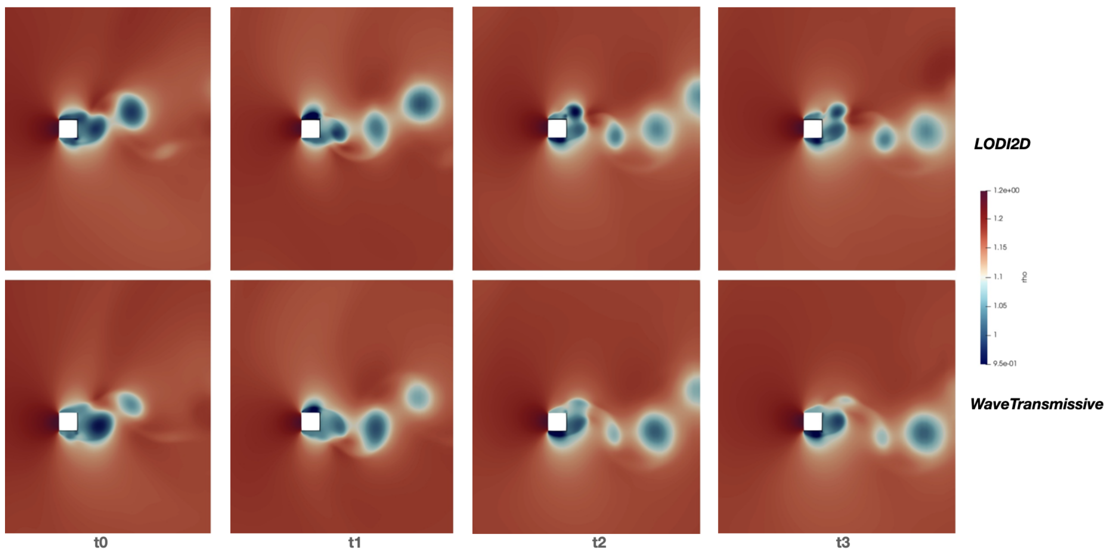


Figure 4.5: Density field of the 2-D bluff body simulation for four successive time steps for the `LODI2D` case (top) and the `waveTranmissive` case (bottom)

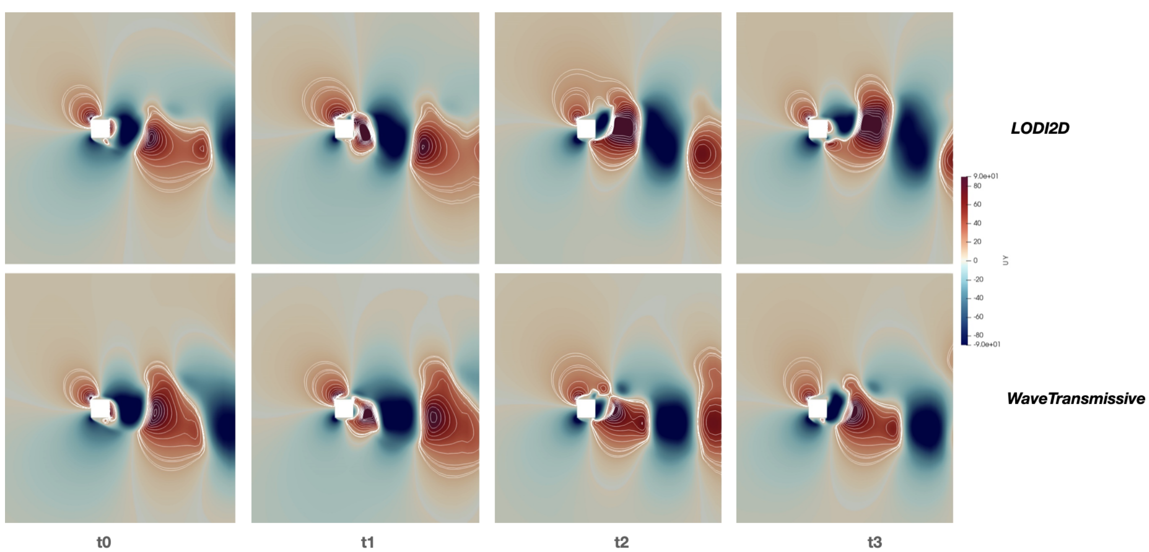


Figure 4.6: Uy field of the 2-D bluff body simulation for four successive time intervals, for the two cases. The iso contours are shown for Uy ranging from 10 to 100 m/s.

boundary conditions is sufficient for studying this particular test case.

