Cite as: Jarfors, B.: Description and modification of the waveTransmissive boundary condition; A partially reflecting 3D wave transmissive boundary condition. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Description and modification of the waveTransmissive boundary condition; A partially reflecting 3D wave transmissive boundary condition

Developed for OpenFOAM-v2112

Author:
Björn Jarfors
Lund University
bjorn.jarfors@energy.lth.se

Peer reviewed by:
Khoder Alhamwi Alshaar
Saeed Salehi
Christer Fureby

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like to learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 24, 2025

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:

- A general description will be given on the use of the `mixed`, `advective` and `waveTransmissive` boundary conditions.

The theory of it:

- A brief description of channel acoustics and reflections at boundaries will be given.
- An overview of Local One-Dimensional Inviscid (LODI) relations will be given together with a new derivation of the boundary equations accounting for reflections.

How it is implemented:

- A description of `mixed`, `advective` and `waveTransmissive` boundary condition's implementation and how their implementations refer to the characteristic equations derived through LODI relations.
- A brief description of the changes implemented by Leandro Lucchese in his `LODI2D` and `mixedV2D` codes.
- A comparison between these implementations and Leandro Lucchese's `LODI2D` boundary condition will be given.
- How these can be extended to 3D.
- How the partially reflecting conditions can be implemented.

How to modify it:

- The equations will be modified to accommodate for the new partially reflecting derivation from the LODI-relations.
- The boundary condition will be extended to 3D.
- The optimal settings for the custom boundary condition is revised and compared to both OpenFOAM's `waveTransmissive` and Leandro Lucchese's `LODI2D` boundary conditions.
- The standard `waveTransmissive`, Leandro Lucchese's `LODI2D`, [1], and the new 3D boundary condition's are tested on a simple 3D non-reacting bluff-body case to check the transmission of inherently 3D structures.

Prerequisites

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

- In-depth knowledge regarding numerical methods for solving fluid dynamical problems.
- A basic understanding of usage and top-level programming in OpenFOAM.
- Basic understanding of C++ programming.
- Preferably the reader has read Lucchese, L. report [1] as this can be seen as a continuation of his work.

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics	1
DNS	Direct Numerical Simulation	1
FFT	Fast Fourier Transform	1
LES	Large Eddy Simulation	1
NSCBC	Navier-Stokes Characteristic Boundary Conditions	1
TDE	Time-Delay Embedding	1

English symbols

$\mathscr{L}_i$	Characteristic waves	1
$\mathbf{A}$	Coefficient matrix to the linear system	1
$\mathbf{S}$	Source term to the linear system	1
$a_{\nabla\cdot}$	Contribution to $\mathbf{A}$ in the divergence term	1
a_{∇^2}	Contribution to $\mathbf{A}$ in the laplacian term	1
c	Speed of Sound	m/s
C_p	Specific heat capacity at constant pressure	$\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$
C_v	Specific heat capacity at constant volume	$\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$
E	Total energy	J
f	Frequency	Hz
f	Value fraction	1
K	Relaxation coefficient	1
L, l	Length	m
M	Mach number	1
p	Pressure	$\mathrm{N/m^2}$
q	Heat flux	$\mathrm{J}/(\mathrm{m^2 s})$
R	Reflection coefficient	1
S_f	Surface normal vector	1
$s_{\nabla\cdot}$	Contribution to $\mathbf{S}$ in the divergence term	1
s_{∇^2}	Contribution to $\mathbf{S}$ in the laplacian term	1
T	Temperature	K
T	Transmission coefficient	1
t	Time	s
u	Velocity	m/s
w	Wave speed	m/s
x_i	Cartesian coordinate	m

Greek symbols

α	Coefficient	1
Δ	A small difference	1
δ	Kronecker delta	1
γ	Ratio of specific heats	1

λ	Eigenvalue	1
ϕ	Transported variable	1
ψ	Compressibility	s^2/m^2
ρ	Fluid density	kg/m^3
τ	Stress tensor	N/m^2

Superscripts

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

Subscripts

∞	far-field value	1
+	positive direction propagation	1
-	negative direction propagation	1
c	cell center value	1
f	face center value	1
in	inlet	1
L	longitudinal wave	1
n	normal	1
out	outlet	1
ref	reference value	1
t	transverse	1

Introduction

When performing high-fidelity simulations such as Large Eddy Simulations (LES) or Direct Numerical Simulations (DNS) [2], precise boundary conditions become all the more important. This is due to the decreased numerical diffusion allowing numerical waves to exist without significant damping. Once these numerical waves interact with e.g. a boundary condition, these may turn into physical waves, altering the final convergence of the simulations solution, which may lead to triggering a limit-cycle behavior which is otherwise not present [3], [4]. One way to mitigate this issue is to simulate a larger domain, until there exists a well-defined boundary. However, regarding acoustic wave propagation, such an approach require sufficiently large domains to be simulated until such a boundary is found. To reduce the computational cost, it is favorable to introduce a more advanced, but still well-defined boundary condition. Such boundary conditions are generally derived from the so-called Navier-Stokes Characteristic Boundary Conditions (NSCBC), which, together with Local One-Dimensional Inviscid (LODI) relations, form a powerful tool [5]. Previously, a wave-transmissive boundary condition was derived by Rudy and Strikwerda [6], and improved by Poinsot and Lele [3]. However, the original definition is based on Perfectly Non-Reflecting waves, which resulted in issues of pressure-drifting. Therefore, even though neglected in the derivations, information from outside of the domain was included in the equations through a term, $\mathscr{L}_1 = K(p - p_\infty)$, to preserve the information regarding the atmospheric pressure outside of the domain [6]. As described by Rudy and Strikwerda [6], this term is meant to result in a perfectly non-reflecting boundary which maintains the mean pressure in the system, and the determination of K is defined as such to converge to this state in as few iterations as possible. Since this is a dynamic boundary condition which will actively change depending on p, it is essentially modulating the reflected waves to maintain the mean pressure. As will be described in the chapter **Theoretical Background**, there exists systems for which reflections naturally occur at the boundary without the intent to maintain a mean pressure.

In channel acoustics, a phenomenon where traveling waves reflect at the boundaries may result in constructive interference with a specific frequency. This is called resonance, and strong standing waves are present in these cases. These are very important in capturing to determine the stability of a wide amount of fluid-dynamical problems. This is due to the nature of such an instability, which is classified as an absolute instability. It grows everywhere and does not dissipate spatially like a convective instability. Therefore, an alternative formulation which is derived based on the assumption of Partially Reflecting waves is reasonable. It should be noted that this is nothing new, but the author was not able to find any active implementation in OpenFOAM regarding this. Furthermore, to make the boundary condition more realistic, a continuation of Leandro Lucchese's work [1] on extending the wave-transmissive boundary condition from 1D to 2D is continued to 3D. The 3D formulation is preferable due to the inherent 3D nature of vorticle structures. As these interact with the boundary, it is optimal to treat all three velocity components to make sure substantial numerical waves are prevented. Finally, the 1D, 2D and 3D formulations are compared with the Perfectly Non-Reflecting assumption, followed by a comparison between the 3D case for Perfectly Non-Reflecting and Partially Reflecting assumptions to reveal their differences.

Chapter 1

Theoretical Background

1.1 Channel Acoustics

The propagation of a traveling wave may be greatly affected by the boundaries it interacts with. An example of the interactions can be seen in e.g. channel acoustics. Imagine a rectangular channel with an open inlet- and outlet boundary such as in Fig. 1.1. At the outlet, there is a discontinuity jump in the cross-sectional area. Based on the characteristics of the discontinuity, the interaction between the traveling wave and the boundary will change. Assume there are two waves, one traveling in the positive x-direction, p_+, and one in the negative x-direction, p_-. The latter is an effect of the traveling wave interacting with the boundary, that is, a reflected wave. This is represented in the zoomed in region in Fig. 1.1. The pressure inside the domain will thus be $p(x,t) = p_+(x,t)+p_-(x,t)$. This highlights the importance of accurately capturing the proportion of the traveling wave which is reflected. Due to these reflections, the system may enter a resonance mode and exhibit standing waves [7].

Standing waves are not physical waves, but the interference of two or more traveling waves. To characterize the properties of a standing wave, one needs to first determine the speed of sound in the medium which is

$$c = \sqrt{\gamma/\psi},$$

for perfect gases, where $\gamma = C_p/C_v$ is the ratio of specific heats where C_p and C_v are the specific heats at constant pressure and volume respectively, and $\psi = \rho/p$ is the compressibility, where ρ and p are the density and pressure, respectively. To calculate approximately at what frequencies resonance will occur, e.g. for the first mode of the previously defined open-inlet-open-outlet domain,

$$f_{2L} = c/(2L),$$

where L is the characteristic length of the domain. This is known as a half-wave [7].

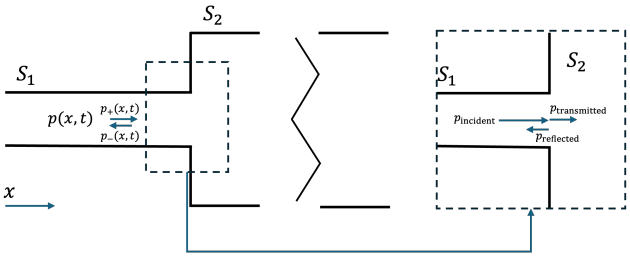

Figure 1.1: Illustration of down- and upstream propagating waves in a confined chamber with a zoom at the boundary showing the origin of the upstream propagating wave.

1.2 Navier-Stokes Characteristic Boundary Conditions

Defining boundary conditions which are realistic is not always trivial. Poinsot and Lele [3] emphasizes that, for well-posedness, it is necessary to base the boundary conditions on a version of the Navier-Stokes equations, such as the Euler equations where viscous effects are neglected. They start from the compressible Navier-Stokes equations

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

$$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_i u_j)}{\partial x_j} + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j}, \tag{1.2}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial(\rho E + p)u_i}{\partial x_i} = \frac{\partial(u_i \tau_{ij})}{\partial x_i} - \frac{\partial q_i}{\partial x_i}, \quad (i = 1 \text{ to } 3), \tag{1.3}$$

where

$$\rho E = \frac{1}{2}\rho u_k u_k + \frac{p}{\gamma - 1}, \text{ and} \tag{1.4}$$

$$\tau_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_k}{\partial x_k} \right), \tag{1.5}$$

where τ_{ij} is the viscous shear stress tensor with μ which is the dynamic viscosity. Using these set of equations together with the perfect gas law $p = \rho RT$, one can derive a set of equations called the Local One-Dimensional Inviscid (LODI) relations.

