Cite as: Vergassola, M.: A continuous forcing immersed boundary approach to solve the VARANS equations in a volumetric porous region. In Proceedings of CFD with OpenSource Software, 2021, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2021

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

A continuous forcing immersed boundary approach to solve the VARANS equations in a volumetric porous region

Developed for OpenFOAM-v2006

Author:
Marco Vergassola
Delft university of technology
M.Vergassola@tudelft.nl

Peer reviewed by:
Lorenzo Melchiorri
Dr. Oriol Colomés
Dr. Saeed Salehi

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 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 18, 2022

Learning outcomes

The main requirement 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:

- Make use of tutorials `movingCylinderInChannelIco` and `porousDamBreak` to show how to work with `pimpleDyMIbFoam`, a solver from `foam-extend` which utilizes the immersed boundary method, and the porosity model of the toolbox `olaFlow`.
- Discuss what files should be included and what options are present. This shows how to use the immersed boundary library and how to introduce a fixed porous region in the domain.

The theory of it:

- Theory of the immersed boundary method, different approaches from the literature.
- Theoretical pressure drop model across a porous barrier.

How it is implemented:

- A look at how the previously described theory is implemented. Both porous models and immersed body method are implemented in different solvers in `OpenFOAM` and `foam-extend`. Therefore, a description of how these are implemented in the existing solvers will be provided.

How to modify it:

- How to modify the penalty force of the immersed boundary method in case of a porous, rather than impermeable, object.
- How to treat the mesh cells at the immersed boundary to improve accuracy and surface sharpness.

Prerequisites

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

- Basic object-oriented programming.
- How to run standard document tutorials like damBreak tutorial.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric
- How to customize a solver and do top-level application programming.
- Access to `OpenFOAM v2006` with `olaFlow` toolbox[1].
- Access to `foam-extend 4.1`[2].

[1] The `olaFlow` toolbox is necessary only to run the tutorial and check the solver files. It is not a requirement to obtain the new solver.

[2] `foam-extend 4.1` is necessary only to run the tutorial and check the class files. It is not a requirement to obtain the new solver.

Contents

Nomenclature

Acronyms

`OpenFOAM`	Open-source Field Operation And Manipulation
CFD	Computational Fluid Dynamics
FSI	Fluid Structure Interaction
IB	Immersed Boundary
IBM	Immersed Boundary Method
MRF	Multiple Reference Frame
PVF	Porous Volume Fraction
VARANS	Volume-Averaged Reynolds-averaged Navier-Stokes
VOF	Volume of Fluid

English symbols

A	Difference terms matrix IBM
a	Coefficient linear porous drag term
b	Coefficient quadratic porous drag term
C	Coefficients matrix IBM
c	Coefficient transient porous drag term
D	Cylinder diameter $[m]$
D_{50}	Nominal diameter porous material $[m]$
I	Hydraulic gradient $[Nm/kg]$
KC	Keulegan-Carpenter number
M	Geometric matrix IBM
n	Porosity
p	Total pressure $\left[N/m^2\right]$
p^*	Pseudo-dynamic pressure $\left[N/m^2\right]$
Re	Reynolds number
U	Velocity vector $[m/s]$
u	Velocity component $[m/s]$
V	Cell volume $\left[m^3\right]$
W	Weighting matrix IBM
X	Position component $[m]$

Greek symbols

β	Relaxation factor IBM
Γ	Fluid-solid mesh boundary
γ	PVF scalar field
κ	Interface curvature
μ	Dynamic viscosity $[kg/(ms)]$
ν	Kinematic viscosity $\left[m^2/s\right]$
Ω	Mesh region
ρ	Density $\left[kg/m^3\right]$

σ	Surface tension coefficient

Subscripts

f	Fluid
s	Solid

Other symbols

FTR	`OpenFOAM` triangulated format
STL	Stereolithography

Chapter 1

Introduction

In this section, an overview of the report is presented. This work is developed within the context of the PhD course 'CFD with OpenSource Software' provided by the Chalmers University of Technology in the person of Håkan Nilsson [1]. The aim of this work is to develop a continuous forcing immersed boundary approach for the Volume-Averaged Reynolds-averaged Navier-Stokes (VARANS) equations in a dynamic porous region. The idea is to start from the already implemented porous models in `OpenFOAM` and combine it with what can be used of the discrete forcing immersed boundary library developed in `foam-extend`.

The study of the interaction between waves and porous structures is of great importance in offshore engineering [2, 3]. In fact, these are often used to dissipate the energy of incoming waves and protect structures, objects or land. For this reason, with time a few porous models have been implemented in `OpenFOAM` and other CFD solvers and most of these are focused on the simulation of large porous structures such as breakwaters. However, in recent years, these models have been applied with success to smaller structures, such as thin perforated plates and cylinders [4]. Traditionally, these structures were simulated introducing a rather complex conformal mesh which requires great refinement at the voids of the structure and the use of a turbulence model. On the other hand, the use of a porous representation of such structures eliminated the need for a very dense mesh and, in some cases, also the need for a turbulence model. The main drawback of such an approach is that it can estimate properly only the global quantities, e.g. the total hydrodynamic force on the structure. For a detailed analysis of the flow within and around the voids, a conformal mesh is required. However, for most cases treated in offshore engineering, often the interest lies in the macroscopic effects.

Currently, in `OpenFOAM`, regardless of the model used, it is only possible to define a static porous region. This derives from the main use of these models which focuses on large fixed structures. However, for the alternative applications aforementioned, it would be useful to be able to move these regions by imposing a predetermined displacement or by computing the forces on the structure. The latter, for example, would enable the use of porous regions for the study of Fluid-Structure-Interaction (FSI) problems involving a floating porous object. Generally speaking, in `OpenFOAM`, a volumetric porous region is introduced by marking a set of cells that defines the location of the region. Then, based on the location in the domain, the momentum equation is modified, accordingly with the selected theory, to account for the pressure drop caused by the porous volume. In particular, in this work we will refer to the implementation used in a toolbox developed specifically for offshore CFD simulations called `olaFlow` [5]. This type of implementation recalls that of the immersed boundary method (IBM) in its continuous forcing formulation. In fact, for this type of IBM, a set of cells are marked to track the location of a fictitious solid within the fluid mesh. Then, a forcing term is introduced in the momentum equation to simulate the presence of a body. An IBM library developed by Jasak et al. [6] is available in `foam-extend`. Together with this library, also two transient solvers are available, one for incompressible flows and another for transonic/supersonic compressible flows. However, this library is based on the discrete forcing approach which makes use of interpolated boundary conditions to model the presence of a solid. Therefore, a new solver

is developed to model volumetric porous regions using a continuous forcing immersed boundary approach. Differently from the IBM available in `foam-extend`, this approach solves the flow both outside and inside the immersed boundary, making it better suitable for the simulation of porous objects. Moreover, compared to the porous media implemented in `olaFlow` and `OpenFOAM`, the solver developed in this work does not require a conformal mesh. This can represent an important advantage when dealing with complex geometries or moving objects. Currently, it is not possible to simulate moving porous bodies, however, the implementation of such a feature is straightforward for the new solver.

Within this context, Chapter 2 presents the theoretical background necessary to understand the implementations of the porous model and the immersed boundary method. Chapter 3 describes the already existing solvers and libraries and how they can be used, including two short tutorials. A deeper look into the implementation of these is also presented. Chapter 4 presents the modification introduced in order to develop the IBM porous solver starting from the existing codes and the `pimpleFoam` solver of `OpenFOAM`. In this section, a validation of the newly developed solver is also presented. Finally, Chapter 5 explains how to set up a simple tutorial to test the new solver[1].

[1]Both the solver and the tutorial files are available for download at this repository

Chapter 2

Background

In this section, the theory of the porous model implemented in `olaFlow` and that of the immersed boundary method are briefly presented.

2.1 Porous media flow for Navier–Stokes equations

In this section, only a brief description of the theory is provided and the reader is referred to the work of Hsu et al. [7], del Jesus et al. [8] and Higuera et al. [9] for a deeper literature review. A general description of the two main methods used to model porous media flow is first provided. Following, the specific theory implemented in `OpenFOAM` is presented in greater detail.

When dealing with porous media flow, it is possible to classify the different methods used to treat this problem as microscopic and macroscopic approaches. The microscopic approach is the most intuitive as it is based on the exact representation of the porous material. The problem with this approach is that it is inapplicable in most fields of research. In fact, this approach would require the exact geometry of the object to be known and the mesh to be able to accurately model elements characterized by a large variety of scales and shapes. Such an approach would provide a very detailed description of the flow inside the porous media, however, it is often of greater interest the global effect that these objects have on the flow. Therefore, an alternative approach has been developed which aims at describing the global effect while drastically reducing the computation cost, i.e. the macroscopic approach. This is based on volume average Navier-Stokes equations and it considers only the macroscopic characteristics of the porous media. In this approach, the effect of the porous media on the flow is modelled using drag forces and the momentum equation is modified to account for a reduced fraction of fluid. To model these drag forces typically the Darcy-Forchheimer model is employed. The reader is referred to the work of Cimolin and Discacciati [10] for a detailed review of this model. There are different ways to average the Navier-Stokes equations but the implementation used in `olaFlow` makes use of VARANS equations. These are here presented in the form proposed by del Jesus et al. [8] for a multiphase problem.

$$\frac{\partial}{\partial x_i}\frac{u_i}{n} = 0, \tag{2.1}$$

$$\frac{\partial}{\partial t}u_i + u_j\frac{\partial}{\partial x_j}\frac{u_i}{n} = -\frac{n}{\rho}\frac{\partial}{\partial x_i}p + ng_i + n\frac{\partial}{\partial x_j}\left(\nu\frac{\partial}{\partial x_i}\frac{u_i}{n}\right) - au_i - bu_i|u_i| - c\frac{\partial}{\partial t}u_i, \tag{2.2}$$

$$\frac{\partial \alpha_1}{\partial t} + \frac{\partial}{\partial x_i}\frac{u_i}{n}\alpha_1 = 0, \tag{2.3}$$

where u is often referred to as the Darcy velocity, n is the porosity, ρ is density, p is the pressure and ν is the kinematic viscosity. Eq. (2.3) is the Volume of Fluid (VOF) advection equation and α_1 represents the volume fraction of water in a cell. On the right-hand side of Eq. (2.2), the last three terms appear as closure terms to account for frictional and pressure forces and added mass in the porous media at the microscopic level. The linear term, also known ad Darcy term, is associated with

viscous effects and dominates at low Re numbers. The quadratic term, also known as Forchheimer term, models the turbulent drag at high Re numbers. Finally, the transient term accounts for fluid acceleration through voids. The three terms together represent the hydraulic gradient, I, experienced by the flow across a porous media.

$$I = au - bu|u| - c\frac{\partial u}{\partial t}, \tag{2.4}$$

where the coefficients a, b and c are related to the properties of the porous media and their relative importance depends on the flow conditions. According to the formulation proposed by van Gent [11], the coefficients a and b can be defined as

$$a = \frac{\alpha}{\rho}\frac{(1-n)^3}{n^2}\frac{\mu}{D_{50}^2}, \tag{2.5}$$

$$b = \beta\left(1 + \frac{7.5}{\mathrm{KC}}\right)\frac{1-n}{n^2}\frac{1}{D_{50}}, \tag{2.6}$$

where D_{50} is the mean nominal diameter of the material and KC is the Keulegan-Carpenter number. The coefficients α, β and c often require calibration from physical tests to obtain accurate results although empirical formulations can be found in the literature for different flow regimes. It should be noted that Eq. 2.5 and 2.6 are strictly valid only for oscillating flows over granular medias and more general relations can be used to describe these coefficients a and b. However, these are the formulations implemented in `olaFlow` and therefore presented here.