4.3 Conclusions

In conclusion, both the simulations performed have shown very similar results in terms of reflected waves at the boundary with both boundary conditions. This shows that regardless of the difference between the implementation of OpenFOAM's native non-reflecting boundary conditions and the theoretical boundary condition, the performance of these boundary conditions is already satisfactory for most applications. It is crucial to keep in mind, however, that the way OpenFOAM's native boundary conditions have to be applied is different to the way they are applied in the tutorials, where

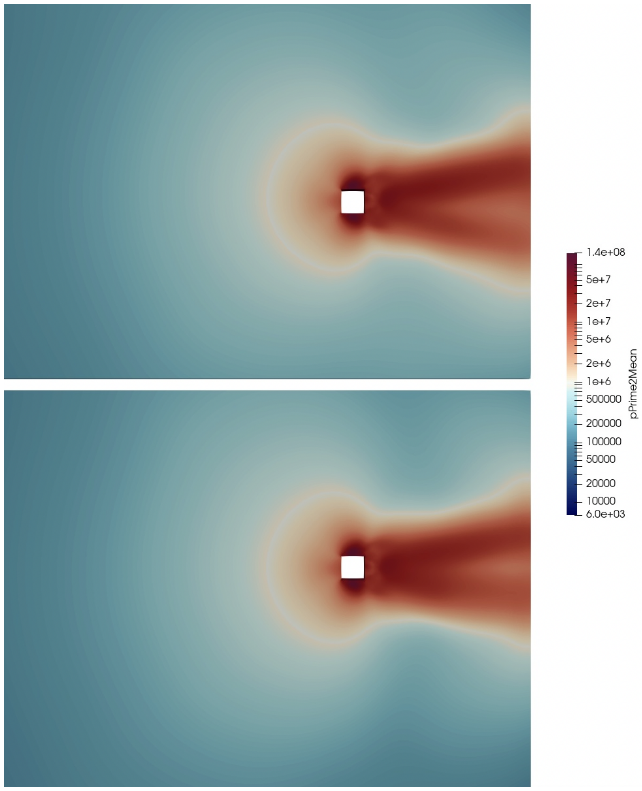


Figure 4.7: pPrime2Mean field of the 2-D bluff body simulation for one time instant, for the two cases.

the `waveTransmissive` conditions are applied only to the pressure, and the rest of the variables are handled with an `inletOutlet` approach.
Although the simulations show a similar performance of the two boundary conditions, further investigation is required in order to properly assess the capabilities of the new `LODI2D` boundary conditions and eventually an extension of their implementation to a three-dimensional case could provide even more space for studying their performance.

Bibliography

[1] A. Mani, "Analysis and optimization of numerical sponge layers as a nonreflective boundary treatment," *Journal of Computational Physics*, vol. 231, no. 2, pp. 704–716, 2012.

[2] T. J. Poinsot and S. Lelef, "Boundary conditions for direct simulations of compressible viscous flows," *Journal of computational physics*, vol. 101, no. 1, pp. 104–129, 1992.

[3] G. Lodato, P. Domingo, and L. Vervisch, "Three-dimensional boundary conditions for direct and large-eddy simulation of compressible viscous flows," *Journal of computational physics*, vol. 227, no. 10, pp. 5105–5143, 2008.

[4] M. Valorani and B. Favini, "On the numerical integration of multi-dimensional, initial boundary value problems for the euler equations in quasi-linear form," *Numerical Methods for Partial Differential Equations: An International Journal*, vol. 14, no. 6, pp. 781–814, 1998.

[5] K. W. Thompson, "Time dependent boundary conditions for hyperbolic systems," *Journal of computational physics*, vol. 68, no. 1, pp. 1–24, 1987.

[6] D. H. Rudy and J. C. Strikwerda, "A nonreflecting outflow boundary condition for subsonic navier-stokes calculations," *Journal of Computational Physics*, vol. 36, no. 1, pp. 55–70, 1980.