1.2.1 LODI Relations

The LODI relations can be derived from the Navier-Stokes equations by using characteristic analysis to reformulate the hyperbolic terms and reformulating the system as done by Thompson [8]. From the characteristic analysis, equations connecting the characteristic waves and the reformulated Navier-Stokes equations are obtained. Lastly, these characteristic waves, $\mathscr{L}_i$, are accompanied with characteristic velocities, λ_i [8]. The LODI relations presented by Poinsot and Lele [3] are

$$\mathscr{L}_1 = \lambda_1 \left(\frac{\partial p}{\partial x_1} - \rho c \frac{\partial u_1}{\partial x_1} \right), \tag{1.6}$$

$$\mathscr{L}_2 = \lambda_2 \left(c^2 \frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1} \right), \tag{1.7}$$

$$\mathscr{L}_3 = \lambda_3 \frac{\partial u_2}{\partial x_1}, \tag{1.8}$$

$$\mathscr{L}_4 = \lambda_4 \frac{\partial u_3}{\partial x_1}, \tag{1.9}$$

$$\mathscr{L}_5 = \lambda_5 \left(\frac{\partial p}{\partial x_1} + \rho c \frac{\partial u_1}{\partial x_1} \right), \tag{1.10}$$

where $\mathscr{L}_i$ is the information the characteristic waves carry, and λ_i, $i = 1 \text{ to } 5$ are the eigenvalues corresponding to the characteristic speeds of information traveling in the domain. An illustration of these waves within a computational domain is presented in Fig. 1.2.
Considering the 1D NSE, a formulation with the characteristic waves may be derived as

$$\frac{\partial \rho}{\partial t} + \frac{1}{c^2}\left(\mathscr{L}_2 + \frac{1}{2}(\mathscr{L}_5 + \mathscr{L}_1) \right) = 0, \tag{1.11}$$

$$\frac{\partial p}{\partial t} + \frac{1}{2}(\mathscr{L}_5 + \mathscr{L}_1) = 0, \tag{1.12}$$

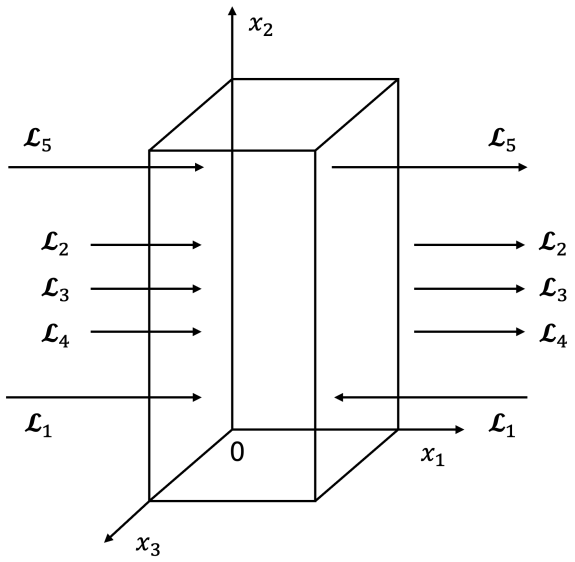

Figure 1.2: Characteristic waves with their respective directions adjusted for subsonic flows.

$$\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}(\mathscr{L}_5 - \mathscr{L}_1) = 0, \tag{1.13}$$

$$\frac{\partial u_2}{\partial t} + \mathscr{L}_3 = 0, \tag{1.14}$$

$$\frac{\partial u_3}{\partial t} + \mathscr{L}_4 = 0. \tag{1.15}$$

Furthermore, these equations may be written in terms of gradients normal to the boundary,

$$\frac{\partial \rho}{\partial x_1} = \frac{1}{c^2}\left(\frac{\mathscr{L}_2}{u_n} + \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_n + c} + \frac{\mathscr{L}_1}{u_n - c}\right)\right), \tag{1.16}$$

$$\frac{\partial p}{\partial x_1} = \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_n + c} + \frac{\mathscr{L}_1}{u_n - c}\right), \tag{1.17}$$

$$\frac{\partial u_1}{\partial x_1} = \frac{1}{2\rho c}\left(\frac{\mathscr{L}_5}{u_n + c} - \frac{\mathscr{L}_1}{u_n - c}\right), \tag{1.18}$$

$$\frac{\partial T}{\partial x_1} = \frac{T}{\rho c^2}\left(-\frac{\mathscr{L}_2}{u_n} + \frac{1}{2}(\gamma - 1)\left(\frac{\mathscr{L}_5}{u_n + c} + \frac{\mathscr{L}_1}{u_n - c}\right)\right). \tag{1.19}$$

By remembering that $\mathscr{L}_i$ for $i = 2$ to 5 correspond to information leaving the domain (for subsonic outlets), and $\mathscr{L}_1$ is the information entering the domain, recall Fig. 1.2, one can realize that for perfectly non-reflecting boundary conditions, $\mathscr{L}_1 = 0$. Taking this into account in Eq. (1.12) together with Eq. (1.17) results in

$$\frac{\partial p}{\partial t} + (u_n + c)\frac{\partial p}{\partial x_1} = 0, \text{ and} \tag{1.20}$$

$$\frac{\partial u_1}{\partial t} + (u_n + c)\frac{\partial u_1}{\partial x_1} = 0 \tag{1.21}$$

A modified version of these equations is used in OpenFOAM in their `waveTransmissive` boundary condition. The modification is based on what is discussed in Poinsot and Lele's work [3]. The justifications for these modifications are due to the lack of information regarding the atmospheric pressure outside of the domain, and the inlets are generally mass flow or velocity driven inlets, the pressure drifts towards zero. In light of this, the solved equations have an extra term,

$$\mathscr{L}_1 = K(p - p_\infty), \tag{1.22}$$

which corresponds to ingoing information regarding the atmospheric conditions outside of the domain. Here, the coefficient $K = (u_1 + c)/l_\infty$ based on OpenFOAM's implementation, where p is the pressure on the boundary-face and p_∞ is the condition set at a distance l_∞ from the boundary[1]. Putting K's definition aside for now, and focusing on the rest of the derivations we get

$$\frac{\partial p}{\partial t} + (u_n + c)\frac{\partial p}{\partial x_1} + K(p - p_\infty) = 0. \tag{1.23}$$

A simple discretization using implicit Euler time and upwind spatial discretization was presented by Lucchese [1]

$$\phi_f^{n+1} = (\phi_f^n + k\phi_\infty)\frac{1}{1 + \alpha + k} + \frac{\alpha}{1 + \alpha + k}\phi_c^{n+1}, \tag{1.24}$$

where $\alpha = U_n \Delta t/\Delta$, and $k = K\Delta t$.

1.2.2 Partially Reflecting Boundary Condition

To obtain the partially reflecting boundary condition equations, the derivation needs to include $\mathscr{L}_1$ from the start. This will be illustrated using the velocity, but both pressure and velocity equations will be presented. Starting from Eq. (1.10) and (1.13) one can derive

$$\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}\left[(u_1 + c)\left(\frac{\partial p}{\partial x_1} + \rho c\frac{\partial u_1}{\partial x_1}\right) - \mathscr{L}_1\right] = 0, \tag{1.25}$$

where we may insert a rearranged version of Eq. (1.6) as well as Eq. (1.22) to retrieve

$$\frac{\partial u_1}{\partial t} + (u_1 + c)\frac{\partial u_1}{\partial x_1} + \frac{1}{2\rho c}\left[\left(\frac{u_1 + c}{u_1 - c}\right) - 1\right]K(p - p_\infty) = 0. \tag{1.26}$$

As seen in Eq. (1.26), the pressure is needed. Here we assume linear acoustics such that we can express $u' = p'/(\rho c)$ which results in

$$\frac{\partial u_1}{\partial t} + (u_1 + c)\frac{\partial u_1}{\partial x_1} + \frac{1}{2}\left[\left(\frac{u_1 + c}{u_1 - c}\right) - 1\right]K(u - u_\infty) = 0. \tag{1.27}$$

Comparing this to the perfectly non-reflecting formulation

$$\frac{\partial u_1}{\partial t} + (u_1 + c)\frac{\partial u_1}{\partial x_1} + K(u - u_\infty) = 0, \tag{1.28}$$

we can see that there is an additional term. This can be reformulated using the Mach number, $Ma = u_1/c$, to obtain

$$\frac{1}{2}\left(\frac{Ma + 1}{Ma - 1} - 1\right),$$

which can be understood as a transmission or reflection coefficient which varies for different types of **subsonic flows**, based on acoustics in moving media.

When discretized using implicit Euler for time and upwind for space, the current time-step for velocity becomes

$$\phi_f^{n+1} = \frac{1}{1 + \alpha_{uc} + Rk}\phi_f^n + \frac{\alpha_{uc}}{1 + \alpha_{uc} + Rk}\phi_c^{n+1} + \frac{Rk}{1 + \alpha_{uc} + Rk}\phi_\infty, \tag{1.29}$$

[1]The coefficient K is used to relax the incoming waves and originally, it was proposed that its definition should be $K = \sigma(1 - Ma^2)c/L$ by Rudy and Strikwerda [6]. Here, σ is a constant, usually set to 0.25, but will be the constant which one may tune to get the desired behavior (even for reflections), M is the maximum Mach number in the flow, and L is the characteristic size of the domain [3].

where $\alpha_{uc} = (u_1 + c)\Delta t/\Delta x$, $k = K\Delta t$ and $R = (u_1 + c)/(u_1 - c) - 1$. Similarly, for pressure

$$\phi_f^{n+1} = \frac{1}{1 + \alpha_{uc} + Tk}\phi_f^n + \frac{\alpha_{uc}}{1 + \alpha_{uc} + Tk}\phi_c^{n+1} + \frac{Tk}{1 + \alpha_{uc} + Tk}\phi_\infty, \tag{1.30}$$

where $T = -R$ due to the phase difference between velocity and pressure fluctuations.

The difference in the transport equation in comparison to Eq. (1.24) is small. The extra scaling term indicates an impact of the amount reflected/transmitted depending on what Mach number flows are exiting the boundary which is reasonable. However, the assumption regarding the treatment $\mathscr{L}_1$ is the same as previous implementations, which may be unreasonable since this formulation is not only supposed to maintain the mean pressure, but introduce physical reflections as well. Thus it is proposed to have a formulation as $\mathscr{L}_1 = a + K(p - p_\infty)$, which is similar to the formulation proposed by Yoo *et al.*[2] [9]. By choosing this, the previous derivation can be split into

$$\frac{\partial p}{\partial t} + \frac{1}{2\rho c}\left(\lambda_5\left(\frac{\partial p}{\partial x} + \rho c\frac{\partial u}{\partial x}\right) - \lambda_1\left(\frac{\partial p}{\partial x} - \rho c\frac{\partial u}{\partial x}\right)_{\text{inside}} - \lambda_1\left(\frac{\partial p}{\partial x} - \rho c\frac{\partial u}{\partial x}\right)_{\text{outside}}\right) \tag{1.31}$$

and modeling the outside term by $K(p - p_\infty)$. However, the author is unknown of the impacts of this assumption at this moment. The motivation to this splitting of the $\mathscr{L}_1$ term is based on a 2^{nd} order central differencing approach, where it is between the cell center, face center and a point outside of the domain (at distance l_∞). This is left for future work but kept here to keep the reader critical to any assumptions used in such derivations as these.

[2] Yoo *et al.* [9] proposed to calculate $L_1 = \beta_1(u - u_\infty) + \mathscr{T}_1$, where $\mathscr{T}_1$ is the transverse contribution to the incoming information.