2.2 Immersed boundary method

Generally speaking, the interaction between fluids and solids can be modelled either with a mesh-conforming defined-body method or with an Immersed Boundary Method (IBM). The first approach is typically preferable for fixed rigid bodies and simple geometries. In fact, in the case of a moving object, the mesh-conforming methods may lead to a distorted grid and it often requires remeshing every few time steps. Similarly, for complex geometries, the generation of a conformal mesh might be a time-consuming and complex task. By contrast, the IBM eliminates the grid remeshing problem completely and therefore it is a valid alternative to simulate moving objects. At the same time, it also removes the problem of the mesh generation since it usually relies on a simple Cartesian grid. The IBM makes use of a extended mesh Ω which cover both fluid, Ω_f, and solid, Ω_s, regions such that $\Omega = \Omega_f \cup \Omega_s$ and $\Gamma_s = \Omega_f \cap \Omega_s$ represents the fluid-solid boundary. The name IBM refers to all those numerical methods which manipulate the governing equations to account for this boundary. Many different formulations of the IBM have been developed but, in general terms, these can be divided into continuous and discretized forcing approaches depending on when the boundary conditions are introduced into the equations. In the continuous forcing approach, the forcing is introduced in the momentum equation as a source term before the discretization. On the other hand, in the discretized forcing approach, the governing equations are firstly discretized and only afterwards the forcing is introduced by means of modification of the discretized equations in the vicinity of the immersed boundary. More details on these approaches can be found, for instance, in the work of Peskin [12] and Senturk et al. [13].

Due to the large variety of different IBMs, only a few examples of continuous and discrete forcing approaches are here described. For the former, usually, the procedure starts by defining the forcing, in the form of a source term, for instance

$$F = \beta\gamma_s\left(u_s - u\right), \tag{2.7}$$

where u_s is the solid velocity and u, often called monolithic velocity, is equal to $\gamma_s u_s + \gamma_f u_f$. Here, γ_* is a scalar field defined as $\gamma_* = V_*/V$, being V_* and V the volume fraction of the phase and the cell volume respectively. A schematic of the computational domain for the IBM is shown in Figure 2.1. The force F weakly imposes equality for the solid and monolithic velocity at the boundary Γ_s and

inside the solid region, Ω_s. It can be seen that the forcing term is non-zero only where γ_s is non-zero as well. The coefficient β is a relaxation factor that dictates how quickly the solid and monolithic velocities equal one another at the interface. This β value can assume different shapes and sometimes its formulation determines the difference between one IBM and another. For instance, Viré et al [14] defines this relaxation factor as

$$\beta = \max\left(\frac{\rho_f}{\Delta t}, \frac{\nu}{L^2}\right), \tag{2.8}$$

so that β depends on whether the interaction is dominated by inertia, $\rho_f/\Delta t$, or viscous,ν/L^2, effects. Here, ρ is density, ν is viscosity, Δt is the time step and L is the local edge length. Several formulations of the forcing term have been proposed over time. For instance, Lai and Peskin [15] treat the solid boundary as a spring system. Exactly like a spring, the boundary reacts to the fluid compressing or extending it and for a large enough stiffness, the boundary becomes rigid. Another example, perhaps pertinent to the current work, is the formulation proposed by Khadra et al. [16]. Here, the solid is treated like a porous medium having permeability nearly equal to zero. The relaxation coefficient is defined as the ratio of viscosity over permeability, K and the forcing term vanishes for $K \gg 1$, i.e. in the fluid. Several other formulations can be found in the literature. The main drawbacks of all these formulations are the low accuracy, they are mostly first order, and the stability problems for large time steps. Moreover, many of these present user-defined parameters in the forcing term which require tuning.

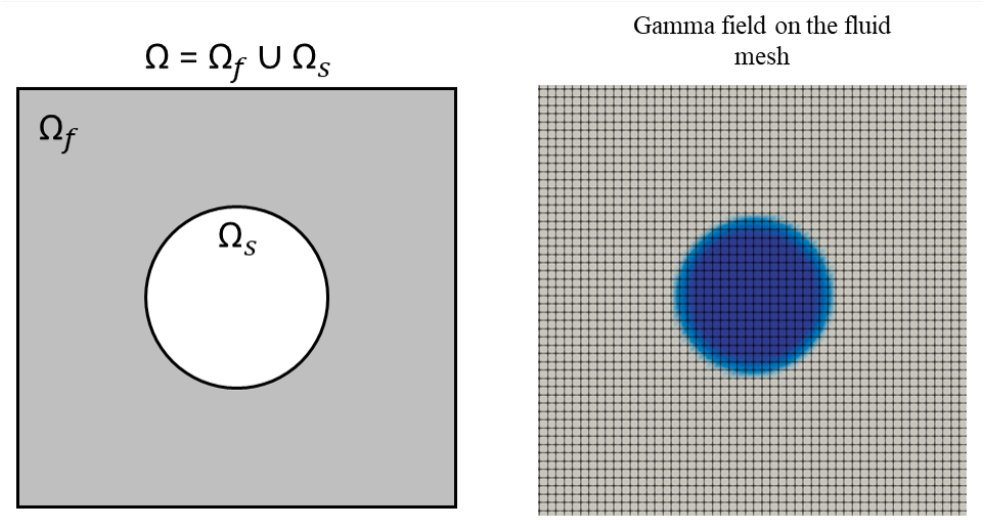


Figure 2.1: Schematic of the computational domain for the immersed body method (left) and example of the fluid mesh containing a solid concentration field used to represent the body (right)

Within this framework, the discrete forcing approach was developed to find an IBM formulation able to retain second-order accuracy and simultaneously eliminate the stability problems. As previously mentioned, in this approach the forces exerted by the solid boundary on the fluid are introduced directly in the discretized governing equations. Mohd-Yusof [17] proposed to reconstruct the velocity field at the immersed boundary to satisfy a no-slip condition. To compute the forcing term, it is necessary to make a prediction based on the current time step. The no-slip condition is then obtained by imposing that the tangential velocity fields inside and outside of the immersed boundary (IB) are opposing each other. This method has the advantage that it retains the accuracy of the interpolation method used to compute the tangential velocity field inside the IB. However, it leads to mass conservation problems in the boundary cells [18].

As mentioned above, the discrete forcing approach satisfies the boundary condition retaining the same accuracy of the interpolation method. Therefore, Fadlun et al. [19] investigated the use of three different approaches to reconstruct the velocity at the IB. For the first method, the boundary

is approximated by grid cells in a stepwise fashion. The second approach introduces a weighting factor accounting for the volume of the cells cut by the IB, similar to a VOF approach. In the last method, the velocity at the boundary is linearly interpolated to that of the first exterior node. This last method resulted to be the best as it could achieve the second-order accuracy. However, because of the linear interpolation, the mesh needs to be refined close to the IB. At the same time though, this approach allows the use of large time steps and therefore it can speed up the computation significantly.

Another slightly different approach is the ghost-cells one. It is relevant to present it because is the most similar to that implemented in `foam-extend 3.2` and newer versions. In this approach, the cells of the base mesh are intersected with the IB to identify the fluid, solid and IB (or ghost) cells. A schematic is presented in Figure 2.2. In this method, the flow is resolved only in the fluid and IB cells while the solid cells are kept inactive. Proper values are then assigned in the IB cells

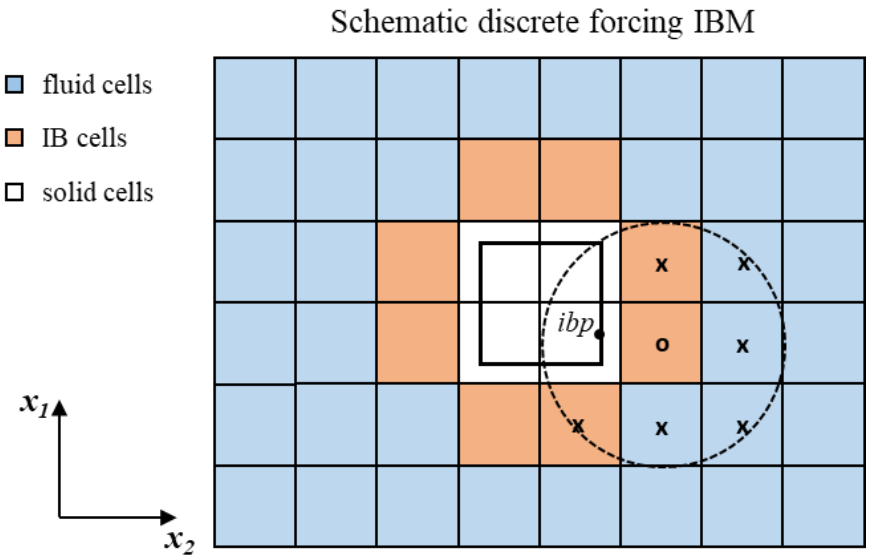


Figure 2.2: Schematic of the IBM implemented in `foam-extend 3.2`

(orange cells) via a weighted least squares interpolation to impose the boundary conditions on the fluid-solid interface. For the interpolation, different stencils can be used. In this example, the stencil consists of six cells identified by a circle centred in the IB cell. The boundary condition for a variable ϕ is then obtained by minimization of the summation

$$\sum_{i=1}^{m} w_i \left(\phi - \tilde{\phi}^2 \right), \tag{2.9}$$

The m number of cells on which the summation is minimized depends on the stencil. The formulation of $\tilde{\phi}$ once again will depend on the chosen interpolation. An example is provided for a Dirichlet boundary condition in 2D [13],

$$\tilde{\phi}_i = \phi_{ibp} + C_0 X_i + C_1 Y_i + C_2 X_i Y_i + C_3 X_i^2 + C_4 Y_i^2, \tag{2.10}$$

where the X_i and Y_i are the coordinate distance of the cell i from the *ibp* point (see Figure 2.2). The coefficients C_* are obtained from $C = \left(M^T W M\right)^{-1} M^T W A$, where A contains the difference terms of Eq. (2.9), W is the weighting matrix and M contains the geometric information. For more details, the reader is referred to the work of Jasak et al. [6] and Senturk et al. [13]. It is relevant to notice that also for this approach there are problems with the mass conservation at the IB cells. In fact, when a Dirichlet boundary condition is imposed at the IB for the velocity, this will lead to either an addition or subtraction of fluid in the cell. This violates the mass conservation and requires

a flux balance correction, this is even more important for a moving IB. Despite the correction, this remains one of the main drawbacks of this approach as it leads to spikes in the forces computed over the body. On the other hand, differently from the continuous forcing approach, this method ensures a sharp representation of the smooth-enough fluid-solid interfaces.

Chapter 3

Exiting solvers and libraries

3.1 olaFlow porous media implementation

The `olaFlow` toolbox is a free and open source project, born as continuation of the `IHFOAM` and `olaFoam` projects [5]. This toolbox aims at making the latest advances in the simulation of wave dynamics available for the `OpenFOAM` and `foam-extend` communities. It provides utilities, solvers and boundary conditions for the active generation and absorption of water waves at the boundaries. It also allows to simulate the interaction of these with porous, floating or static structures in both coastal and offshore environments. The `olaFlow` toolbox is provided with one main solver which is called `olaFlow` as well. The structure of the solver is presented below.

olaFlow solver structure

```
olaFlowOFv19xx-21xx/
|-- alphaCourantNo.H
|-- alphaEqn.H
|-- alphaEqnSubCycle.H
|-- alphaSuSp.H
|-- correctPhi.H
|-- createAlphaFluxes.H
|-- createFields.H
|-- createPorosity.H
|-- initCorrectPhi.H
|-- Make
|   |-- files
|   |__ options
|-- olaFlow.C
|-- pEqn.H
|-- rhofs.H
|-- setDeltaT.H
|-- setRDeltaT.H
|__ UEqn.H
```

If we compare it to the structure of the most similar `OpenFOAM` solver, namely `interFoam`, we can easily identify the differences.

interFoam solver structure

```
interFoam/
|-- alphaSuSp.H
|-- correctPhi.H
|-- createFields.H
|-- initCorrectPhi.H
|-- interFoam.C
|-- Make
|   |-- files
|   |__ options
|-- pEqn.H
```

```
|-- rhofs.H
|__ UEqn.H
```