[7] D. H. Rudy and J. C. Strikwerda, "Boundary conditions for subsonic compressible navier-stokes calculations," *Computers & Fluids*, vol. 9, no. 3, pp. 327–338, 1981.

[8] D. Lysenko, I. S. Ertesvåg, and K. E. Rian, "Turbulent bluff body flows modeling using openfoam technology," *MekIT*, pp. 189–208, 2011.

[9] S. Pirozzoli and T. Colonius, "Generalized characteristic relaxation boundary conditions for unsteady compressible flow simulations," *Journal of Computational Physics*, vol. 248, pp. 109–126, 2013.

[10] F. R. Menter, "Two-equation eddy-viscosity turbulence models for engineering applications," *AIAA journal*, vol. 32, no. 8, pp. 1598–1605, 1994.

Study questions

1. What type of boundary conditions exist in OpenFOAM?
2. What does the `updateCoeffs` function do?
3. What equation does the `advective` boundary condition solve at the boundary face?
4. What is the difference between the `advective` and the `waveTransmissive` boundary conditions?
5. What is a templated class, and why are many boundary conditions templated?
6. What type of field does the `valueInternalCoeffs` function of the `waveTransmissive` boundary condition return when applied to the velocity field?
7. What is the difference between the newly implemented `LODI2D` and OpenFOAM's `waveTransmissive` boundary conditions?

Appendix A

Developed codes

A.1 The singleSinusoidalPressureInlet boundary condition

These are the `*.C` and `*.H` files necessary to implement the `singleSinusoidalPressureInlet` which has been utilized in the 1-D conduct simulation presented in section 2.3. The boundary condition is extremely simple and is based on the `oscillatingParabolicVelocity` boundary condition developed by professor H. Nilsson during the "CFD with OpenSource software" coruse.

`singleSinusoidalPressureInlet.H` file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2022 Hrvoje Jasak, Wikki Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "singleSinusoidalPressureInletFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "surfaceFields.H"

// * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //



// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
```

```
Foam::singleSinusoidalPressureInletFvPatchScalarField::
singleSinusoidalPressureInletFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(p, iF),
    meanValue_(0),
    amplitude_(101325),
    tstart_(0),
    f_(0)
{
}


Foam::singleSinusoidalPressureInletFvPatchScalarField:: //constructor used when the BC is set through
     the dictionary file in the time dir
singleSinusoidalPressureInletFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF),
    meanValue_(readScalar(dict.lookup("meanValue"))),
    amplitude_(readScalar(dict.lookup("amplitude"))),
    tstart_(readScalar(dict.lookup("tstart"))),
    f_(readScalar(dict.lookup("f")))
{

    fixedValueFvPatchScalarField::evaluate();
}


Foam::singleSinusoidalPressureInletFvPatchScalarField::
singleSinusoidalPressureInletFvPatchScalarField
(
    const singleSinusoidalPressureInletFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
    meanValue_(ptf.meanValue_),
    amplitude_(ptf.amplitude_),
    tstart_(ptf.tstart_),
    f_(ptf.f_)
{}


Foam::singleSinusoidalPressureInletFvPatchScalarField::
singleSinusoidalPressureInletFvPatchScalarField
(
    const singleSinusoidalPressureInletFvPatchScalarField& ptf
)
:
    fixedValueFvPatchScalarField(ptf),
    meanValue_(ptf.meanValue_),
    amplitude_(ptf.amplitude_),
    tstart_(ptf.tstart_),
    f_(ptf.f_)
{}


Foam::singleSinusoidalPressureInletFvPatchScalarField::
```

```
singleSinusoidalPressureInletFvPatchScalarField
(
    const singleSinusoidalPressureInletFvPatchScalarField& ptf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(ptf, iF),
    meanValue_(ptf.meanValue_),
    amplitude_(ptf.amplitude_),
    tstart_(ptf.tstart_),
    f_(ptf.f_)
{}


// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //



void Foam::singleSinusoidalPressureInletFvPatchScalarField::updateCoeffs()
{

    if (updated())
    {
        return;
    }

    scalar pi = constant::mathematical::pi;
    const scalar t = this->db().time().timeOutputValue();

    scalar A = amplitude_/2;

    const scalar tt = t - tstart_;
    if (tt < 0)
    {
    scalarField::operator=(meanValue_);
    }
    else if (tt > 1/f_)
    {
       scalarField::operator=(meanValue_);
    }
    else
    {
    scalarField::operator=(meanValue_ + A + A*(sin(2 * pi * f_ * tt - pi/2)));
    }
}


void Foam::singleSinusoidalPressureInletFvPatchScalarField::write
(
    Ostream& os
) const
{
    fvPatchScalarField::write(os);
    os.writeKeyword("meanValue") << meanValue_ << token::END_STATEMENT << nl;
    os.writeKeyword("amplitude") << amplitude_ << token::END_STATEMENT << nl;
    os.writeKeyword("tstart") << tstart_ << token::END_STATEMENT << nl;
    os.writeKeyword("f") << f_ << token::END_STATEMENT << nl;
    writeEntry("value", os);
}


// * * * * * * * * * * * * * Build Macro Function  * * * * * * * * * * * //

namespace Foam
{
    makePatchTypeField
    (
        fvPatchScalarField,
```

```
        singleSinusoidalPressureInletFvPatchScalarField
    );
}

// ************************************************************************* //
```

singleSinusoidalPressureInlet.H file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2022 Hrvoje Jasak, Wikki Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::singleSinusoidalPressureInletFvPatchVectorField

Group
    grpGenericBoundaryConditions

Description
    Boundary condition specifies a parabolic velocity inlet profile
    (fixed value) that oscillates in time, given maximum velocity value
    (peak of the parabola), flow direction n and direction of the parabolic
    coordinate y and frequency of the time oscillation

Usage
    \table
        Property   | Description                 | Req'd | Default
        scalarData | single scalar value         | yes |
        data       | single vector value         | yes |
        fieldData  | vector field across patch   | yes |
        timeVsData | vector function of time     | yes |
        wordData   | word, eg name of data object | no  | wordDefault
    \endtable

    Example of the boundary condition specification:
    {
        type       singleSinusoidalPressureInlet;
        scalarData -1;
        data       (1 0 0);
        fieldData  uniform (3 0 0);
        timeVsData table (
                             (0 (0 0 0))
                             (1 (2 0 0))
                         );
        wordName   anotherName;
        value      uniform (4 0 0); // optional initial value
    }
```

```
    \endverbatim

SourceFiles
    singleSinusoidalPressureInletFvPatchVectorField.C

\*---------------------------------------------------------------------------*/

#ifndef singleSinusoidalPressureInletFvPatchScalarField_H
#define singleSinusoidalPressureInletFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"
#include "Function1.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

/*---------------------------------------------------------------------------*\
     Class singleSinusoidalPressureInletFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class singleSinusoidalPressureInletFvPatchScalarField
:
    public fixedValueFvPatchScalarField
{
    // Private Data

    //- pressure mean value
    scalar meanValue_;

    //- Amplitude of oscillation
    scalar amplitude_;

    //- Start of the oscillation
    scalar tstart_;

    //- Frequency of the oscillation in time
    scalar f_;

    // Private Member Functions

public:

    //- Runtime type information
    TypeName("singleSinusoidalPressureInlet");


    // Constructors

        //- Construct from patch and internal field
        singleSinusoidalPressureInletFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        singleSinusoidalPressureInletFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping onto a new patch
        singleSinusoidalPressureInletFvPatchScalarField
        (
```

```
            const singleSinusoidalPressureInletFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Copy construct
        singleSinusoidalPressureInletFvPatchScalarField
        (
            const singleSinusoidalPressureInletFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new singleSinusoidalPressureInletFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        singleSinusoidalPressureInletFvPatchScalarField
        (
            const singleSinusoidalPressureInletFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new singleSinusoidalPressureInletFvPatchScalarField
                (
                    *this,
                    iF
                )
            );
        }


    // Member Functions

        // functions that can be used to return the values of the member data and to change them
        //

    //- Return max value
    scalar meanValue()
    {
        return meanValue_;
    }

    //- Return amplitude
    scalar& amplitude()
    {
        return amplitude_;
    }

    //- Return flow direction
    scalar& tstart()
    {
        return tstart_;
    }
```

```
    //- Return frequency
    scalar& f()
    {
        return f_;
    }

        // Evaluation functions

            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        //- Write
        virtual void write(Ostream& os) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //


#endif

// ************************************************************************* //
```

Index