Chapter 2

Boundary Conditions in OpenFOAM

2.1 General

In general, boundary conditions are divided into two subsections, Dirichlet and Neumann boundary conditions. These refer to zeroth-order term (fixed value) and the first order term (fixed gradient) being fixed, respectively. In OpenFOAM there is an option to use a so-called `mixed` boundary condition[1]. In this case, a mixture of zeroth- and first-order terms are used. Boundary conditions affect the system of equations differently as they can either affect the divergence (denoted with subscript $\nabla\cdot$) or laplacian term (denoted with subscript ∇^2) in the equations, as well as impacting the solution explicitly, or implicitly. An explicit implementation refers to an alteration of the source terms, whilst an implicit one affects the coefficient matrix ($\mathbf{A}$) of the linear system. The member functions to look out for regarding this include `valueInternalCoeffs`, `valueBoundaryCoeffs`, `gradientInternalCoeffs`, and `gradientBoundaryCoeffs`. Their importance include:

- `valueInternalCoeffs` implicitly affects the equations through the diagonal elements in the coefficient matrix for the divergence term, $a_{\nabla\cdot}$.
- `valueBoundaryCoeffs` explicitly affects the equations through the source term in the divergence term, $s_{\nabla\cdot}$.
- `gradientInternalCoeffs` implicitly affects the equations through the diagonal elements in the coefficient matrix for the laplacian term, a_{∇^2}.
- `gradientBoundaryCoeffs` explicitly affects the equations through the source term in the laplacian term, s_{∇^2}.

These terms are introduced in Eq.'s (2.1), (2.2), (2.3), and (2.4).

2.1.1 `mixed`

The `mixed` boundary condition folder includes templated functions which are used for all mixed boundary conditions. The main member functions of interest in this "base" class includes: `valueInternalCoeffs`, `valueBoundaryCoeffs`, `gradientInternalCoeffs`, `gradientBoundaryCoeffs`, `evaluate`, and `snGrad`. A list is presented in order to assign an alias to these variables and their dependent variables and parameters for the sake of brevity,

- `valueFraction`, f,

[1] This is a generic boundary condition which will be explained more in the `mixed` section.

- `refValue`, ϕ_{ref},
- `refGrad`, $\nabla\phi_{\text{ref}}$,
- `this->patch().deltaCoeffs`, $\mathbf{d}$,
- `evaluate`, e, and
- `snGrad`, $\nabla^{\perp}_{f}(\phi)$, where ϕ is a general field variable.

These member functions are defined mathematically as

$$a_{\nabla\cdot} = 1 - f, \tag{2.1}$$

$$s_{\nabla\cdot} = f\phi_{\text{ref}} + (1-f)\nabla\phi_{ref}\mathbf{d}, \tag{2.2}$$

$$a_{\nabla^2} = -f\mathbf{d}, \tag{2.3}$$

$$s_{\nabla^2} = f\mathbf{d}\phi_{\text{ref}} + (1.0-f)\nabla\phi_{\text{ref}}, \tag{2.4}$$

$$e = f\phi_{\text{ref}} + (1.0-f)(\phi_c^{n+1} + \nabla\phi_{\text{ref}}/\mathbf{d}), \tag{2.5}$$

$$\nabla^{\perp}_{f}\phi = f(\phi_{ref} - \phi_c^{n+1}) * \mathbf{d} + (1.0-f)\nabla\phi_{\text{ref}}. \tag{2.6}$$

From these equations we can see three undefined terms, namely, f, ϕ_{ref}, and $\nabla\phi_{ref}$. The aim of the classes `advective` and `waveTransmissive` will be to help define these undefined terms, which will then be used as input variables to the `mixed` class.

2.1.2 `advective`

An example of a mixed boundary condition is the `advective` boundary condition. The former undefined terms are determined in the member function `updateCoeffs()` seen in Listing 2.1 where for Euler or Crank Nicolson time discretization, f is defined on line 31.

Listing 2.1: Definition of `updateCoeffs` in `advective`

```
1  template<class Type>
2  void Foam::advectiveFvPatchField<Type>::updateCoeffs()
3  .
4  .
5  .
6      // Calculate the advection speed of the field wave
7      // If the wave is incoming set the speed to 0.
8      const scalarField w(Foam::max(advectionSpeed(), scalar(0)));
9
10     // Calculate the field wave coefficient alpha (See notes)
11     const scalarField alpha(w*deltaT*this->patch().deltaCoeffs());
12 .
13 .
14 .
15    if (lInf_ > 0)
16     {
17         // Calculate the field relaxation coefficient k (See notes)
```

```
18          const scalarField k(w*deltaT/lInf_);
19
20          if
21          (
22              ddtScheme == fv::EulerDdtScheme<scalar>::typeName
23           || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
24          )
25          {
26              this->refValue() =
27              (
28                  field.oldTime().boundaryField()[patchi] + k*fieldInf_
29              )/(1.0 + k);
30
31              this->valueFraction() = (1.0 + k)/(1.0 + alpha + k);
32          }
33  .
34  .
35  .
```

This can be written as

$$f = (1.0 + k)/(1.0 + \alpha + k). \tag{2.7}$$

For ϕ_{ref} it is seen in Listing 2.1 on line 26 to 29 which may be recast into

$$\phi_{ref} = (\phi_f^n + k\phi_\infty)/(1.0 + k). \tag{2.8}$$

The term, $\nabla\phi_{\text{ref}}$, remains unchanged after initialization which occurs in Listing 2.2 on line 15.

Listing 2.2: Initialization of `refGrad` in `advective`

```
1  template<class Type>
2  Foam::advectiveFvPatchField<Type>::advectiveFvPatchField
3  (
4      const fvPatch& p,
5      const DimensionedField<Type, volMesh>& iF
6  )
7  :
8      mixedFvPatchField<Type>(p, iF),
9      phiName_("phi"),
10     rhoName_("rho"),
11     fieldInf_(Zero),
12     lInf_(-GREAT)
13 {
14     this->refValue() = Zero;
15     this->refGrad() = Zero;
16     this->valueFraction() = 0.0;
17 }
```

Thus the gradient of the reference value of the generic variable remains

$$\nabla\phi_{\text{ref}} = 0, \text{ and} \tag{2.9}$$

not altered anywhere in the code. Furthermore, there is an important member function called `advectionSpeed()`, w_u, which calculates the velocity normal to the boundary patch, either by considering the mass flux, or simply the velocity face flux. This can be seen in Listing 2.3 on line 23 considering mass flux and line 27 considering the velocity face flux.

Listing 2.3: Definition of `advectionSpeed()` in `advective`

```
1  template<class Type>
2  Foam::tmp<Foam::scalarField>
3  Foam::advectiveFvPatchField<Type>::advectionSpeed() const
4  {
5      const surfaceScalarField& phi =
6          this->db().objectRegistry::template lookupObject<surfaceScalarField>
```

```
7           (phiName_);
8
9       fvsPatchField<scalar> phip =
10          this->patch().template lookupPatchField<surfaceScalarField, scalar>
11          (
12              phiName_
13          );
14
15      if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
16      {
17          const fvPatchScalarField& rhop =
18              this->patch().template lookupPatchField<volScalarField, scalar>
19              (
20                  rhoName_
21              );
22
23          return phip/(rhop*this->patch().magSf());
24      }
25      else
26      {
27          return phip/this->patch().magSf();
28      }
29  }
```

The latter is defined as

$$w_u = U \cdot S_f, \tag{2.10}$$

where U is the velocity vector, and S_f is the surface normal vector, effectively projecting the velocity onto the surface normal vector. The parameters α and k are introduced as a function of w, corresponding to the wave speed. The wave speed is determined as a scalar field by

$$w_{\max} = \max(w, 0), \tag{2.11}$$

whilst

$$\alpha = (w_{\max}\Delta t)/\mathbf{d}, \text{ and} \tag{2.12}$$

$$k = (w_{\max}\Delta t)/l_\infty, \tag{2.13}$$

which can be seen in Listing 2.1 on lines 11 and 18, respectively.

2.1.3 `waveTransmissive`

An extension to the definition of the `advectionSpeed()` member function is provided by the `waveTransmissive` boundary condition which can be seen in Listing 2.4 on line 29.

Listing 2.4: Definition of `advectionSpeed()` in `waveTransmissive`

```
1   template<class Type>
2   Foam::tmp<Foam::scalarField>
3   Foam::waveTransmissiveFvPatchField<Type>::advectionSpeed() const
4   {
5       // Lookup the velocity and compressibility of the patch
6       const fvPatchField<scalar>& psip =
7           this->patch().template
8               lookupPatchField<volScalarField, scalar>(psiName_);
9
10      const surfaceScalarField& phi =
11          this->db().template lookupObject<surfaceScalarField>(this->phiName_);
12
13      fvsPatchField<scalar> phip =
14          this->patch().template
15              lookupPatchField<surfaceScalarField, scalar>(this->phiName_);
```

```
16
17      if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
18      {
19          const fvPatchScalarField& rhop =
20              this->patch().template
21                  lookupPatchField<volScalarField, scalar>(this->rhoName_);
22
23          phip /= rhop;
24      }
25
26      // Calculate the speed of the field wave w
27      // by summing the component of the velocity normal to the boundary
28      // and the speed of sound (sqrt(gamma_/psi)).
29      return phip/this->patch().magSf() + sqrt(gamma_/psip);
30  }
```

Simply put, it alters the `advectionSpeed()` to that of the wave speed

$$w_{uc} = U \cdot S_f + \sqrt{\gamma/\psi}. \tag{2.14}$$

This enters Eq. 2.11 to obtain the maximum value, which results in new definitions for α and k which will be noted with the subscript "uc", whilst the `advective` version corresponds to the subscript "u".

2.2 LODI2D versus modifiedLODI2D

A previous student to the *CFD with Open Source Software*, Lucchese [1], developed a code called `LODI2D` with its generic boundary condition `mixedV2D`. This boundary condition is based on the combination of `mixed`, `advective`, and `waveTransmissive` boundary condition. He implemented the `advective` boundary condition directly in the `LODI2D` together with the `waveTransmissive` parts. In order to extend the formulation from 1D to 2D, a necessary change in `mixed` was needed as well, therefore a part of his implementation is included in `mixedV2D` code. The changes were specifically related to the transportation of velocity at the boundary. A link to Leandros report and files is found in [1].

In the code, Lucchese [1] generalised the boundary condition such that the normal and transverse direction in the 2D case did not need to align with the x-, and y-coordinates. This generalization was performed by implementing a rotation after the velocity components were calculated in the normal and transverse directions. The difference in constructor can be seen as in the following two Listings 2.5 and 2.6, comparing `mixedV2DFvPatchVectorField.C` by Lucchese [1] with the simplified version `modifiedMixedV2DFvPatchVectorField.C` with the assumption that the boundary-normal vector aligns with the x-direction.