The `.C` files contain the main function of the two solvers while the `.H` files contain pieces of code. Some of the files missing in `interFoam` are simply included differently in the solver. An example is the `alphaCourantNo.H` file which can be seen in the structure tree of `olaFlow`. In `interFoam`, this is included in the `overInterDyMFoam` subfolder which was not included in this structure for clarity. Said this, the main differences between the two solvers can be identified in the `createPorosity.H` and `UEqn.H` files. The former defines all the parameters used to describe the porous zone physical characteristic (the coefficients A, B and c discussed in Chapter 2) and it is presented below. Please note that only the most relevant lines are kept. The `porosityIndex` field was not introduced previously. This is used to identify the location of the porous media in the flow. This field is equal to 1 where the flow interacts with the media and 0 elsewhere. In the lines 24 to 39 of the `createPorosity.H` extract this index is used to allocate the defined non-uniform coefficient fields.

createPorosity.H

```
1   ...
2
3   volScalarField porosityIndex
4   (
5       IOobject
6       (
7           "porosityIndex",
8           runTime.timeName(),
9           mesh,
10          IOobject::READ_IF_PRESENT,
11          IOobject::NO_WRITE
12      ),
13      mesh,
14      dimensionedScalar( "porosityIndex", dimless, 0.0 )
15  );
16
17  ...
18
19  if( activePorosity )
20  {
21
22      ...
23
24      forAll(porosityIndex, item)
25      {
26          if( porosityIndex[item] > 0.0 )
27          {
28              aPorField[item] = aPor[porosityIndex[item]];
29              bPorField[item] = bPor[porosityIndex[item]];
30              cPorField[item] = cPor[porosityIndex[item]];
31              KCPorField[item] = KC;
32              D50Field[item]  = D50Por[porosityIndex[item]];
33              porosity[item]  = phiPor[porosityIndex[item]];
34              if ( useTransient )
35              {
36                  useTransMask[item] = 1.0;
37              }
38          }
39      }
40      porosityF = fvc::interpolate(porosity);
41
42      // Write out porosity
43      porosity.write();
44  }
```

Before looking at the `UEqn.H` file of `olaFlow`, a short discussion about how Eq. (2.2) and Eq. (2.1) should be modified to fit the way `OpenFOAM` formulates RANS equations is introduced. OpenFOAM typical RANS equations for two incompressible fluids are presented below. This is also the formu-

lation used in `interFoam`.

$$\nabla \cdot U = 0, \tag{3.1}$$

$$\frac{\partial \rho U}{\partial t} + \nabla \cdot (\rho U U) - \nabla \cdot (\mu_{\text{eff}} \nabla U) = -\nabla p^* - g \cdot X \nabla \rho + \sigma \kappa \nabla \alpha_1, \tag{3.2}$$

$$\frac{\partial \alpha_1}{\partial t} + \nabla \cdot U \alpha_1 + \nabla \cdot U_c \alpha_1 (1 - \alpha_1) = 0 \tag{3.3}$$

Here, U is the velocity vector, p^* is the pressure deprived of its hydrostatic part, μ_{eff} is the effective dynamic viscosity, X is the position vector, σ is the surface tension coefficient, and κ is the curvature of the interface. The last equation is the VOF advection equation. In order to adapt the VARANS equations presented in Chapter2 to the `OpenFOAM` formulation, Eq. (2.1) and Eq. (2.2) need to be reformulated in terms of intrinsic velocity, $U^* = U/n$. If both sides of these equations are multiplied by ρ and divided by n, then Eq. (2.1) and Eq. (2.2) become

$$\frac{\partial}{\partial x_i} \frac{u_i}{n} = 0, \tag{3.4}$$

$$(1 + c) \frac{\partial}{\partial t} \frac{\rho u_i}{n} + \frac{u_j}{n} \frac{\partial}{\partial x_j} \frac{\rho u_i}{n} = -\frac{\partial}{\partial x_i} p + \rho g_i + \frac{\partial}{\partial x_j} \left(\mu \frac{\partial}{\partial x_i} \frac{u_i}{n} \right) - a \frac{u_i}{n} - b \frac{u_i}{n} \left| \frac{u_i}{n} \right| \tag{3.5}$$

that are identical to the RANS formulation, Eq. (3.1) and (3.2), if we apply the substitution for the intrinsic velocity U^*. Here, the density has been incorporated into the coefficients A and B. The so obtained equations are implemented in the solver `olaFlow`. The UEqn.H file of the solver, containing the momentum equation, is presented below. Here, it can be seen that the momentum equation derived in the last step, Eq. (3.5), is implemented precisely in the same manner in the solver. The term `useTransMask` is used to switch between two different formulations of the B coefficient. The `MRF.DDt(rho, U)` term is introduced to take care of Multiple Reference Frame (MRF) problems that are outside of the scope of this work and won't be discussed here.

Momentum equation olaFlow

```
    MRF.correctBoundaryVelocity(U);

    surfaceScalarField muEff
    (
        "muEff",
        mixture.muf()
      + fvc::interpolate(rho*turbulence->nut())
    );

    fvVectorMatrix UEqn
    (
        (1.0 + cPorField) / porosity * fvm::ddt(rho, U)
        + (1.0 + cPorField) / porosity * MRF.DDt(rho, U)
        + 1.0/porosity * fvm::div(rhoPhi/porosityF, U)
        - fvm::laplacian(muEff/porosityF , U)
        - 1.0/porosity * ( fvc::grad(U) & fvc::grad(muEff) )
        // Closure Terms
        + aPorField * pow(1.0 - porosity, 3) / pow(porosity,3)
        * mixture.mu() / pow(D50Field,2) * U
        + bPorField * rho * (1.0 - porosity) / pow(porosity,3) / D50Field
        * mag(U) * U *
        // Transient formulation
        (1.0 + useTransMask * 7.5 / KCPorField)
    ==
        fvOptions(rho, U)
    );

    UEqn.relax();

    fvOptions.constrain(UEqn);

```

```
32      if (pimple.momentumPredictor())
33      {
34          solve
35          (
36              UEqn
37          ==
38              fvc::reconstruct
39              (
40                  (
41                      mixture.surfaceTensionForce()
42                    - ghf*fvc::snGrad(rho)
43                    - fvc::snGrad(p_rgh)
44                  ) * mesh.magSf()
45              )
46          );
47
48          fvOptions.correct(U);
49      }
```

3.1.1 The porousDamBreak tutorial

In `olaFlow` the `porousDamBreak` tutorial is called `CR35_damBreak` and its structure tree is presented below. This case simulates a two-phase incompressible fluid interacting with a porous medium after a dam-break-like event. A schematic of the case is presented in Figure 3.1. The blue area represent the water reservoirs before the dam break event. The red area identifies the location of the porous region. The domain is modelled as a box without the lid.

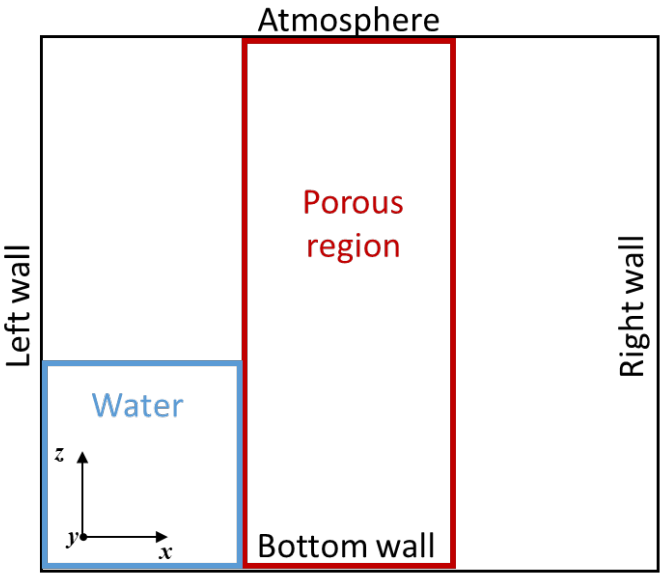


Figure 3.1: Schematic of the simulation domain including boundary information

porousDamBreak tutorial tree structure

```
1 CR35_dambreak/
2 |-- 0.org
3 |   |-- alpha.water
4 |   |-- alpha.water.org
5 |   |-- porosityIndex
6 |   |-- porosityIndex.org
7 |   |-- p_rgh
8 |   |__ U
```

```
|-- constant
|   |-- dynamicMeshDict
|   |-- g
|   |-- porosityDict
|   |-- transportProperties
|   |__ turbulenceProperties
|__ system
    |-- blockMeshDict
    |-- controlDict
    |-- decomposeParDict
    |-- fvSchemes
    |-- fvSolution
    |__ setFieldsDict
```

Differently from single-phase simulations, the boundary and initial conditions are stored in a folder called `0.org`. This is done because information about the field will be written in the `alpha.water` and `porosityIndex` files, making them unusable for a different mesh. Therefore, the original files are kept in the `0.org` folder and this is copied into the `0` folder before running the simulation. Focusing on the inputs which are specific to the porous medium, the first file to analyse is the `porosityIndex`. This is presented below. It can be seen that the boundary conditions and initial conditions are the same as those imposed for the `alpha.water` as both fields are used for interface tracking. The `empty` boundary condition for the `defaultFaces` patch means that the simulation is 2D.

porosityIndex

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      porosityIndex;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    leftWall
    {
        type            zeroGradient;
    }
    rightWall
    {
        type            zeroGradient;
    }
    lowerWall
    {
        type            zeroGradient;
    }
    atmosphere
    {
        type            zeroGradient;
    }
    defaultFaces
    {
        type            empty;
    }
}


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

The next file that needs to be included is the `porosityDict`, a dictionary providing values for the different coefficients of the porous drag terms and located in the `constant` folder. Below, it is possible to see how the coefficients a, b and c are provided. The first number indicates the number of materials (always at least two, porous and non-porous regions) while in brackets the actual value of the coefficient is provided. Typically, the first material is the fluid interacting with the porous object and therefore all the coefficients equal either zero or one in order to deactivate the drag terms in the momentum equation. The last option, `useTransiente`, if set to true, activate a formulation of the b coefficient for which the KC number is assumed to be equal to unity.

porosityDict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      porosityDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

a               2(0 50);
b               2(0 2.0);
c               2(0 0.34);

D50             2(1 0.0159);
porosity        2(1 0.49);

useTransient    false;
// ************************************************************************* //
```

Finally, the `setFieldsDict` file present in the `system` folder has to be modified from that of the traditional `damBreak tutorial` in order to include the definition of the non-uniform `porosityIndex` field. The content of this is presented below and it can be seen that the implementation is identical to that of the `alpha.water` field.

setFieldsDict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      setFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

regions
(
    boxToCell // Water reservoir
    {
        box (-1 -1 -1) (0.29 1 0.35);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
    boxToCell // Water bottom level
    {
```

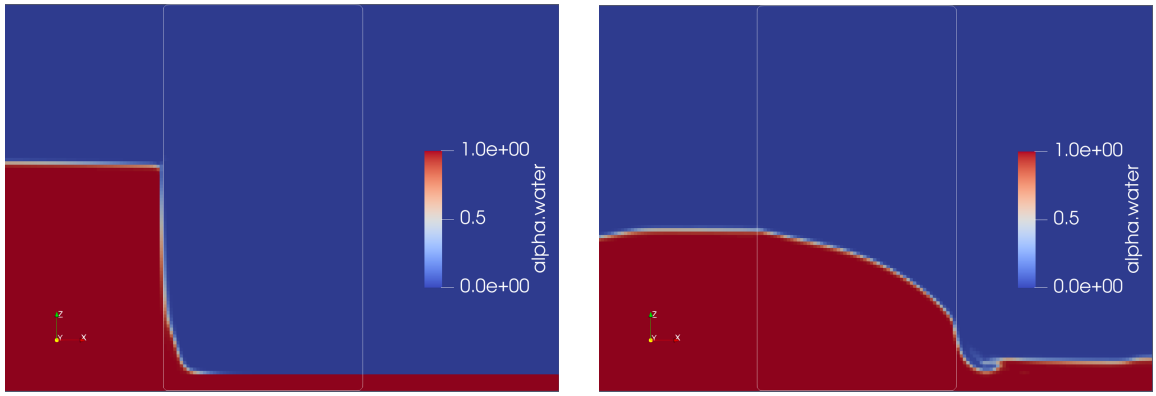
```
        box (-1 -1 -1) (1 1 0.025);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }

    boxToCell // Porous zone
    {
        box (0.30 -1 -1) (0.59 1 0.6);
        fieldValues
        (
            volScalarFieldValue porosityIndex 1
        );
    }
);


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

Finally, to run the tutorial the following lines should be copied in a terminal window in which the `OpenFOAM` installation has been sourced. The `>*.log` piece of command will write the output into `.log` files rather than in the terminal.

Run script