Listing 2.5: Definition of the first constructor in `mixedV2DFvPatchVectorField.C`

```
1  Foam::mixedV2DFvPatchVectorField::mixedV2DFvPatchVectorField
2  (
3      const fvPatch& p,
4      const DimensionedField<vector, volMesh>& iF
5  )
6  :
7      fvPatchVectorField(p, iF),
8      Un_(p.size()),
9      Ut_(p.size()),
10     n_(p.size()),
11     refValueU_(p.size()),
12     refValueUC_(p.size()),
13     refGrad_(p.size()),
14     valueFractionU_(p.size()),
15     valueFractionUC_(p.size()),
16     source_(p.size(), Zero),
17     vector1_(p.size()),
18     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;
        }
```

Listing 2.6: Definition of the first constructor in `modifiedMixedV2DFvPatchVectorField.C`

```
Foam::modifiedMixedV2DFvPatchVectorField::modifiedMixedV2DFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    refValueU1_(p.size()),
    refValueU2_(p.size()),
    refValueUC_(p.size()),
    refGrad_(p.size()),
    valueFractionU1_(p.size()),
    valueFractionU2_(p.size()),
    valueFractionUC_(p.size()),
    source_(p.size(), Zero)
{
}
```

The motivation to this simplification is based on the assumption which is presented in Lucchese [1], that the characteristic waves are derived based on assuming a boundary with a normal vector parallel to x_1, such that the only non-conservative Jacobian matrix which needs to be diagonalized corresponds to the one in the x_1-direction. It can be seen that with this simplification allows a direct extraction of the velocities through `vectorField U = this->patchInternalField()`, `vectorField Uold = field.oldTime().boundaryField()[patchi]`, and `vectorField Uoold = field.oldTime().oldTime().boundaryField()[patchi]`. This also implies that, when returning the contributions of the different velocity components to the coefficient matrix, $\mathbf{A}$, or source term, $\mathbf{S}$, a simple unit vector multiplication is performed as seen in Listing 2.7.

Listing 2.7: Returning the contributions to the linear equation to be solved

```
.
.
.
Foam::tmp<Foam::vectorField>
Foam::modifiedMixedV2DFvPatchVectorField::valueInternalCoeffs
(
    const tmp<scalarField>&
)   const
    {
        scalarField valueU =
            (1.0 - valueFractionU_);

        scalarField valueUC =
```

```
            (1.0 - valueFractionUC_);

        return valueUC*vector(1.0, 0.0, 0.0) + valueU*vector(0.0, 1.0, 0.0);
    }
```

2.2.1 Comparison

To verify that this is a reasonable simplification for the given test case which is schematically illustrated in Fig. 2.1 which consists of an open domain with a square cylinder bluff-body which sheds vortices. The probe location is chosen such that the periodicity of the vortex-shedding is analyzed between the cases and its time-series of the transverse (y) velocity behind a bluff body is tracked, compared and shown in Fig. 2.2. These are shown to completely overlap and thus the simplification is valid for this specific test case. An example when this simplification no longer is valid would be when the boundary normal vector is a linear combination of your base coordinate system, (x, y, z). If the boundary-normal vector is perpendicular to x, then it is recommended to switch the coordinate system such that it aligns with the x-direction.

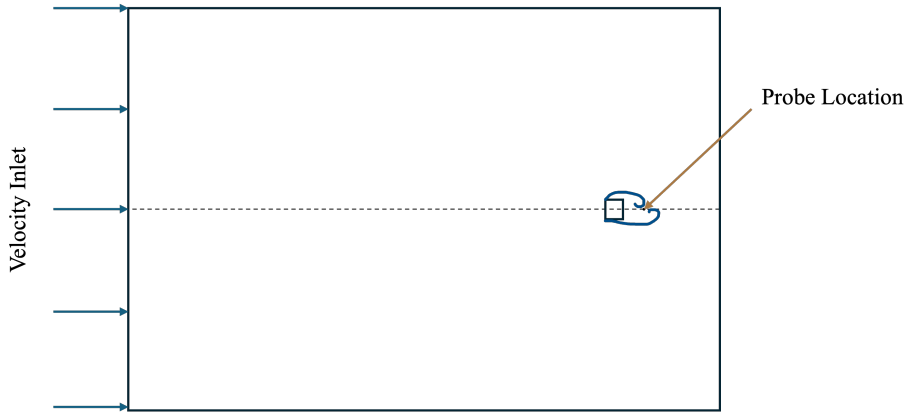

Figure 2.1: Schematic illustration of the computational domain and probe location.

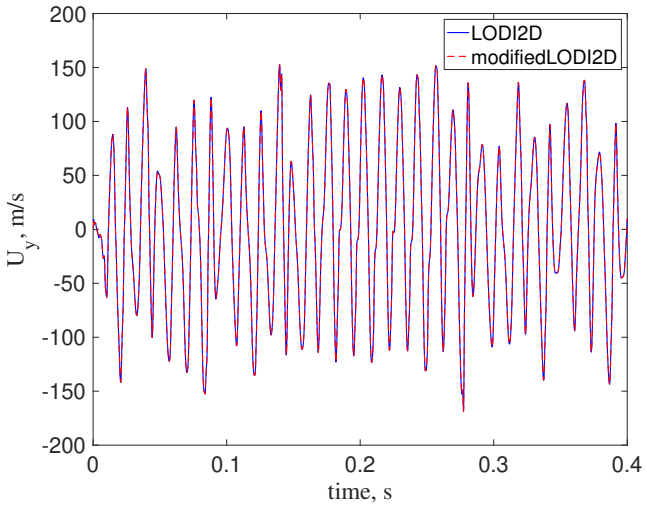

Figure 2.2: A comparison between the time-series of the y-component of velocity behind the square cylinder bluff body, between `LODI2D` and `modifiedLODI2D`.

Chapter 3

Implementation of the New Boundary Condition

DISCLAIMER: Note that parts the code is identical to Leandro Lucchese's version [1], which inherits its structure from `waveTransmissive`, `advective` and `mixed` classes. Strictly, the reduction in complexity, introduction of the third components and the inclusion of the additional reflection coefficient were extended from the author's work.

3.1 Initial modifications

Since the complexity of rotations increases with an extension from 2D to 3D, it is assumed that the boundary normal vector is always **parallel** with the x**-direction**. This is also a part of the derivations of the LODI-relations making it a reasonable assumption. This greatly simplifies the code as there is no need to construct member data such as the normal and transverse velocities amongst other member data, since they are directly accessible through the `patchInternalField()` member function from the `fvPatch` class.

3.2 `modifiedMixedV3D`

The implementation of the third transported velocity component is included as a simple extension by inserting an extra component for the member functions `refValue`, and `valueFraction`.

The transverse components of the velocity, corresponding to the y- and z-components, are transported by the characteristic waves defined in Eqs. (1.8) and (1.9). Therefore, these are transported with the convective speed w_u. This results in an additional ϕ, thus we have $\phi_{u,1}$, $\phi_{u,2}$ and ϕ_{uc} in lines 7, 8, and 9 in Listing 3.1. Similarly, f needs to be modified, which is seen on lines 11, 12, and 13.

Listing 3.1: Definition of the first constructor in `modifiedMixedV3DFvPatchVectorField.C`

```
1  Foam::modifiedMixedV3DFvPatchVectorField::modifiedMixedV3DFvPatchVectorField
2  (
3      const fvPatch& p,
4      const DimensionedField<vector, volMesh>& iF
5  )
6  :
7      refValueU1_(p.size()),
8      refValueU2_(p.size()),
9      refValueUC_(p.size()),
10     refGrad_(p.size()),
11     valueFractionU1_(p.size()),
12     valueFractionU2_(p.size()),
13     valueFractionUC_(p.size()),
```

```
14      source_(p.size(), Zero)
```

Next, the definition of $\phi_{u,1}$, $\phi_{u,2}$, and ϕ_{uc} are given as

$$\phi_{u,1} = f_{u,1}\phi_{\text{ref}}^{u,1} + (1 - f_{u,1})u_2, \tag{3.1}$$

$$\phi_{u,2} = f_{u,2}\phi_{\text{ref}}^{u,2} + (1 - f_{u,2})u_3, \tag{3.2}$$

$$\phi_{uc} = f_{uc}\phi_{\text{ref}}^{uc} + (1 - f_{uc})u_1, \tag{3.3}$$

in lines 11-18, 20-27, and 29-36 respectively, in Listing 3.2. Lastly, the x-, y- and z-components are multiplied by their respective directional unit vector and added to create the boundaries velocity vector which is returned by the `evaluate()` member function on line 40.

Listing 3.2: Definition of the `evaluate` member function in `modifiedMixedV3DFvPatchVectorField.C`

```
1
2   void Foam::modifiedMixedV3DFvPatchVectorField::evaluate(const Pstream::commsTypes)
3   {
4       if (!this->updated())
5       {
6           this->updateCoeffs();
7       }
8
9       vectorField U = this->patchInternalField();
10
11      Foam::scalarField valueU1 =
12      (
13          valueFractionU1_*refValueU1_
14        + (1.0 - valueFractionU1_)
15         *(
16              U.component(1)
17          )
18      );
19
20      Foam::scalarField valueU2 =
21      (
22          valueFractionU1_*refValueU2_
23        + (1.0 - valueFractionU2_)
24         *(
25              U.component(2)
26          )
27      );
28
29      Foam::scalarField valueUC =
30      (
31          valueFractionUC_*refValueUC_
32        + (1.0 - valueFractionUC_)
33         *(
34              U.component(0)
35          )
36      );
37
38      vectorField::operator=
39      (
40          valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*vector(0.0, 0.0, 1.0)
41      );
42
43      fvPatchVectorField::evaluate();
44  }
```

The contributions of the different velocity components to the aforementioned key member functions, $a_{\nabla\cdot}$, $s_{\nabla\cdot}$, a_{∇^2}, and s_{∇^2} are shown in Listing 3.3. On lines 17, 36, 52, and 68, it is seen that the contributions from the x-, y-, and z-directions are multiplied by a unit vector in each respective direction.

Listing 3.3: How the coefficient and source term contributions are returned in `modifiedMixedV3DFvPatchVectorField.C`

```
1
2  Foam::tmp<Foam::vectorField>
3  Foam::modifiedMixedV3DFvPatchVectorField::valueInternalCoeffs
4  (
5      const tmp<scalarField>&
6  ) const
7  {
8      scalarField valueU1 =
9          (1.0 - valueFractionU1_);
10
11     scalarField valueU2 =
12         (1.0 - valueFractionU2_);
13
14     scalarField valueUC =
15         (1.0 - valueFractionUC_);
16
17     return valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*vector(0.0, 0.0,
       1.0);
18 }
19
20
21 Foam::tmp<Foam::vectorField>
22 Foam::modifiedMixedV3DFvPatchVectorField::valueBoundaryCoeffs
23 (
24     const tmp<scalarField>&
25 ) const
26 {
27     scalarField valueU1 =
28         valueFractionU1_*refValueU1_;
29
30     scalarField valueU2 =
31         valueFractionU2_*refValueU2_;
32
33     scalarField valueUC =
34         valueFractionUC_*refValueUC_;
35
36     return valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*vector(0.0, 0.0,
       1.0);
37 }
38
39
40 Foam::tmp<Foam::vectorField>
41 Foam::modifiedMixedV3DFvPatchVectorField::gradientInternalCoeffs() const
42 {
43     scalarField valueU1 =
44          -valueFractionU1_*this->patch().deltaCoeffs();
45
46     scalarField valueU2 =
47          -valueFractionU2_*this->patch().deltaCoeffs();
48
49     scalarField valueUC =
50          -valueFractionUC_*this->patch().deltaCoeffs();
51
52     return valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*vector(0.0, 0.0,
       1.0);
53 }
54
55
56 Foam::tmp<Foam::vectorField>
57 Foam::modifiedMixedV3DFvPatchVectorField::gradientBoundaryCoeffs() const
58 {
59     scalarField valueU1 =
60         valueFractionU1_*this->patch().deltaCoeffs()*refValueU1_;
61
62     scalarField valueU2 =
```

```
63          valueFractionU2_*this->patch().deltaCoeffs()*refValueU2_;
64
65      scalarField valueUC =
66          valueFractionUC_*this->patch().deltaCoeffs()*refValueUC_;
67
68      return valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*vector(0.0, 0.0,
        1.0);
69  }
```

3.2.1 modifiedLODI3D

The next part of the implementations lies in `modifiedLODI3DFvPatchVectorField.*`, where `refValue` and `valueFraction` member data need to be initialized. Since the additional contributions from the 3D extension are present, these are also added to the code, which is seen in Listing 3.4, lines 15 through to 21.

Listing 3.4: Definition of the first constructor in `modifiedLODI3DFvPatchVectorField.C`

```
1  Foam::modifiedLODI3DFvPatchVectorField::modifiedLODI3DFvPatchVectorField
2  (
3      const fvPatch& p,
4      const DimensionedField<vector, volMesh>& iF
5  )
6  :
7      modifiedMixedV3DFvPatchVectorField(p, iF),
8      phiName_("phi"),
9      rhoName_("rho"),
10     fieldInf_(Zero),
11     lInf_(-GREAT),
12     psiName_("thermo:psi"),
13     gamma_(0.0)
14 {
15     this->refValueU1() = Zero;
16     this->refValueU2() = Zero;
17     this->refValueUC() = Zero;
18     this->refGrad() = Zero;
19     this->valueFractionU1() = 0.0;
20     this->valueFractionU2() = 0.0;
21     this->valueFractionUC() = 0.0;
22 }
```

These member data are then updated in `updateCoeffs()`, simply by calling the `field.oldTime().boundaryField()` member function (corresponding to the previous time-step) to the `fvPatch` class and updating it according to their respective definitions. Similarly, to obtain the fields from two time-steps ago, an extra `oldTime()` is included. This is seen in Listing 3.5, lines 37 and 38 for the assigning velocity vector fields, and lines 54 through to 67.

Listing 3.5: A piece of the member function `updateCoeffs()` in `modifiedLODI3DFvPatchField.C`

```
1  void Foam::modifiedLODI3DFvPatchVectorField::updateCoeffs()
2  {
3      if (this->updated())
4      {
5          return;
6      }
7
8      const fvMesh& mesh = this->internalField().mesh();
9
10
11     word ddtScheme
12     (
13         mesh.ddtScheme(this->internalField().name())
14     );
15     scalar deltaT = this->db().time().deltaTValue();
16
```

```
17      const GeometricField<vector, fvPatchField, volMesh>& field =
18          this->db().objectRegistry::template
19          lookupObject<GeometricField<vector, fvPatchField, volMesh>>
20          (
21              this->internalField().name()
22          );
23
24      // Calculate the advection speed of the field wave
25      // If the wave is incoming set the speed to 0.
26      // advection speed U
27      const scalarField wU(Foam::max(advectionSpeed(), scalar(0)));
28      // advection speed U +- C
29      const scalarField wUC(Foam::max(advectionSpeedWT(), scalar(0)));
30
31      // Calculate the field wave coefficient alpha with U and U+-C(See notes)
32      const scalarField alphaU(wU*deltaT*this->patch().deltaCoeffs());
33      const scalarField alphaUC(wUC*deltaT*this->patch().deltaCoeffs());
34
35      label patchi = this->patch().index();
36
37      vectorField Uold = field.oldTime().boundaryField()[patchi];
38      vectorField Uoold = field.oldTime().oldTime().boundaryField()[patchi];
39
40      // Non-reflecting outflow boundary
41      // If lInf_ defined setup relaxation to the value fieldInf_.
42      if (lInf_ > 0)
43      {
44          // Calculate the field relaxation coefficient k (See notes)
45          // K calculated with the advection speed U +- C (not U)
46          const scalarField k(wUC*deltaT/lInf_); // was calculated with wU initially
47
48          if
49          (
50              ddtScheme == fv::EulerDdtScheme<scalar>::typeName
51           || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
52          )
53          {
54              this->refValueU1() = Uold.component(1);
55
56              this->refValueU2() = Uold.component(2);
57
58              this->refValueUC() =
59              (
60                  Uold.component(0) + k*fieldInf_.component(0)
61              )/(1.0 + k);
62
63              this->valueFractionU1() = 1.0/(1.0 + alphaU);
64
65              this->valueFractionU2() = 1.0/(1.0 + alphaU);
66
67              this->valueFractionUC() = (1.0 + k)/(1.0 + alphaUC + k);
68          }
```

The latter set of lines correspond to Eqs. (2.8) and 2.7 for the UC-components on lines 58 - 61 and 67, respectively. The other components correspond to advected components. When f and ϕ_{ref} are multiplied, the first term in Eq. (1.24) is obtained for the UC component, similarly $U1$ and $U2$ components result in the first term for the advected component at the speed U_n, $u_2/(1.0+\alpha_U)$ and $u_3/(1.0+\alpha_U)$.

3.2.2 modifiedLODI3DPR

To further extend the code to the partially reflecting definition, the transmission/reflection coefficient needs to be implemented. This is done by creating a new `scalarField` which is denoted as `PR` for "Partially Reflecting". This is illustrated in Listing 3.6, on line 43. We can note here that there is a new `const scalarField wUC2` on line 41, which collects the minimum advection speed from

the member function `advectionSpeedWT2()`. The definition of this member function is presented in the same listing in lines 1-29 and returns $U_n - c$, corresponding to the wave speed of the waves traveling back into the domain from the outlet. This is needed to describe the new coefficient which is shown in Eq. (1.27), and the minimum value is collected to be consistent with the definition of `advectionSpeedWT1()` which collects the maximum value.

Listing 3.6: Definition of `advectionSpeedWT2()` in `modifiedLODI3DPRFvPatchVectorField.C` and `PR`

```
1  Foam::tmp<Foam::scalarField>
2  Foam::modifiedLODI3DPRFvPatchVectorField::advectionSpeedWT2() const
3  {
4      // Lookup the velocity and compressibility of the patch
5      const fvPatchField<scalar>& psip =
6          this->patch().template
7              lookupPatchField<volScalarField, scalar>(psiName_);
8
9      const surfaceScalarField& phi =
10         this->db().template lookupObject<surfaceScalarField>(this->phiName_);
11
12     fvsPatchField<scalar> phip =
13         this->patch().template
14             lookupPatchField<surfaceScalarField, scalar>(this->phiName_);
15
16     if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
17     {
18         const fvPatchScalarField& rhop =
19             this->patch().template
20                 lookupPatchField<volScalarField, scalar>(this->rhoName_);
21
22         phip /= rhop;
23     }
24
25     // Calculate the speed of the field wave w
26     // by summing the component of the velocity normal to the boundary
27     // and the speed of sound (sqrt(gamma_/psi)).
28     return phip/this->patch().magSf() - sqrt(gamma_/psip); // U - C
29 }
30
31 void Foam::modifiedLODI3DPRFvPatchVectorField::updateCoeffs()
32 .
33 .
34 .
35     // Calculate the advection speed of the field wave
36     // If the wave is incoming set the speed to 0.
37     // advection speed U
38     const scalarField wU(Foam::max(advectionSpeed(), scalar(0)));
39     // advection speed U +- C
40     const scalarField wUC1(Foam::max(advectionSpeedWT1(), scalar(0)));
41     const scalarField wUC2(Foam::min(advectionSpeedWT2(), scalar(0)));
42     // partially reflecting factor
43     const scalarField PR(0.5*(wUC1/wUC2 - 1.0));
44     // Calculate the field wave coefficient alpha with U and U+-C(See notes)
45     const scalarField alphaU(wU*deltaT*this->patch().deltaCoeffs());
46     const scalarField alphaUC(wUC1*deltaT*this->patch().deltaCoeffs());
```

The transmission/reflection coefficient, `PR`, is then added to the transport Eq. (1.29), which is seen in Listing 3.7, on lines 20-29, where it is only added to the transportation of u_1, where u_2 and u_3 remain implemented the same as in `LODI3D`. It can be clearly seen that this term will affect the weighting of how much u_1 will be based on the old boundary value (`Uold.component(0)`), versus the feedback from outside the domain with a prescribed value by the user (`fieldInf_.component(0)`).

Listing 3.7: Placement of `PR` in `modifiedLODI3DPRFvPatchVectorField.C`

```
1
2      // Non-reflecting outflow boundary
```

```
    // If lInf_ defined setup relaxation to the value fieldInf_.
    if (lInf_ > 0)
    {
        // Calculate the field relaxation coefficient k (See notes)
        // K calculated with the advection speed U +- C (not U)
        const scalarField k(wUC1*deltaT/lInf_); // was calculated with wU initially

        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValueU1() = Uold.component(1);

            this->refValueU2() = Uold.component(2);

            this->refValueUC() =
            (
                Uold.component(0) + PR*k*fieldInf_.component(0)
            )/(1.0 + PR*k);

            this->valueFractionU1() = 1.0/(1.0 + alphaU);

            this->valueFractionU2() = 1.0/(1.0 + alphaU);

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

3.3 Compiling

The compilation is identical to Lucchese [1], with the tree structure

```
Make
    linux64GccDPInt32Opt
        fields
            fvPatchFields
fields
    fvPatchFields
        basic
            modifiedMixedV2D
            modifiedMixedV3D
        derived
            modifiedLODI2D
            modifiedLODI3D
            modifiedLODI3DPR
lnInclude
```

where the `Make/files` and `Make/options` are shared such that all the boundary conditions from both `basic` and `derived` folders compile with the same `wmake` command. The `lnInclude` folder is created and collects symbolic links to the respective files to be compiled. The upstream structure (or where to place this tree structure) is done like mentioned in [1], where we follow the consistancy of `OpenFOAM`'s structure where it is placed in `src/finiteVolume` directory. The `files` and `options` files are shown in Listing 3.8 and 3.9, respectively.

Listing 3.8: The `files` file

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

```

```
$(basicFvPatchFields)/modifiedMixedV2D/modifiedMixedV2DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI2D/modifiedLODI2DFvPatchVectorField.C
$(basicFvPatchFields)/modifiedMixedV3D/modifiedMixedV3DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI3D/modifiedLODI3DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI3DPR/modifiedLODI3DPRFvPatchVectorField.C
LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
```

Listing 3.9: The `options` file

```
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
```

Chapter 4

Unconfined Bluff Body

In this chapter the test case will be presented. The test case is based on the 2D flow over a bluff body presented in Lucchese's work, [1], which was inspired by Pirozzoli and Colonius article,[10]. The modifications made are to enable a 3D simulation, which, according to Lucchese, is very important due to the inherently 3D nature of turbulence and vortices. This case is used for this analysis due to the strong vorticity generated at high Reynolds numbers ($\approx$ 40000) behind a bluff body. Furthermore, the outlet is place close to the bluff-body's trailing edge, resulting in strong vorticle structures which need to be transmitted, presenting a difficult case for a wave-transmissive boundary condition. Furthermore, a benefit of this case is that the 2D-version was performed by Lucchese [1] which adds to the discussion when expanding to a 3D space.

4.1 Case Set-up

The case is set up with a characteristic length of the bluff body $L = 0.1$ m, length of the domain is $L_D = 115L$, with a free-stream Mach number of $Ma_\infty = 0.34$, which are all identical to Lucchese, [1]. The 3D extension is kept small to $t = 0.1$ m to reduce the computational cost. An illustration of the mesh is shown in Fig. 4.1, which consists of 200 000 cells, with 10 cells in the spanwise direction to have cells with aspect ratio of approximately 1 in proximity to the bluff body. This implies 20 000 cells on the x-y plane.