```
$ blockMesh > blockMesh.log
$ cp -r 0.org 0
$ setFields > setFields.log
$ olaFlow > olaFlow.log
```

To visualise the results, one can type `paraFoam` in a terminal window in which the `OpenFOAM` installation has been sourced. Figures 3.2a and 3.2b presents the results for the `alpha.water` field at t=0.05s and t=1.0s respectively.



(a) CR35 damBreak tutorial results at t=0.05s (b) CR35 damBreak tutorial results at t=1.0s

Figure 3.2: CR35 damBreak tutorial results showing the alpha.water field. The white contour line identifies edges of the porous region

3.2 Immersed boundary method implementation

The IB method implemented in `foam-extend` was developed by Jasak et al. [6] and follows the discrete forcing approach with direct boundary condition imposition using a ghost-cell-like method as described in the previous chapter. The IBM of `foam-extend` consist of three main classes:

- `class immersedBoundaryPolyPatch`

- `class immersedBoundaryFvPatch`
- `class immersedBoundaryFvPatchFields`

Depending on the `foam-extend` version, these classes may differ in terms of what tasks each of them does. For instance, in `foam-extend 3.2`, the first class includes the basic mesh support for the IB mesh. Nevertheless, in `foam-extend 4.1` this class also takes care of identifying the IB mask cells, i.e. solid, fluid and IB cells as shown below. In this class, the solid cells are identified as "dead", or inactive (see lines 40 to 46).

Class declaration immersedBoundaryPolyPatch, extract from the .H file

```
1  class immersedBoundaryPolyPatch
2  :
3      public polyPatch
4  {
5      ...
6
7      // Member Functions
8
9          // Access
10
11             //- Return immersed boundary surface mesh
12             const triSurfaceMesh& ibMesh() const
13             {
14             return ibMesh_;
15             }
16
17             //- Return true if solving for flow inside the immersed boundary
18             bool internalFlow() const
19             {
20             return internalFlow_;
21             }
22
23             // Is this a wall patch?
24             virtual bool isWall() const
25             {
26             return isWall_;
27             }
28
29             //- Return triSurface search object
30             const triSurfaceSearch& triSurfSearch() const;
31
32             //- Return true if immersed boundary is moving
33             bool movingIb() const
34             {
35             return movingIb_;
36             }
37
38         ...
39
40         // Immersed boundary blanking
41
42             //- Return dead cells
43             const labelList& deadCells() const;
44
45             //- Return dead faces
46             const labelList& deadFaces() const;
47
48     ...
49 };
```

The second class takes care of defining the IB mask. This is shown below in the extract from the `immersedBoundaryFvPatch.H` file. In this class, also the geometric information required to construct the interpolation matrices are computed, i.e. intersection points, face normals and distances. These pieces of information are then used to construct the interpolation matrix.

Class declaration immersedBoundaryFvPatch, extract from the .H file

```
class immersedBoundaryFvPatch
:
    public fvPatch
{
    // Private data

        //- Reference to immersed boundary patch
        const immersedBoundaryPolyPatch& ibPolyPatch_;

        //- Finite volume mesh reference
        const fvMesh& mesh_;

...

        // Access to immersed boundary components

            //- Return reference to immersed boundary polyPatch
            const immersedBoundaryPolyPatch& ibPolyPatch() const
            {
                return ibPolyPatch_;
            }

            //- Return immersed boundary surface mesh
            const triSurfaceMesh& ibMesh() const
            {
                return ibPolyPatch_.ibMesh();
            }
};
```

Finally, the last class is responsible for the evaluation of the boundary conditions. Therefore, in this class the interpolation of the field data at the IB is computed to impose the proper boundary conditions. Nowadays, a few different boundary conditions are defined on top of Dirichlet and Neumann ones. Below, the structure of the class is provided.

Class immersedBoundaryFvPatchFields structure, for clarity, not all the files are shown

```
immersedBoundaryFvPatchFields/
|--  basic
|   |-- fixedValueIbFvPatchField
|   |   |-- fixedValueIbFvPatchField.C
|   |   |__ fixedValueIbFvPatchField.H
|   |-- mixedIbFvPatchField
|   |   |-- mixedIbFvPatchField.C
|   |   |__ mixedIbFvPatchField.H
|   |__ zeroGradientIbFvPatchField
|       |-- zeroGradientIbFvPatchField.C
|       |__ zeroGradientIbFvPatchField.H
|-- derived
|   |__ movingImmersedBoundaryVelocity
|       |-- movingImmersedBoundaryVelocityFvPatchVectorField.C
|       |__ movingImmersedBoundaryVelocityFvPatchVectorField.H
|-- immersedBoundaryFieldBase
|   |-- immersedBoundaryFieldBase.C
|   |__ immersedBoundaryFieldBase.H
|__ immersedBoundaryFvPatchField
    |-- immersedBoundaryFvPatchField.C
    |__ immersedBoundaryFvPatchField.H
```

The `basic` folder contains the definition of the basic boundary conditions, i.e. Dirichlet, Neumann and Robin. The `derived` folder includes all the other boundary conditions, only one in this case. In the `immersedBoundaryFieldBase` folder there are some functions for dealing with the dead cells and updating them, including passage of information between partitions for parallel simulations. Finally, the files in the `immersedBoundaryFvPatchField` folder contain the definition of the templated class. This is shown below in the extract from the `immersedBoundaryFvPatchField.H` file. Templated

classes are typically employed to create a class whose functionality can be adapted to more than one type without repeating the entire code for each type.

Template class declaration immersedBoundaryFvPatchField, extract from the .H file

```
template<class Type>
class immersedBoundaryFvPatchField
:
    public fvPatchField<Type>,
    public immersedBoundaryFieldBase<Type>
{
public:

    //- Runtime type information: constrained type
    TypeName(immersedBoundaryFvPatch::typeName_());

...

};
```

In `foam-extend 4.1`, two additional classes are present, the `class immersedBoundaryDynamicMesh` and the `class immersedBoundaryTurbulence`. The first is responsible for managing the movement of the IB and the dynamic mesh refinement around it. The second mainly introduces wall function boundary conditions adapted to the case of an IB patch. Finally, of course, `foam-extend 4.1` includes also specific solvers written to work with an IB patch. However, they do not differ much from their standard version and the investigation of these is left to the reader.

3.2.1 The movingCylinderInChannelIco tutorial

In `foam-extend 4.1` the `movingCylinderInChannelIco` tutorial simulates a single-phase incompressible constant flow around an oscillating immersed cylindrical boundary. A schematic of the case is presented in Figure 3.3. Here, a no-slip boundary condition is imposed on the sides of the domain making it a channel. Although the name suggests that the `icoFoam` solver is used, in reality,

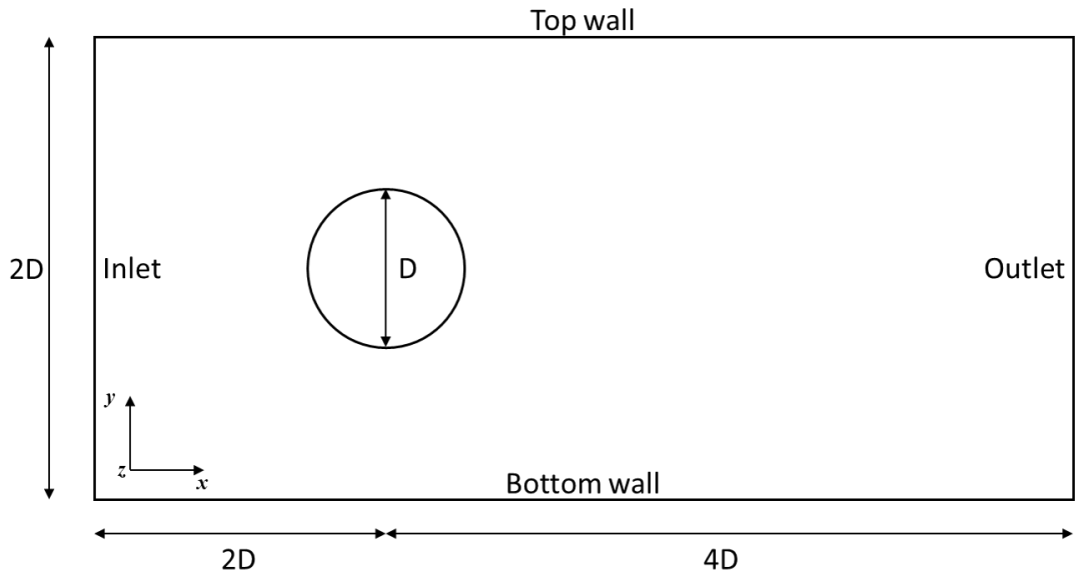


Figure 3.3: Schematic of the simulation domain including boundary information

the IBM version of the `pimpleFoam` solver is used, namely `pimpleDyMIbFoam`. This solver differs from the standard `pimpleFoam` only by a few details. First of all, it can deal with moving meshes (as the `DyM` part of the name suggests) and therefore a step to update the mesh is included in the

solver routine. Then, after the `createFields.H` file is included in the main function file, also the `createIbMasks.H` file is included to create the `gamma` and `sGamma` fields that define the location of the IB surface. The last difference is represented by the Courant number calculation. In the `pimpleDyMIbFoam` solver, a IBM dedicated formulation is implemented. The pressure and velocity equations and the steps taken to solve them remain identical to those of the standard `pimpleFoam`. The structure tree of the tutorial is presented below.

movingCylinderInChannelIco tutorial structure

```
movingCylinderInChannelIco
|-- 0_org
|   |-- p
|   |__ U
|-- Allclean
|-- Allrun
|-- constant
|   |-- dynamicMeshDict
|   |-- polyMesh
|   |   |__ blockMeshDict
|   |-- RASProperties
|   |-- transportProperties
|   |-- triSurface
|   |   |-- ibCylinder.ftr
|   |   |__ ibCylinder.stl
|   |__turbulenceProperties
|-- save
|   |-- blockMeshDict
|   |__ boundary
|__ system
    |-- controlDict
    |-- decomposeParDict
    |-- fvSchemes
    |-- fvSolution
    |__ mapFieldsDict
```

Most of these files have been already discussed for the `olaFlow` tutorial. Therefore, the focus will be on those files including inputs specific to the IBM. First of all, the `constant` folder now contains two surfaces, i.e. `ibCylinder.ftr` and `ibCylinder.stl`. The stereolithography (STL) surface is in principle not needed but it is used to generate the `OpenFOAM` triangulated format (FTR) surface used by the IB library. It is also possible to notice that now the `blockMeshDict` is inside the `constant\polyMesh` folder. This is one of the many small differences between `OpenFOAM` and `foam-extend`. The `save` folder contains a `boundary` file used to correct the one generated by `blockMesh` to account for the IB. The two versions are presented below.

boundary file before correction

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       polyBoundaryMesh;
    location    "constant/polyMesh";
    object      boundary;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

5
(
    in
    {
        type            patch;
        nFaces          25;
        startFace       3650;
    }
...
)
```

boundary file after correction

```
...
6
(
    ibCylinder
    {
        type            immersedBoundary;
        nFaces          0;
        startFace       3650;

        internalFlow    no;
        isWall          yes;
    }
    in
    {
        type            patch;
        nFaces          25;
        startFace       3650;
    }
...
)
```

The added boundary is constructed as follows:

- The name of the boundary must be equal to that of the FTR file.
- The `nFaces` value is always 0 while the `startFace` value should be the same as that of the next boundary.
- The switch `interalFlow` determines if the flow has to be solved inside or outside of the IB. If this is equal to "yes", then the IB is a domain boundary.
- The switch `isWall` determines whether the IB is a wall patch (no-slip boundary condition) or not. This is set as "yes" in all the tutorials of `foam-extend 4.1`.