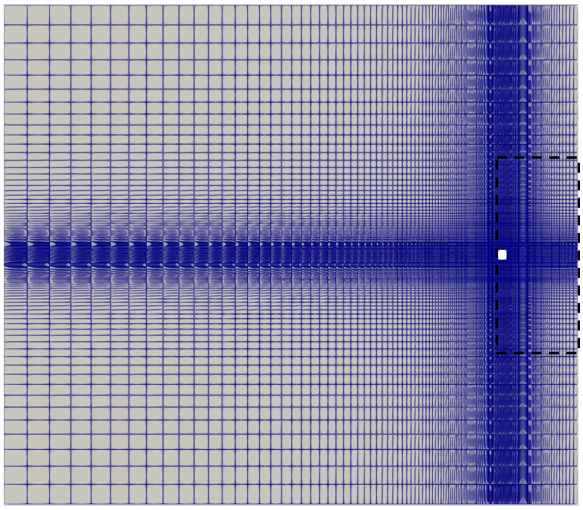

Figure 4.1: An illustration of the bluff body test case's mesh. A dashed zone is included to demonstrate the clip used for the result section.

The boundary condition for the front and back boundaries are set to cyclic as seen in Listing 4.1.

Listing 4.1: Modification of the `blockMeshDict`

```
.
.
.
    front
    {
        type cyclic;
        separationVector (0 0 1);
        neighbourPatch back;
        faces
        (
                        (0 1 2 3)
                        (3 2 20 16)
                        (16 20 21 17)
                        (1 8 9 2)
                        (20 24 25 21)
                        (8 12 13 9)
                        (9 13 28 24)
                        (24 28 29 25)
        );
    }
    back
    {
        type cyclic;
        separationVector (0 0 -1);
        neighbourPatch front;
        faces
        (
                        (7 6 5 4)
                        (19 23 6 7)
                        (18 22 23 19)
                        (6 10 11 5)
                        (22 26 27 23)
                        (10 14 15 11)
                        (27 31 14 10)
                        (26 30 31 27)
        );
    }
);
```

The main variables such as pressure, velocity and temperature are described as seen in Listings 4.2, 4.3, and 4.4. The `freeStreamPressure` and `freeStreamVelocity` are inlet/outlet type boundary conditions that blend a zero-gradient inlet condition with a fixed value outlet condition. It matches the matches the boundary values to the undisturbed free-stream flow condition which allows it to minimize numerical reflections and artefacts [11].

Listing 4.2: Pressure settings in the `0/` directory

```
/*--------------------------------*- 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];
```

```
18
19 internalField   uniform 101325;
20
21 boundaryField
22 {
23     inlet
24     {
25         type            zeroGradient;
26     }
27
28     outlet
29     {
30         type            waveTransmissive;
31         gamma           1.4;
32         fieldInf        101325;
33         lInf            10;
34         value           $internalField;
35     }
36
37     upperAndLower
38     {
39         type            freestreamPressure;
40         freestreamValue $internalField;
41     }
42
43     obstacle
44     {
45         type            zeroGradient;
46     }
47     front
48     {
49         type            cyclic;
50     }
51     back
52     {
53         type            cyclic;
54     }
55
56 }
57
58
59 // ************************************************************************* //
```

In Listing 4.3, line 31, the type of outlet boundary condition is varied between the settings mentioned in Table 4.1.

Listing 4.3: Velocity settings in the `0/` directory

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  v2112                                 |
5  |   \\  /    A nd           | Website:  www.openfoam.com                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       volVectorField;
13     object      U;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [0 1 -1 0 0 0 0];
18
19 internalField   uniform (120 0 0);
20
```

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

    outlet
    {
        type            LODI2D; // modifiedLODI3D; // modifiedLODI3DPR; // waveTransmissive;
        value           $internalField;
        field           U;
        gamma           1.4;
        rho             rho;
        lInf            10;
        fieldInf        (120 0 0);
    }

    upperAndLower
    {
        type            freestreamVelocity;
        freestreamValue $internalField;
    }

    obstacle
    {
        type            noSlip;
    }
    front
    {
        type            cyclic;
    }
    back
    {
        type            cyclic;
    }

}


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

Listing 4.4: Temperature settings in the `0/` directory

```
/*--------------------------------*- 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
{
    inlet
```

```
    {
        type            zeroGradient;
    }

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

    upperAndLower
    {
        type            zeroGradient;
    }

    obstacle
    {
        type            zeroGradient;
    }
    front
    {
        type            cyclic;
    }
    back
    {
        type            cyclic;
    }

}


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

The `controlDict` is shown in Listing 4.5, where the time-step is controlled by the Courant number such that $CFL < 0.3$ to ensure numerical stability and better pressure-velocity coupling. To reduce computational cost, the PISO loop is used, instead of PIMPLE loop (PISO but with multiple `nOuterCorrector` loops). The statistics are gathered after 0.1 s for 0.3 s, corresponding to approximately three flow-through times. Additionally, a probe located 0.3 m behind the bluff body is used to analyze the vortex-shedding behavior.

Listing 4.5: Settings in the `controlDict` directory

```
/*--------------------------------*- 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       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     rhoPimpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;
```

```
24
25 endTime         0.4;
26
27 deltaT          2e-6;
28
29 writeControl    adjustableRunTime;
30
31 writeInterval   0.01;
32
33 purgeWrite      0;      // was 10
34
35 writeFormat     ascii;
36
37 writePrecision  8;
38
39 writeCompression off;
40
41 timeFormat      general;
42
43 timePrecision   6;
44
45 runTimeModifiable true;
46
47 adjustTimeStep  yes;
48
49 maxCo           0.3;
50
51 maxDeltaT       1;
52
53 functions
54 {
55     fieldAverage
56     {
57         type            fieldAverage;
58         libs            (fieldFunctionObjects);
59         enabled         true;
60         timeStart       0.10;
61         timeEnd         1;
62         writeControl    writeTime;
63         fields
64         (
65             U
66             {
67                 mean        on;
68                 prime2Mean  on;
69                 base        time;
70             }
71
72             p
73             {
74                 mean        on;
75                 prime2Mean  on;
76                 base        time;
77             }
78         );
79     }
80     probes
81     {
82         type probes;
83         functionObjectLibs ("libsampling.so");
84         name probes;
85         probeLocations
86         (
87            ( 10.2 5 0 )
88
89         );
90         fields ( p T U );
91         writeControl   timeStep;
```

```
92          writeInterval  1;
93      }
94  }
95
96  libs ("libmyFiniteVolume.so");
97
98  // ************************************************************************* //
```

A table with all of the cases are presented in Table 4.1, where all the cases share the same boundary conditions for p, T, and the same setting for l_∞, which stays consistent with Lucchese's work [1], such that they can be compared and used together. The influence of l_∞ is not investigated, however, as described in Eqs. (1.28) and (2.13), k may be interpreted as a spring stiffness constant which is decreased with increasing l_∞ which affects the boundary's feedback amplitude.q

Table 4.1: Boundary conditions of 3D-bluff body test-case

	Case 1	**Case 2**	**Case 3**	**Case 4**
p	waveTransmissive	waveTransmissive	waveTransmissive	waveTransmissive
U	waveTransmissive	LODI2D	LODI3D	LODI3DPR
T	advective	advective	advective	advective
lInf	10	10	10	10

4.2 Results and Discussion

In this section the results will be presented and discussed, including mean and root-mean-squared (rms) statistics on the velocity and pressure fields, fast Fourier transform of the y-velocity component behind the bluff body to analyze the vortex-shedding behavior, time-delay embedding and lastly, the computational costs are compared. The mean and rms values are used to give an overview of the time-averaged differences between the cases as the instantaneous ones may differ simply due to a slight phase change between each other. The phase changes and temporal differences are illustrated with time-delay embedding which indicates the trajectories of a 1D-data-series over time in a phase space. A frequency analysis is performed to reveal difference in the underlying vortex-shedding behavior based on outlet boundary conditions.

The mean and rms pressure are presented in Fig. 4.2 and show that the statistical fields are very similar to each other. The main differences are seen in the rms fields, where Case 1 with `waveTransmissive` boundary condition leads to the largest at 5.64 kPa, and Case 3 with `LODI3D` the lowest rms amplitude of 5.3 kPa in the wake region in comparison to 5.61 kPa, 5.59 kPa, for Case 2 and 4, respectively.

Moving over to Fig. 4.3, all the cases' mean values except for Case 3 with `LODI3D` are slightly bottom favored, which is confirmed when analysing the rms values where Case 3 shows very symmetric mean velocity fluctuations. This does not mean the other cases will not converge to a similar result, but within the same physical time it indicates `LODI3D` helps converge the solution faster.

The statistics on the y-component of velocity seen in Fig. 4.4 are very similar as well, and difficult to comment. But when moving over to the z-component of velocity in Fig. 4.5, Case 2 with `LODI2D` show numerical artifacts by the boundary, which is not seen in the other cases. The numerical artifacts are expected as `LODI2D` is made for 2D- and not 3D-cases. Note that the amplitudes are very small. It is peculiar that this occurs in Case 2 but not Case 1 which also neglects transporting the z-velocity component, indicating an unknown influence, potentially due to a stiffness issue when one transverse component is treated, whilst one is left untreated. If both are left untreated, the conservational properties of the finite volume method could potentially use both transverse components to negate their errors. Another possibility which could be in conjunction with the aforementioned explanation is if there is a default treatment to the transverse components, such as `zeroGradient` which could lead to a more well-behaved scenario, than one component `zeroGradient` and another being treated as `advective`. This does however show some resilience

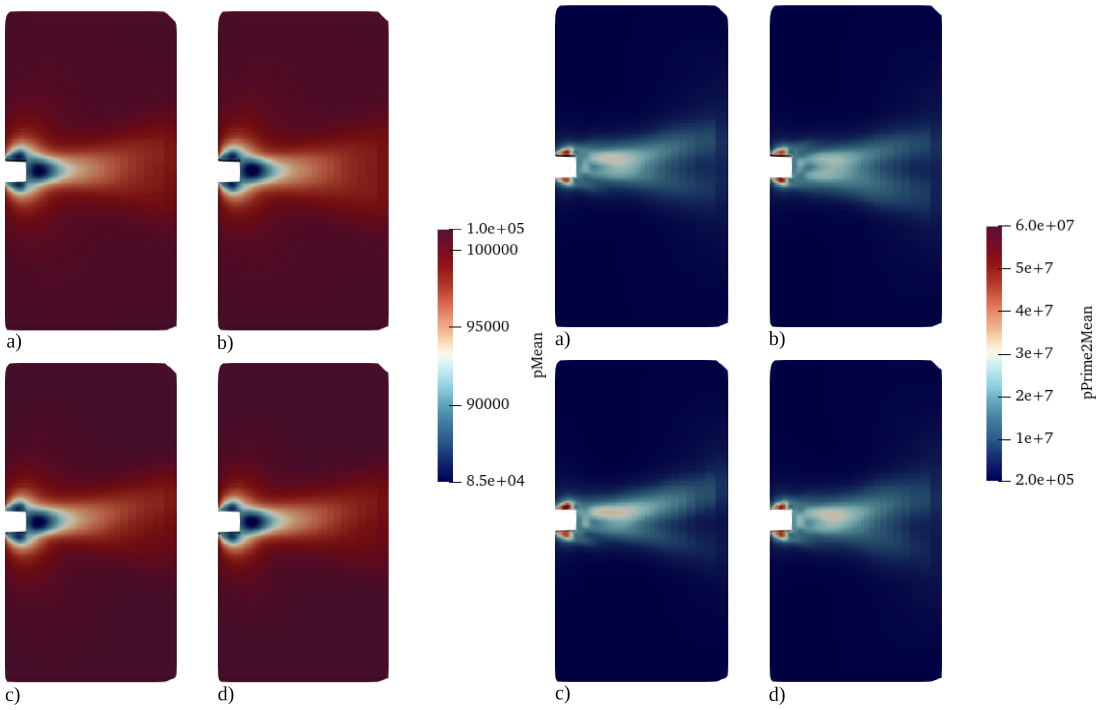

Figure 4.2: Mean (left group of panels) and rms (right group of panels) pressure-fields. The panels correspond to a) `LODI2D`, b) `LODI3D`, c) `waveTransmissive`, and d) `LODI3DPR`.

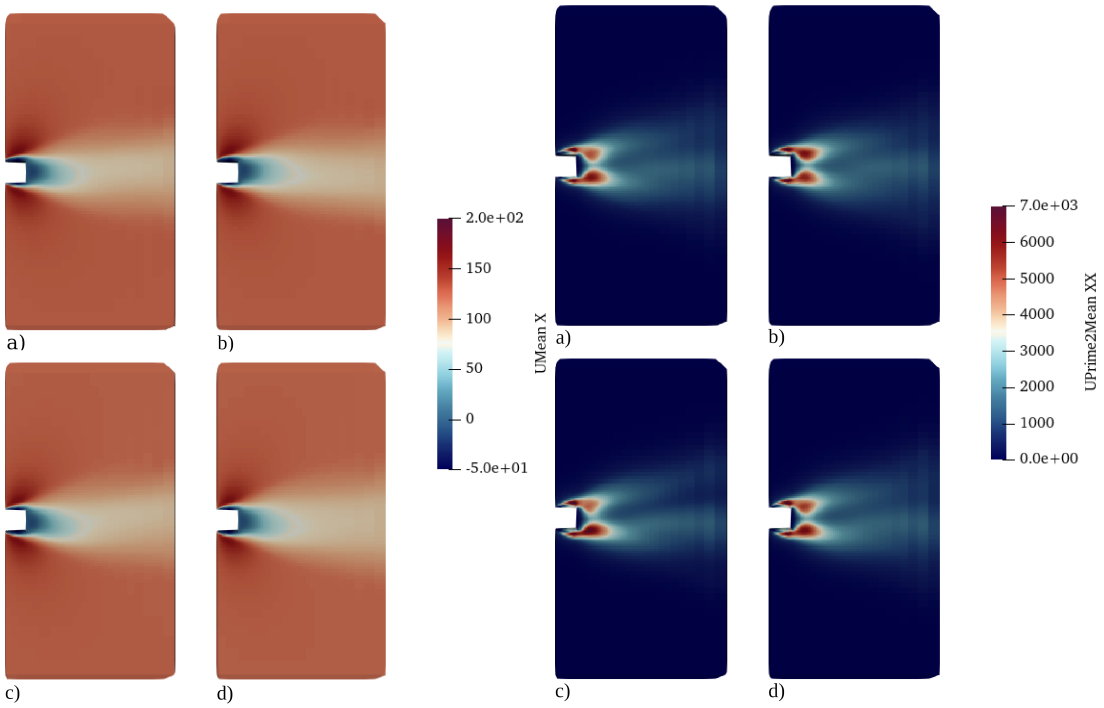

Figure 4.3: Mean (left group of panels) and rms (right group of panels) x-velocity-fields. The panels correspond to a) `LODI2D`, b) `LODI3D`, c) `waveTransmissive`, and d) `LODI3DPR`.

in the established `waveTransmissive` boundary condition. Cases 1 and 3 are quite similar in their mean and rms values here, and an increased amount of fluctuations and mean values are present for Cases 2 and 4 which indicates a portion of the incident z-velocity is reflected. In the latter case, this is exactly what is expected due to the partially reflecting derivation. It should also be noted that it is done without the presence of numerical artifacts in the same magnitude, successfully mimicking a partially reflective boundary. The same observations are made when analyzing the other z-dependent components of the rms velocity in Fig. 4.6.

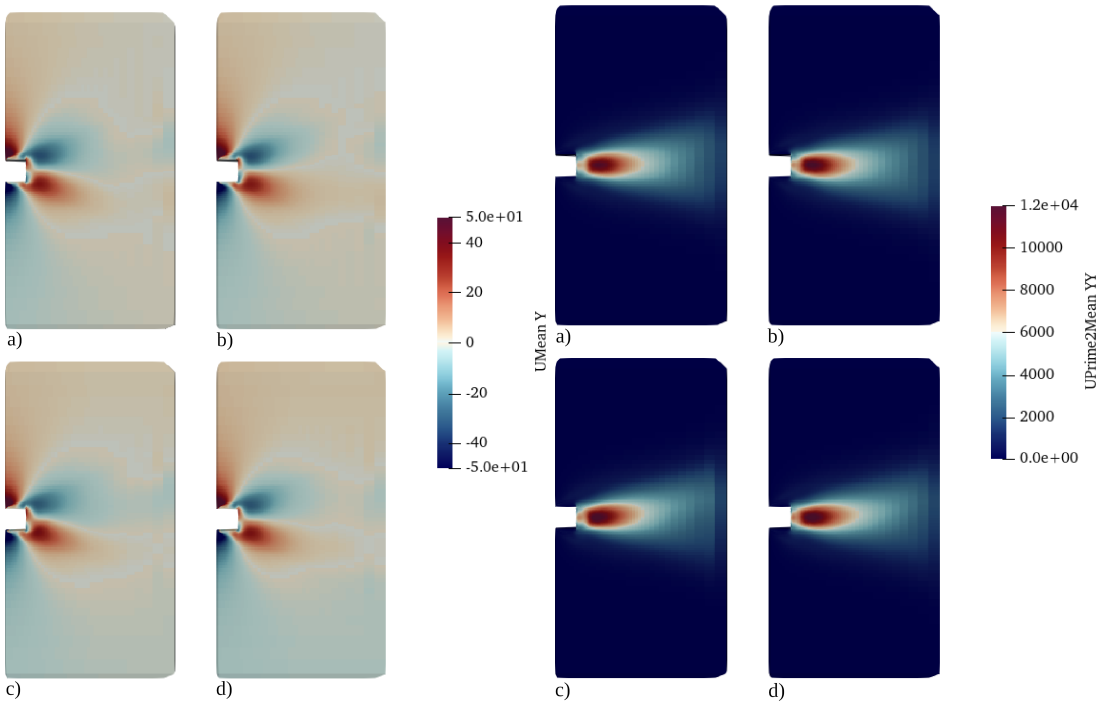

Figure 4.4: Mean (left group of panels) and rms (right group of panels) y-velocity-fields. The panels correspond to a) `LODI2D`, b) `LODI3D`, c) `waveTransmissive`, and d) `LODI3DPR`.

A common way to analyze bluff-body flows is to check the frequency of vortex-shedding. In Fig. 4.7, the FFT of each case time-series[1] for the y-component of the velocity behind the bluff body is performed. The FFT indicates that all cases share the same maximum peak, but there is a spreading of the shedding frequency for Case 4, a double peak for Case 1, a triple peak for Case 2 and a clear peak for Case 3. The spreading of the shedding frequency for Case 4 is expected due to the increased amplitude of the perturbations which interact with the hydrodynamic instability, slightly knocking the dynamical system off its stable trajectory which later begins to approach back to.

Another note is that some of these may be artifacts of the signal length as an FFT approach assumes a periodic signal, and stitches the beginning with the end of the supplied dataseries. The stitching is done using a window operator, which in this case, is the Kaiser window.

Furthermore, Case 3 shows the clearest vortex-shedding frequency peak which is also connected to the highest amplitude. This is a good indication of the effectiveness with the 3D implementation. The vortex-shedding faces less perturbations which would otherwise be present in the case of reflections at the boundary.

For nonlinear dynamical system, a popular theory to apply is the Koopman theory. It implies that, for Hamiltonian systems, you may inflate your data into a larger dimensional space, which eventually leads to a linear instead of nonlinear system [12]. In conjunction with such a theory, a common method of analysis is the so-called Time-Delay Embedding (TDE) (based on Taken's

[1]This time-series is obtained from the probe which is shown in Listing 4.5, between lines 80 to 93.

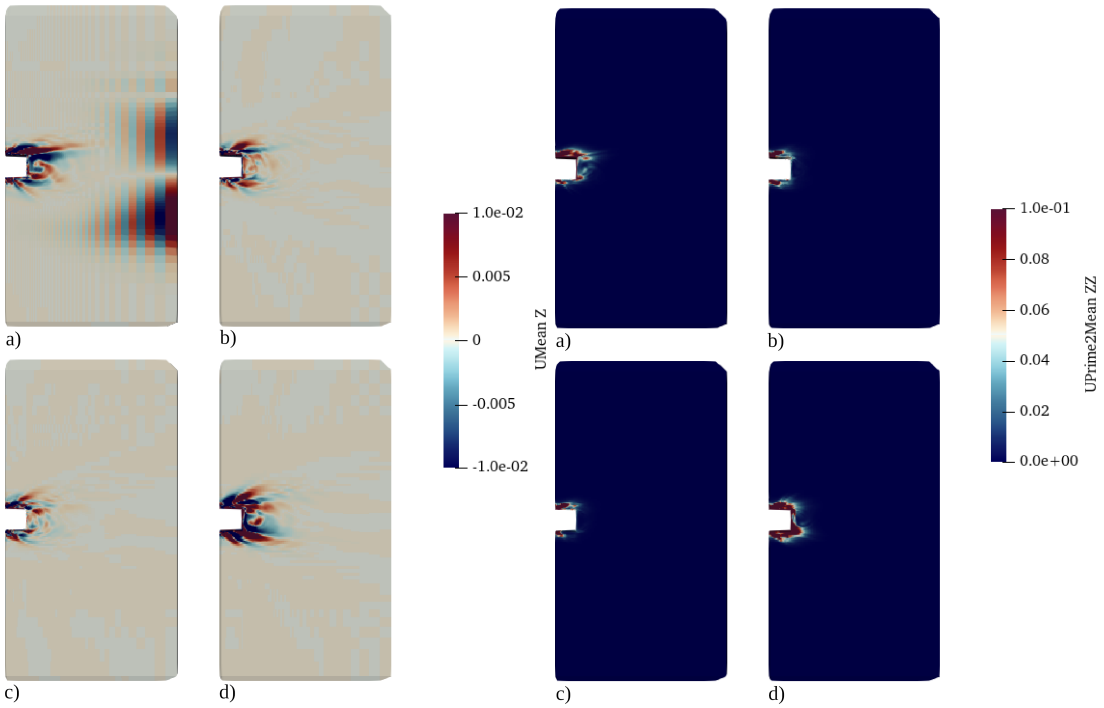

Figure 4.5: Mean (left group of panels) and rms (right group of panels) z-velocity-fields. The panels correspond to a) `LODI2D`, b) `LODI3D`, c) `waveTransmissive`, and d) `LODI3DPR`.