Finally, the IB is included in the boundary and initial conditions files. Below the `U` file is shown.

Initial and boundary velocity condition file

```
FoamFile
{
version     2.0;
format      ascii;
class       volVectorField;
object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform (1 0 0);

boundaryField
{
    ibCylinder
    {
        type movingImmersedBoundaryVelocity;
        patchType immersedBoundary;
        setDeadValue   yes;
        deadValue      (0 0 0);
        value uniform (0 0 0);
    }
...
}
```

Once again, the name of the boundary is always the same as the FTR file. The `type` of boundary condition is self-explanatory and the `patchType` is obviously `immersedBoundary`. If the switch

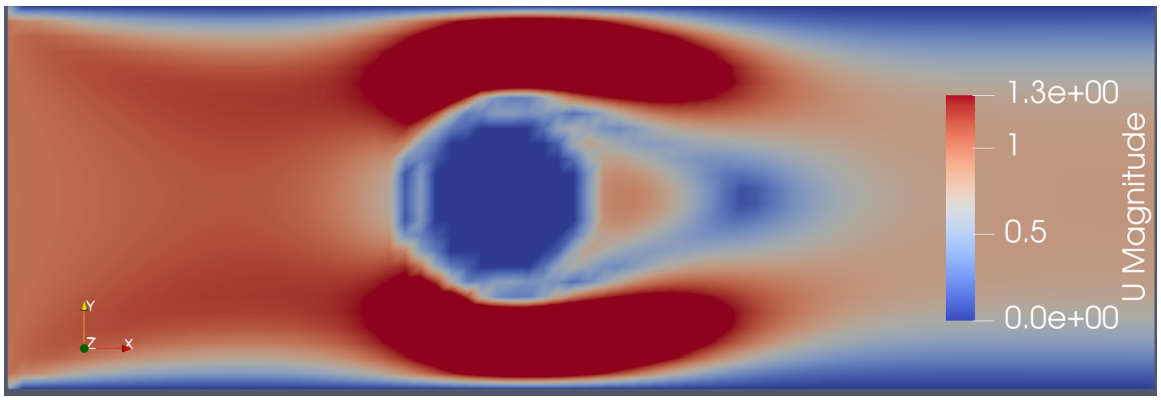
`setDeadValue` is set to "yes", then the velocity value of the dead cells is set to `deadValue`, (000) in this case. The `value` entry is the same as for `OpenFOAM`. The pressure boundary condition at the IB follows the same structure. The only difference is that the `type` entry is set as `zeroGradientIb`.

Finally, to run the tutorial the following lines should be copied in a terminal window in which the `foam-extend 4.1` installation has been sourced. The second line is used to correct the `boundary` file in the `polyMesh` folder.

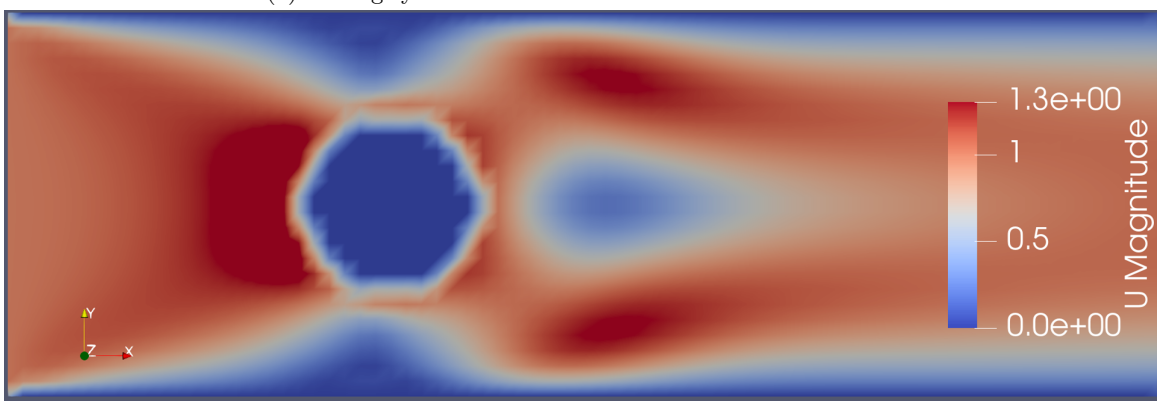
Run script

```
$ blockMesh > log.blockMesh
$ cp save/boundary constant/polyMesh/
$ cp -fr 0_org 0
$ potentialFoam > log.potentialFoam
$ writeIbMasks > log.writeIbMasks
$ pimpleDyMIbFoam > log.pimpleDyMIbFoam
```

To facilitate converge of the `pimpleDyMIbFoam`, the case is first initialized with a potential flow solver. After the initialization, the `writeIbMasks` utility writes the scalar field `gamma` which locates the IB mask. Finally, the simulation can be run with the command `pimpleDyMIbFoam`. To visualise the results, one can type `paraFoam` in a terminal window in which the `OpenFOAM` installation has been sourced. Figures 3.4a and 3.4b presents the results for the velocity field at t=4.0s and t=5.0s respectively.



(a) movingCylinderInChannelIco tutorial results at t=4.0s



(b) movingCylinderInChannelIco tutorial results at t=5.0s

Figure 3.4: movingCylinderInChannelIco tutorial results showing the velocity field

Chapter 4

The porousPimpleIbFoam

4.1 Implementation

In this section, the steps required to modify the solver `pimpleFoam` of `OpenFOAM v2006` into the new `porousPimpleIbFoam` are described. The new solver is able to deal with porous structures using a continuous forcing approach. Moreover, the porosity formulation implemented in the new solver differs from the standard porous region of `OpenFOAM` as it keeps the porosity inside the differential operators, see Eq. (3.5). This step is important whenever multiple porous layers with different characteristics are present as it ensures that fluxes across the interfaces are accurately computed [9]. In principle, this is also important whenever the porosity varies within a single layer. However, currently, it is only possible to define a single constant value of the porosity per layer. This formulation is the same as that implemented in `olaFlow` with the difference that this is derived for a single-phase flow.

The continuous forcing approach is preferred over the discrete one because of its simplicity and lack of mass conservation problems. Moreover, the discrete approach (in its ghost-cell variant) implemented in `foam-extend` does not fit well with the purpose of simulating a porous object. In fact, for such a body, the flow needs to be resolved in the entire domain and, therefore, there are no dead cells (solid cells) where the flow is not resolved. Moreover, there is not a boundary condition to be imposed at the IB but rather an interface condition. Finally, the formulation of the porous drag force proposed by del Jesus et al. [8] works well with the approach of a forcing term included in the governing equations before the discretization step. Despite the lack of a boundary, the interface between fluid and porous body will be still called IB hereafter.

Before diving into the implementation, a brief discussion over the value of porosity in the cells at the IB is necessary. In Section 3.1, it was shown how the porosity field is constructed in the `olaFlow` solver. Given a mesh, the user defines a porous region and assigns the porosity value in the specified region. Elsewhere, the porosity is set to 1 (complete permeability) and the porous drag force goes to zero. However, this approach works fine only for a body-fitted mesh. In the case of an IB approach instead, it is necessary to treat the cut cells at the IB in a special manner. Several approaches can be taken here, for this implementation it was decided to use a VOF-like approach. The idea is the same as that investigated by Fadlun et al. [19] and briefly described in Section 2.2. Since the porosity outside of the porous medium is always equal to 1, this can be advantageously redefined as follows

$$n_{\text{corr}} = \gamma + n\,(1-\gamma)\,, \tag{4.1}$$

where n is the porosity and γ is the VOF-like field defined as PVF$= V_{\text{subCell}}/V_{\text{cell}}$, where PVF stands for Porous Volume Fraction. Here, V_{cell} is the volume of the cell and V_{subCell} indicates the volume of the piece of cut cell internal to the IB. Thus, inside of the immersed body $\gamma = 0$ and $n_{\text{corr}} = n$. At the IB instead, $0 < \gamma < 1$ and the porosity value is corrected accordingly. Elsewhere, the γ field is set to 1 and thus $n_{\text{corr}} = \gamma = 1$, perfect permeability. The advantage of this formulation is that

there is no need to define two separate fields for the porosity, for instance, n_f and n_p, as it is done instead for multi-phase simulations.

The first step of the implementation consists of copying the standard `pimpleFoam` solver of `OpenFOAM` into the `$WM_PROJECT_USER_DIR` directory. This can be done by typing these two commands in a terminal window. Remember to source the `OpenFOAM` installation first.

```
$ foam
$ cp -r --parents applications/solvers/incompressible/pimpleFoam/ $WM_PROJECT_USER_DIR
```

Compared to `pimpleFoam`, the `porousPimpleIbFoam` solver will require a `createPorosity.H` file, to define the porosity fields and a `createPorousIbMask.H` file, where the IB mask is constructed. Then, the `UEqn.H` file needs to be modified to accommodate the porous drag terms. The final folder tree is presented below.

porousPimpleIbFoam solver tree

```
porousPimpleIbFoam/
|-- correctPhi.H
|-- createFields.H
|-- createPorosity.H
|-- createPorousIbMask.H
|-- Make
|   |-- files
|   |-- options
|-- pEqn.H
|-- porousPimpleIbFoam.C
|-- UEqn.H
```

First of all, the IB mask needs to be generated from a surface file, e.g. an STL or FTR file. This file needs to be provided by the user and therefore, a dictionary is created and information is read from it. Two user inputs are read from the `porousIbMaskDict`, the surface file name and the coordinates of a point located outside of the immersed body (see `surfaceSets::getSurfaceSets` function below).

Dictionary, createPorousIbMask.H file

```
// Define dictionary
IOdictionary porousIbMaskDict
(
    IOobject
    (
        "porousIbMaskDict",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    )
);

// Read dictionary
fileName surfName(porousIbMaskDict.get<fileName>("surface"));
surfName.expand();
pointField outsidePts(porousIbMaskDict.lookup("outsidePoints"));
const label nCutLayers(porousIbMaskDict.getOrDefault<label>("nCutLayers", 0));
```

Given the surface, the IB can be identified and the IB mask created. The first step consists of identifying the status of each mesh cell. These are divided into `inside`, `outside` and `cut` cells depending on their location with respect to the IB.

Cells identification, createPorousIbMask.H file

```
// Surface.
const triSurface surf(surfName);

// Search engine on surface
```

```
const triSurfaceSearch querySurf(surf);

// Divide the cells according to status compared to surface. Constructs sets:
// - cutCells : all cells cut by surface
// - inside   : all cells inside surface
// - outside  :   ,,      outside ,,

cellSet inside(mesh, "inside", mesh.nCells()/10);
cellSet outside(mesh, "outside", mesh.nCells()/10);
cellSet cutCells(mesh, "cutCells", mesh.nCells()/10);

surfaceSets::getSurfaceSets
(
    mesh,
    surfName,
    querySurf.surface(),
    querySurf,
    outsidePts,

    nCutLayers,

    inside,
    outside,
    cutCells
);
```

Before computing the PVF of the cut cells, it is necessary to define a scalar field, called *gamma*, used to correct the porosity at the IB. This field has a default value of 1. The exact implementation is shown below.

Scalar field gamma, createPorousIbMask.H file

```
Info<< "Create immersed boundary cell mask" << endl;

volScalarField gamma
(
    IOobject
    (
        "gamma",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar( "gamma", dimless, 1.0 )
);
```

Now, it is finally possible to compute the PVF. For each cut cell, the surface normal at the IB point closest to the cut cell centre and the distance between the cut cell centre and the surface is computed. Then, the `calcSubCell` function is used to compute the PVF of the cut cell and the value is assigned to the `gamma` field. Finally, in all the inside cells `gamma` is set to zero. This concludes the implementation of the IB mask. Once again, the exact code lines are reported below.