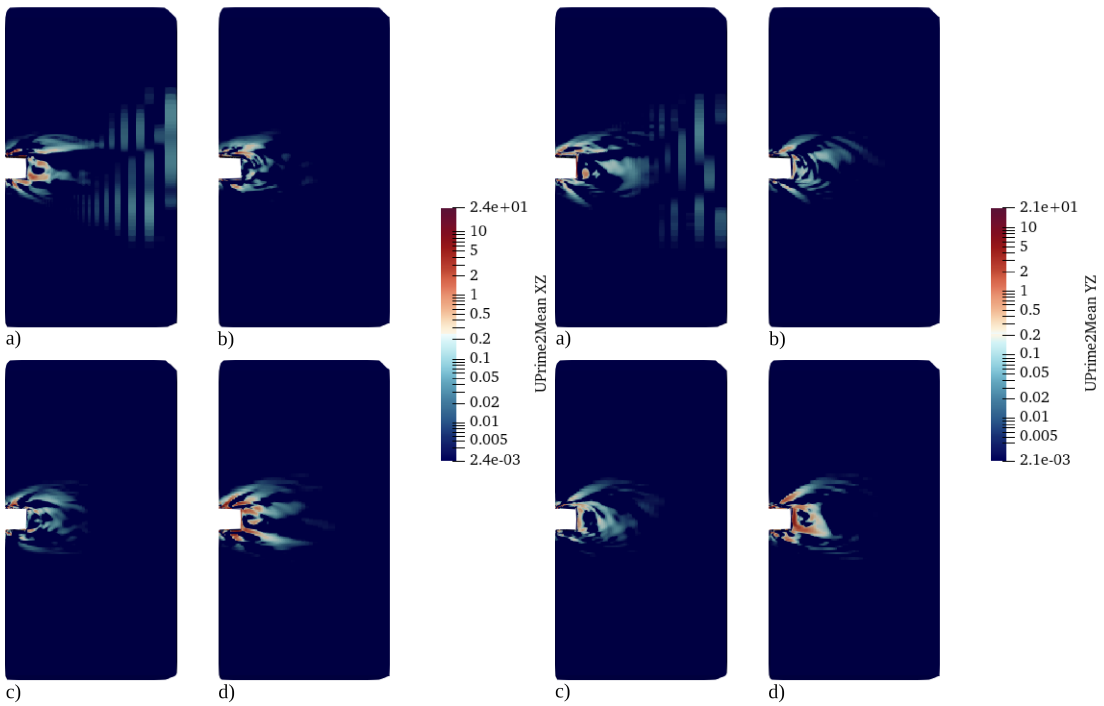

Figure 4.6: The rms values of the xz (left) and yz (right) components. The panels correspond to a) `LODI2D`, b) `LODI3D`, c) `waveTransmissive`, and d) `LODI3DPR`.

Theorem[2]), which consists of constructing a Hankel matrix where you superimpose a new column corresponding to the same time-series but time-delayed by a factor τ. This is repeated until the prescribed number of dimensions are reached [13]. In this case it is done until the basis functions are simple sinusoidals, indicating linear basis functions. The settings correspond to $\tau = 20$ and expanding the 1D signal to $\mathbb{R}^{100}$. The Hankel matrix, H, is decomposed using Singular Value Decomposition (SVD), into

$$H = USV^T,$$

where V^T is right singular vectors which are ordered from most to least contribution to the system. Thus the three most dominant modes are plotted which is seen in Fig. 4.8. These can e.g. represent different terms such as how the pressure affects velocity, or quadratic velocity terms etc. The trajectories of Cases 1, 2, 3 and 4 are all very similar, leading to a limit-cycle with some intermittency disrupting the limit-cycle behavior which is seen in Fig. 4.8 in the panel to the right. The initial starting point of the trajectory differs from each case, indicating an influence on the initial condition by the outlet boundary condition.

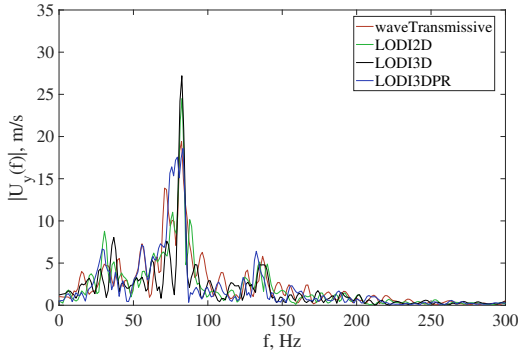

Figure 4.7: Fast Fourier transform of the y-velocity component behind the bluff body for Cases 1, 2, 3, and 4.

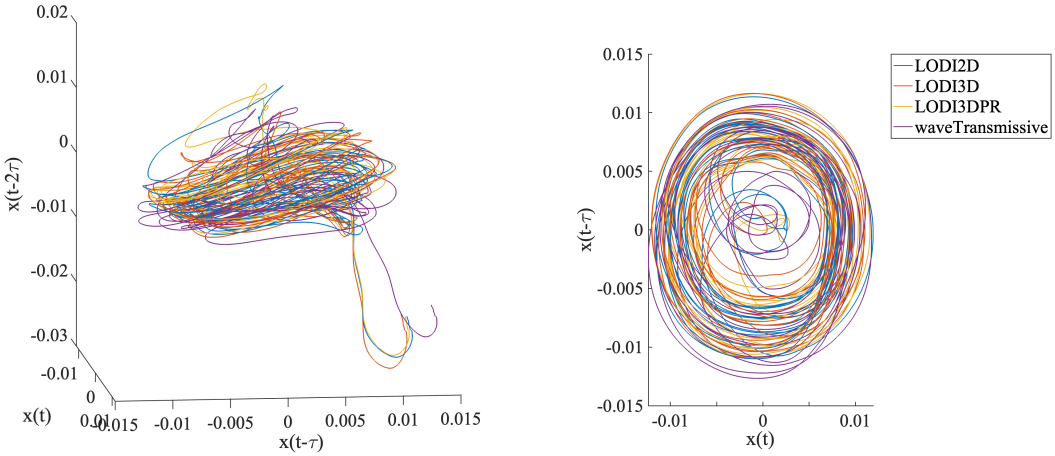

Figure 4.8: Time-delay embedding of the y-velocity components behind the bluff body for Cases 1, 2, 3, and 4. Left panel includes a 3D view and the right panel reveals the 2D view of a limit-cycle type of behavior.

[2]The fundamental assumption of Taken's Theorem is such that the reconstruction of the dynamical system preserves diffeomorphism, however, it does not preserve the geometric shape of the structures in space. This leads to an optimisation problem where we need to find the best parameters to make the reconstruction interpretable [13].

A last note regarding the results for the different boundary conditions is that the computational cost corresponds to 5 and 32 core hours for Case 1, and Case 2, 3, and 4, respectively. This difference, similarly to the lack of reflections and numerical artifacts in Case 1 indicates a difference in the implementation which optimizes the implementation and reduces reflections. This begs the question, "how do these boundary conditions scale?". This is left for a future analysis.

Conclusion

In this work, two new boundary conditions were created: `LODI3D` and `LODI3DPR` with the basic boundary condition implemented in `mixedV3D`. These correspond to 3D implementation of the `waveTransmissive, advective, mixed` class combinations, as well as an extension to include a reflection coefficient based on a derivation from the LODI relations. A guide to how and where to compile the code was given, together with a test case using an unconfined bluff-body simulation which compared the existing boundary conditions, `waveTransmissive` and `LODI2D`, with the new `LODI3D` and `LODI3DPR` boundary conditions. The comparison led to the following conclusions:

- The mean and root-mean-squared values are similar to a large extent between all the boundary conditions, indicating only a small change on the overall statistics for the given case. It is worth noting that, for the same physical time elapsed, `LODI3D`'s statistics seemed to converge the earliest.
- The boundary condition by Lucchese [1], `LODI2D`, show numerical artifacts in the z-velocity field which are removed once extending the boundary condition to 3D. These numerical artifacts were not present in the `waveTransmissive` boundary condition, which is not understood.
- The frequency contents are similar in all cases, but a cleaner peak is present for Case 3 with `LODI3D` than the others, which may be due to less reflections, leading to less nonlinear interactions. Furthermore, the time-delay embedding based on Taken's theorem was performed and the trajectories were very similar, but phase-delayed resulting in a slight nudge in the trajectories between cases as well as indicating a limit-cycle behavior with a certain intermittent disruption which was seen for all cases.
- Lastly, the computational cost for the simulations based on regular classes such as `LODI2D` and `LODI3D`, was 32 core hours, whilst the templated class, `waveTransmissive`, only cost 5 core hours. The reason for this difference in not understood, and their scaling with increased resolution is not yet known either.

Bibliography

[1] L. Lucchese, "Implementation of non-reflecting boundary conditions in OpenFOAM." *In Proceedings of CFD with OpenSource Software*, 2022.

[2] H. K. Versteeg and W. Malalasekera, *Description of Accurate Boundary Conditions for the Simulation of Reactive Flows.* Pearson, 2007, pp. 40–114.

[3] T. J. Poinsot and S. K. Lele, "Boundary Conditions for Direct Simulations of Compressible Viscous Flows," *Journal of Computational Physics*, vol. 101, pp. 104–129, 1992.

[4] T. Baritaud, T. Poinsot, and M. Baum, *Description of Accurate Boundary Conditions for the Simulation of Reactive Flows.* Editions Technip, 1998, pp. 11–32.

[5] J. Nordström, "The use of characteristic boundary conditions for the navier-stokes equations," *Computers Fluids*, vol. 24, no. 5, pp. 609–623, 1995.

[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] P. M. Morse and K. U. Ingard, *Theoretical acoustics.* Princeton university press, 1986, chapter 9.

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

[9] C. S. Yoo, Y. Wang, A. Trouvé, and H. G. Im, "Characteristic boundary conditions for direct simulations of turbulent counterflow flames," *Combustion Theory and Modelling*, vol. 9, no. 4, pp. 617–646, 2005.

[10] 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.

[11] "OpenFOAM," https://doc.openfoam.com/2312/tools/processing/boundary-conditions/rtm/derived/inletOutlet/freestreamPressure/, accessed: 2025-01-15.

[12] S. L. Brunton, M. Budišić, E. Kaiser, and J. N. Kutz, "Modern koopman theory for dynamical systems," *arXiv preprint arXiv:2102.12086*, 2021.

[13] H. Kantz and T. Schreiber, *Advanced embedding methods.* Cambridge University Press, 2003, p. 143–173.

Study questions

1. How is the coefficient K defined, and what does it do to the solution?
2. What is the benefit of using a wave transmissive boundary condition?
3. Why is it so important to generate boundary conditions based on the NSCBC principal?
4. What is the purpose of `valueInternalCoeffs`, `valueBoundaryCoeffs`, `gradientInternalCoeffs`, and `gradientBoundaryCoeffs`? What makes them differ from one another?
5. What is the purpose of the `waveTransmissive` class?
6. How would you implement the partially reflecting code for the pressure? What would you need to change?