PVF computation, createPorousIbMask.H file

```
forAll(cutCells, cellI)//loop over the cells intersecting the object surface
{
    const point& c = ctrs[cutCells.sortedToc()[cellI]];

    const pointIndexHit inter = querySurf.nearest(c, span);

    const label trii = inter.index();
    const vector normal = normals[trii]/mag(normals[trii]); //compute surface normal

    const scalar cutValue = (inter.hitPoint() - c) & (normal); //compute distance to surface

```

```
    cutCell.calcSubCell
    (
        cellI,
        cutValue,
        normal
    );
    gamma[cutCells.sortedToc()[cellI]] = cutCell.VolumeOfFluid(); //assigned field value
}

forAll(inside,cellI)//loop over the cells inside the object surface
{
    gamma[inside.sortedToc()[cellI]] = 0; //assigned field value
}
```

Similarly to what was presented for the `olaFlow` solver, also here there is the need to create a porosity dictionary and define some fields. This is done in the same fashion as presented above. The main difference is represented by the correction of the porosity field. This operation is done using the `gamma` field as follows.

Correction in createPorosity.H file

```
//Correct porosity field to account for porous volume fraction at interface
Info << "\nCorrecting porosity field\n" << endl;
porosity = gamma + (1-gamma)*porosity;
```

In this way, the porosity will have a value of 1 outside the body (complete permeability), of n (user-defined level of permeability) inside the body and between n and 1 at the interface. Finally, the `UEqn.H` file is modified to accommodate the forcing term. This is done in an `olaFlow` like fashion. The modified momentum equation is presented below. The reader is referred back to Section 3.1 for all the details.

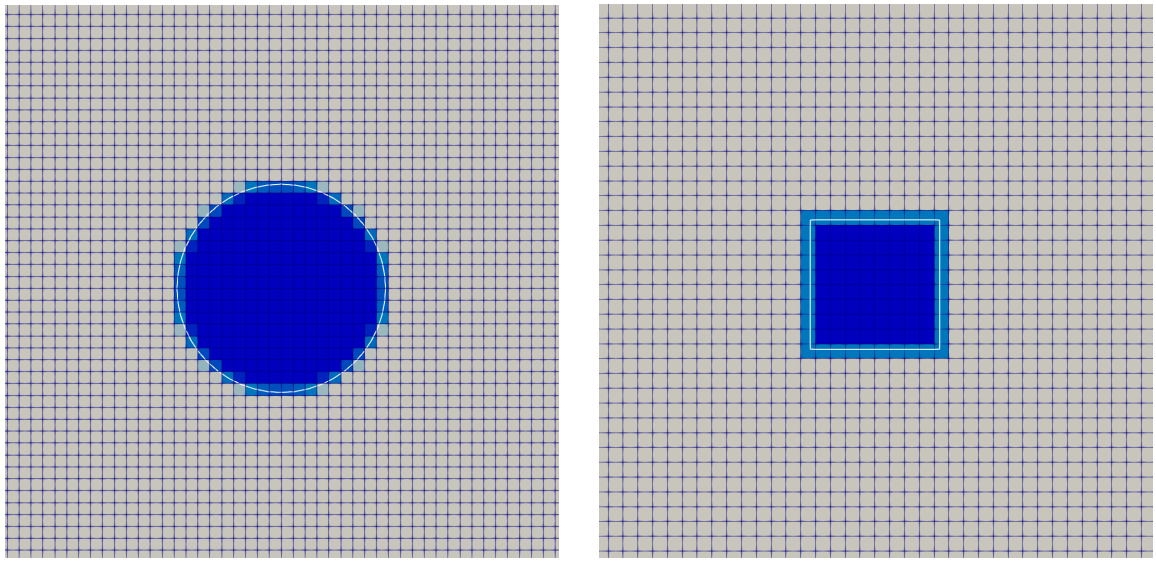
Momentum equation porousPimpleIbFoam

```
// Solve the Momentum equation
MRF.correctBoundaryVelocity(U);

surfaceScalarField nuEff
(
    "nuEff",
    fvc::interpolate(turbulence->nuEff())
);

tmp<fvVectorMatrix> tUEqn
(
   (1.0+cPorField)/porosity*fvm::ddt(U)
   + (1.0+cPorField)/porosity*MRF.DDt(U)
   + 1.0/porosity*fvm::div(phi/porosityF, U)
   - fvm::laplacian(nuEff/porosityF,U)
   - 1.0/porosity*(fvc::grad(U) & fvc::grad(nuEff))
   // Closure Terms
   + aPorField*pow(1.0-porosity,3)/pow(porosity,3)*turbulence->nu()/pow(D50Field,2)*U
   + bPorField*(1.0-porosity)/pow(porosity,3)/D50Field*mag(U)*U*
   // Transient formulation
   (1.0 + useTransMask * 7.5 / KCPorField)
  ==
   fvOptions(U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
solve(UEqn == -fvc::grad(p));

```

```
35 fvOptions.correct(U);
36 }
```

4.2 Verification and results

First of all, the correct computation of the PVF and the γ field defining the IB mask is verified for two different body shapes. Figure 4.1 shows the results of the algorithm for the cases of a circular, Figure 4.1a, and squared, Figure 4.1b, body. The white lines represent the sharp IB. In Figure 4.1b,



(a) Circular body (b) Squared body

Figure 4.1: Gamma field computed for different body shapes

it can be noticed that the volume fraction at the corner cells is not computed correctly. This shows that the algorithm used to compute the PVF cannot handle sharp features. This problem is caused by the fact that a single IB point is used to compute the distance to the nearest cell centre. Further development is necessary to solve this issue. For the circular shape, this problem is not relevant. Here, the algorithm works as expected and the γ field is properly computed in all the cut cells. However, the precision of the algorithm should be investigated further since this can spoil the order of accuracy of the overall scheme. For the square object, the calculation of the PVF on the sides is exact, however, on the corner cells, the computed PVF is about 66% larger than it should be, in this case. Regarding the circular body, it is hard to say what the accuracy is. In fact, in this case, the accuracy of the PVF will depend on the mesh size. Since the curvilinear shape is locally approximated in a linear fashion, the finer the mesh the more accurate is the PVF and vice versa. Further investigation is required to establish a relation between mesh size and the accuracy of the algorithm. To generate this field, the user needs to provide a dictionary, as explained previously. An example of this dictionary is shown below.

porousIbMask dictionary

```
1 FoamFile
2 {
3     version     2.0;
4     format      ascii;
5     class       dictionary;
6     location    "constant";
7     object      porousIbMaskDict;
8 }
```

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

surface         "<constant>/triSurface/ibCylinder.stl";
outsidePoints   ((0 -1 0));

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

The next step of the validation consists of comparing the results obtained with the IBM against those of a body-fitted mesh approach. For the body-fitted simulations, a modified version of the `pimpleFoam` solver including the porous model of `olaFlow` is used. The investigated case is that of a porous cylinder in a constant speed flow. Two different inflow conditions are considered, namely Re=40 and Re=100. Simulations are performed in 2D. In the first run, it is expected to find a fixed pair of symmetric vortices behind the cylinder. At Re=100 instead, vortex shedding is expected. Therefore, for the first case, the results will be compared in time while for the second case, these will be first averaged over 10 vortex shedding periods. Figure 4.2 presents the geometric dimensions of the domain used in both simulations. At the sides, a `symmetryPlane` condition is imposed, this provides a no-slip boundary condition. At the inlet, the velocity is set with `fixedValue` while for the pressure a `zeroGradient` boundary conditions is used. At the outlet, it is exactly the opposite. For the `porosityIndex` field, a `zeroGradient` boundary condition is imposed at both inlet and outlet while a `symmetryPlane` condition is imposed on the sides. Usually, a `zeroGradient` condition would be imposed on every boundary for this field, however, when a patch is defined as `symmetryPlane`, `OpenFOAM` only allows that boundary condition for any field. This does not affect the results since the two boundary conditions are almost identical. The complete files including boundary and initial conditions for pressure, velocity and porous index are provided in the Appendix A. A porosity of 10%

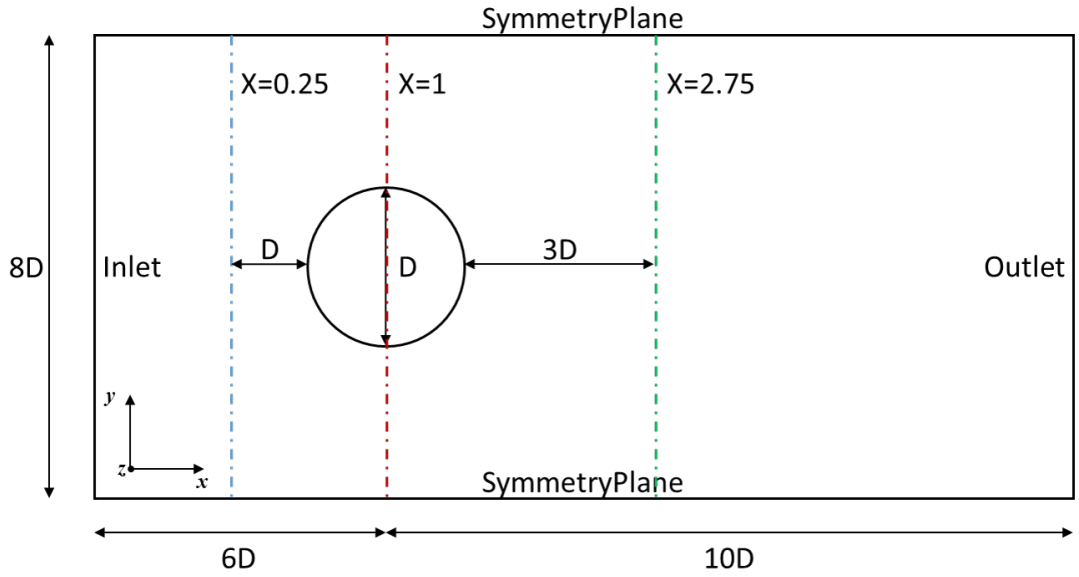
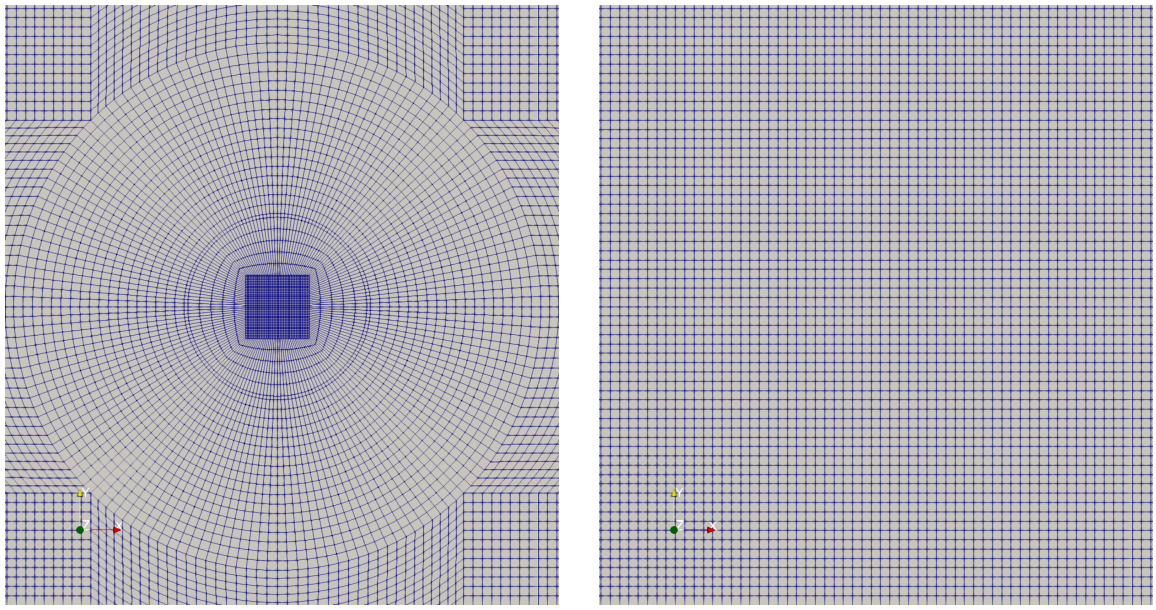


Figure 4.2: Geometric dimensions of the simulation domain including boundary information and probes locations

is chosen for these simulations. An adaptive time step with a maximum Courant number condition of 0.1 is employed to ensure stability. For both simulations, the transport model is set to `laminar`, i.e. no turbulence model is used. All the information regarding the settings of the `fvSchemes` and `fvSolution` files are included in the Appendix A. One should note that the geometric dimensions choices and the simulations settings are intended for verification and comparison only. This setup should not be used to obtain accurate physical results.

Firstly, the lower Re case is investigated. The meshes used for the two cases are as much as possible identical. The two meshes are presented in Figure 4.3. The body-fitted mesh is slightly

more refined near the body for meshing reasons. This results into around 55000 hexahedral cells for the conformal mesh and 50000 hexahedral cells for the IBM one. The results of the two approaches



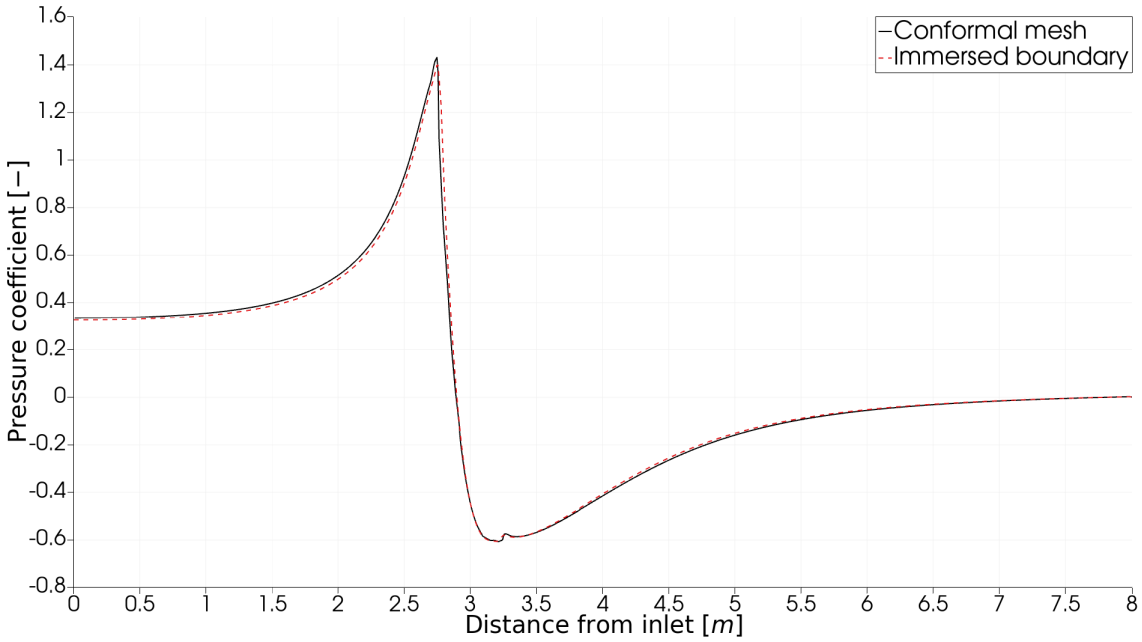
(a) Close-up of the mesh around the cylinder for the conformal case

(b) Close-up of the mesh around the cylinder for the IBM case

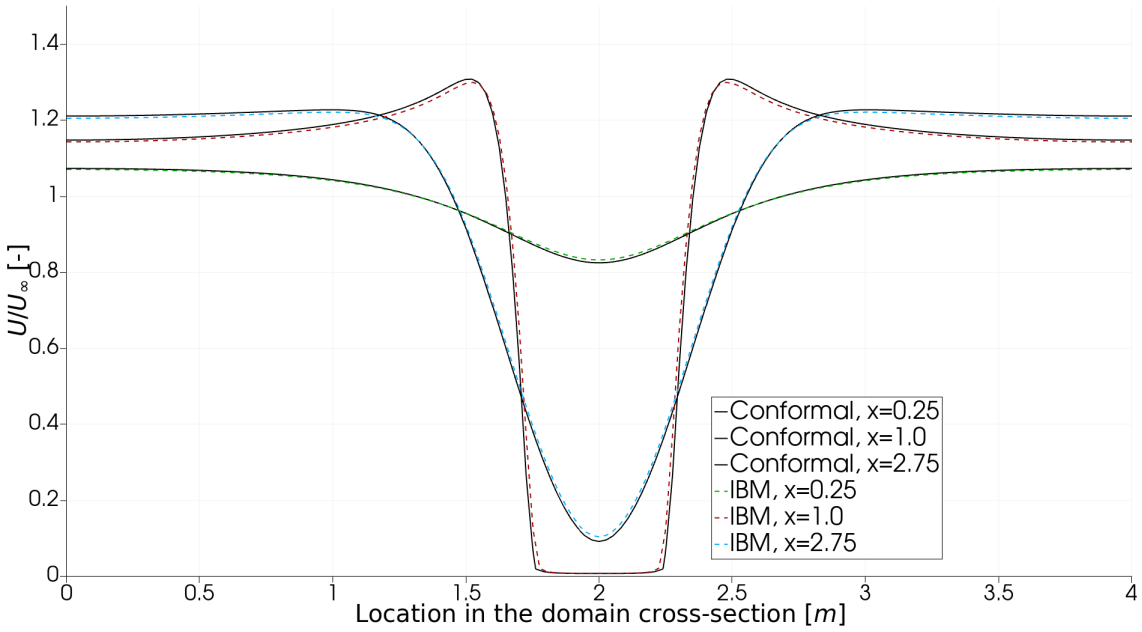
Figure 4.3: Comparison between conformal mesh and IBM mesh

are compared in both quantitative and qualitative manner. Figure 4.4 presents a quantitative comparison in terms of both pressure and velocity in different regions of the domain. Figure 4.4a presents the pressure coefficient along the central line of the domain. It can be identified an increase in pressure in front of the cylinder, a sudden drop at the boundary, a mostly constant value inside the porous region and then a pressure recovery behind the cylinder. The output of the two approaches matches very well for this check. Figure 4.4b presents the velocity profile at three different locations in the domain. The locations correspond to one diameter in front of the cylinder ($x = 0.25$), the middle of the cylinder ($x = 1$) and three diameters behind the cylinder ($x = 2.75$). The results match the expectations, i.e. moving along the y-direction, the velocity increases approaching the cylinder, then it drops, reaching a minimum in the centre, and then the shape is mirrored. Moving from downwind locations to upwind locations, the velocity drop increases until it reaches a minimum at the cylinder and then it slowly decreases again. Once again, the output of the two approaches presents an almost one-to-one match. In more qualitative terms, the results are also compared by plotting half IB domain and half conformal mesh domain side-by-side. This is done in Figure 4.5 for the normalized velocity field and a few pressure contour lines. The two halves are almost identical and only the presence of the contour lines allow spotting some small differences. Overall, the results are satisfactory. A second qualitative check is presented in Figure 4.6. Here, the velocity streamlines are computed on the two halves to construct a sort of complete domain solution. Once again, the results look very promising and the two halves match very well.

The same comparison metrics are also used for the case at Re=100. Since this inflow condition leads to vortex shedding, the compared quantities have been averaged in time over 10 vortex shedding periods. Figure 4.7 presents the quantitative comparison for pressure coefficient along the central line and velocity profiles at different locations. The analysed locations are the same as for the lower Re case. The results are very similar to those presented above. The main difference can be identified in a lower pressure and velocity recovery in the wake. Very good agreement is found between the two approaches also for this case. Finally, Figure 4.8 and 4.9 present the qualitative comparison. In Figure 4.8, the normalized time-averaged velocity field is shown together with some pressure contour



(a) Pressure coefficient along the central line of the domain



(b) Normalized velocity profiles at different locations

Figure 4.4: Comparison between conformal mesh approach and IBM at Re=40

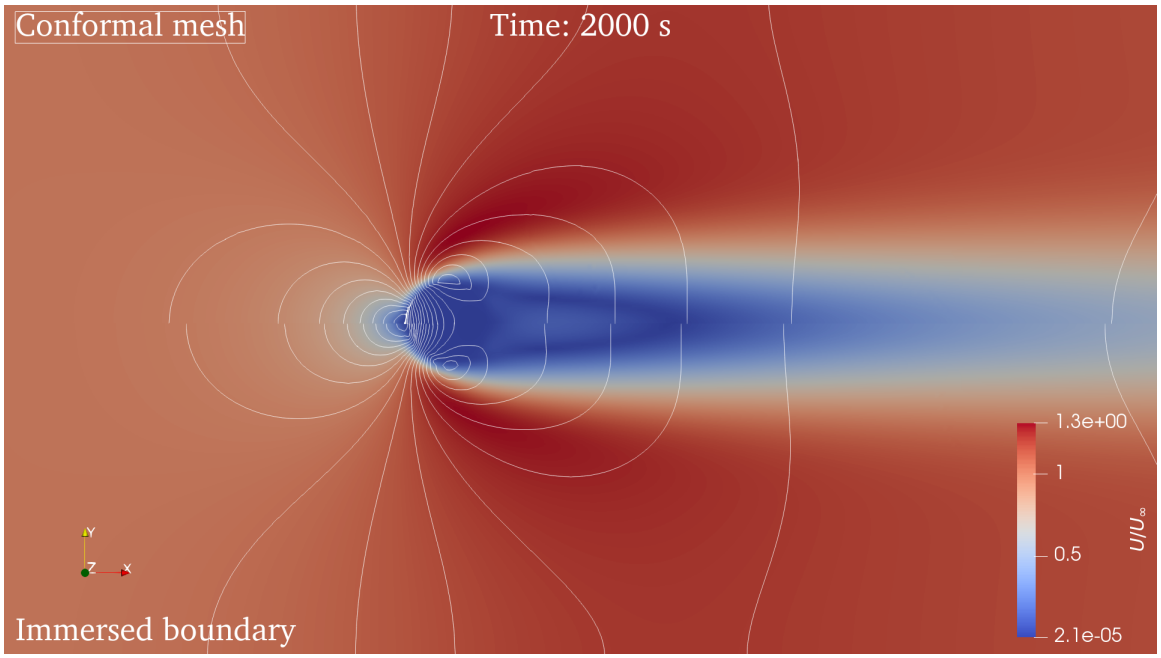


Figure 4.5: Normalized velocity field and pressure contours, comparison between conformal mesh and IBM at Re=40

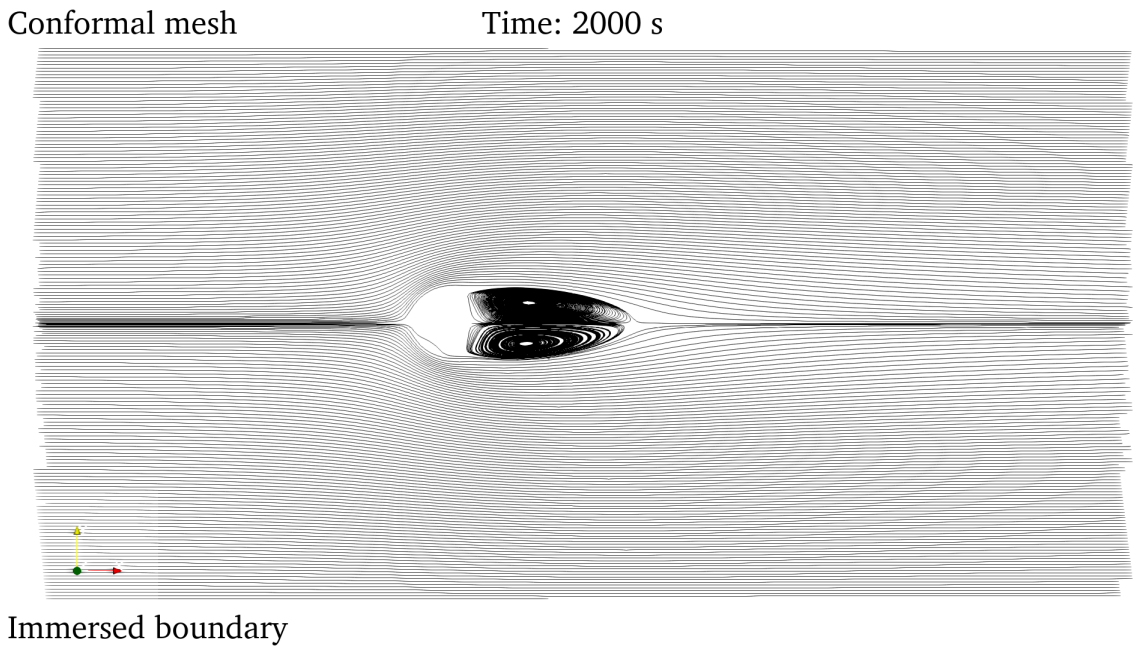
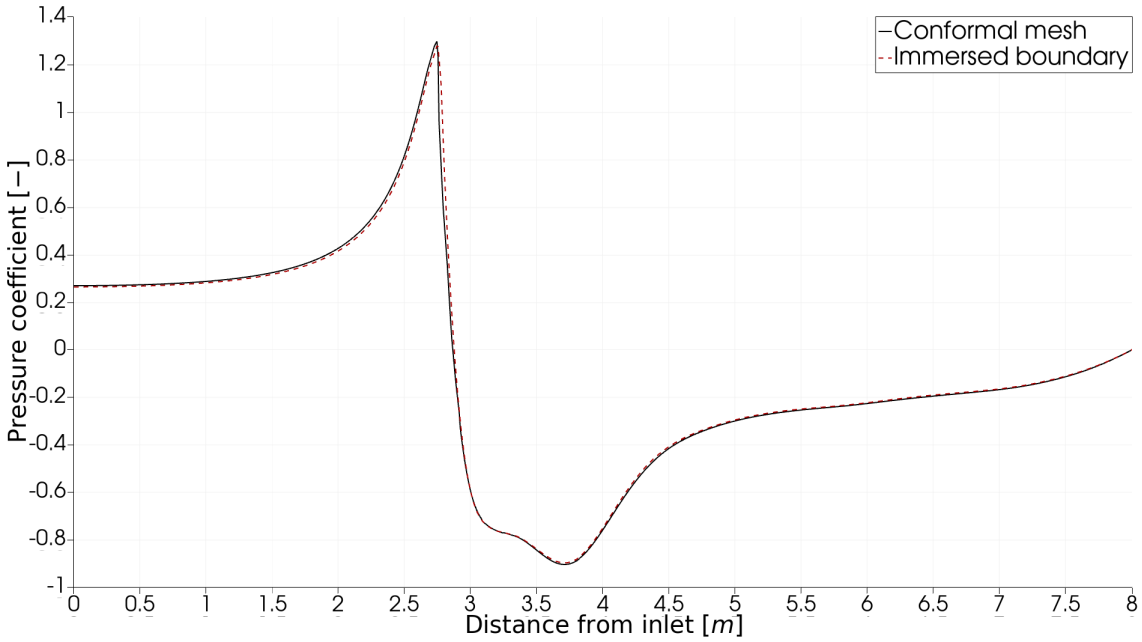
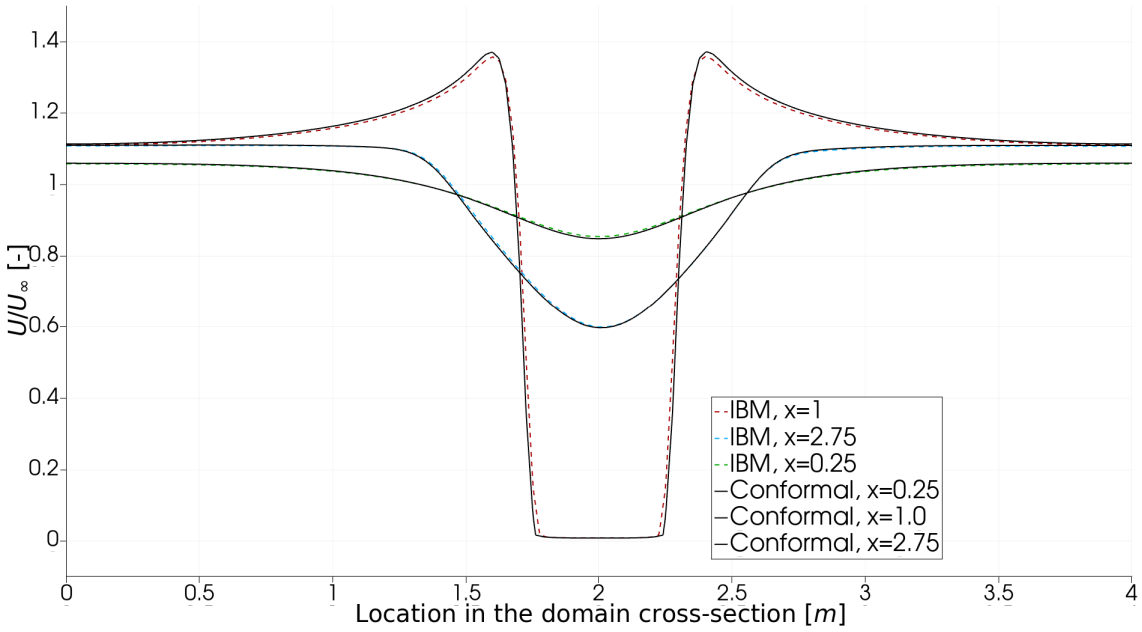


Figure 4.6: Velocity streamlines, comparison between conformal mesh and IBM at Re=40



(a) Pressure coefficient along the central line of the domain



(b) Normalized velocity profiles at different locations

Figure 4.7: Comparison between conformal mesh approach and IBM at Re=100

lines. The two models match very well and they predict the same separation location for the flow. Figure 4.9 attempts to recreate the complete vortex street behind the porous cylinder by merging the two half domains from the different simulations. Differently from the others, these results are not time-averaged. The comparison shows how the two solutions agree very well although there is a time lag between the two.

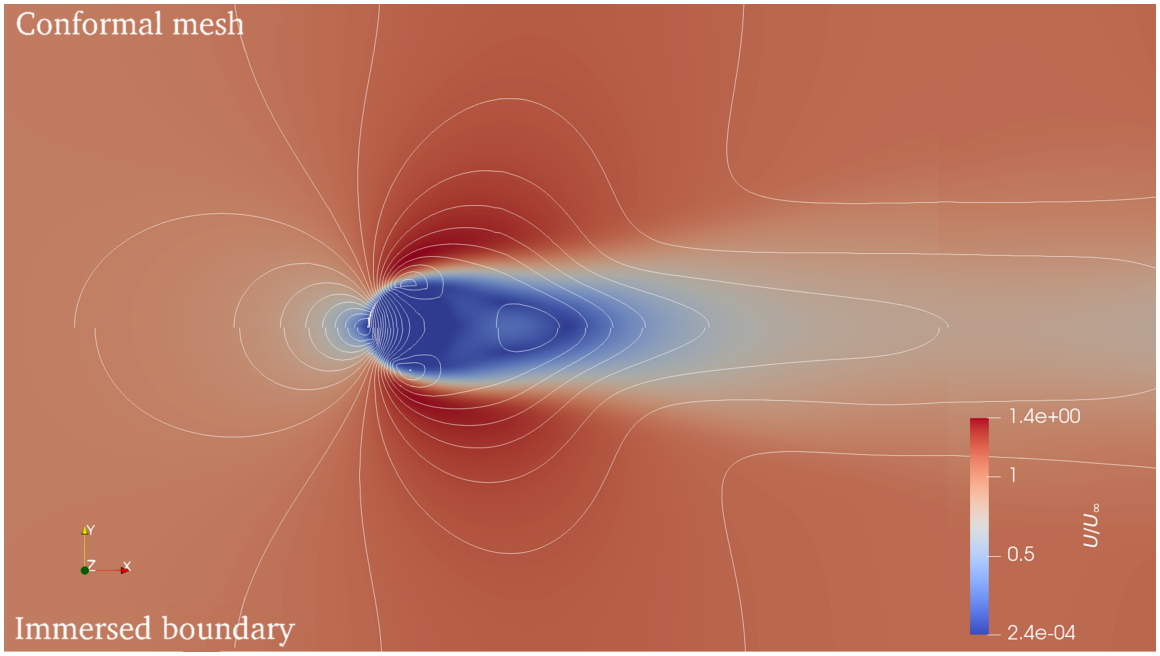


Figure 4.8: Normalized velocity field and pressure contours, comparison between conformal mesh and IBM at Re=100

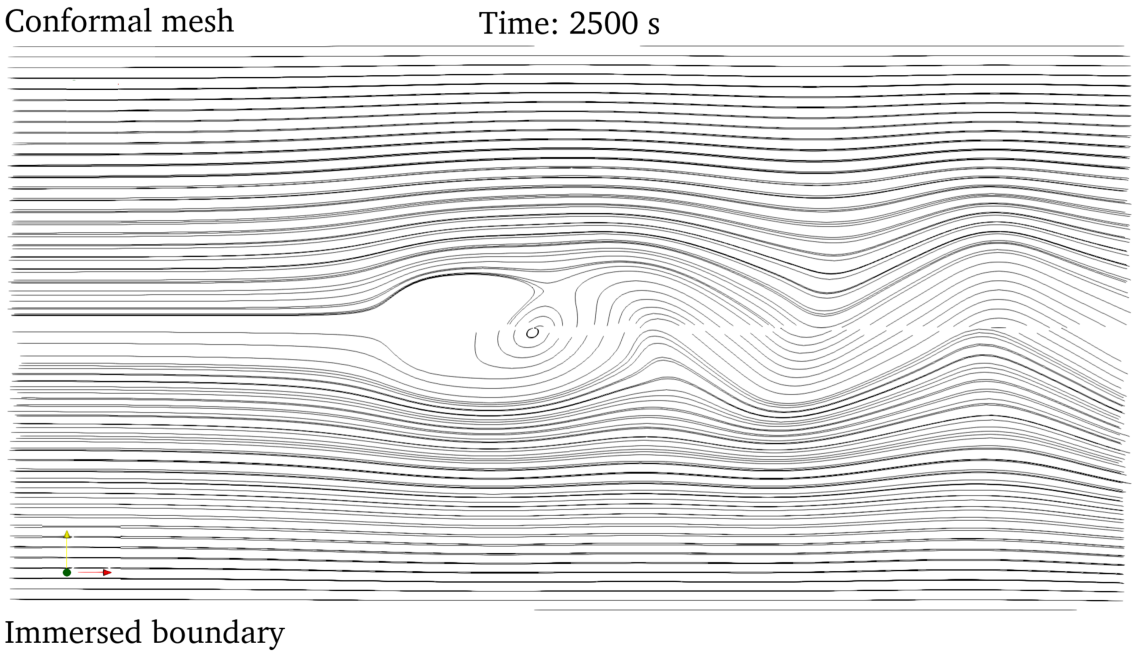


Figure 4.9: Velocity streamlines, comparison between conformal mesh and IBM at Re=100

Overall, for the investigated cases, it can be concluded that the `porousPimpleIbFoam` solver is very well capable of resolving the flow in and around an immersed porous body at a relatively low Re. However, a broader validation is still required to investigate the accuracy of the solver at higher Re. For the second case investigated, one last comparison is performed regarding the computational cost of the two simulations. Table 4.1 shows the computational performance of the two solvers over a simulation time of 100 s. The same machine and the same number of cores are used for the two simulations. From the data retrieved, it can be seen how the IBM is almost twice as fast as the conformal mesh approach for the investigated case. On top of this, the time spent to generate a conformal mesh should also be included. However, this is hardly quantifiable and strongly dependent on the user abilities.

Table 4.1: Computational performance over 100 s of simulation

Case	CPU	Cores	Clock time [s]	Speedup
Conformal mesh	Intel Core i7-8850H 2.60GHz	4	86.4	-
Immersed body	Intel Core i7-8850H 2.60GHz	4	45.88	1.88

Finally, as previously discussed, one of the reasons to prefer the continuous forcing approach over the discrete one is its lack of mass conservation problems. The issue is easily identifiable because it generally results in small nonphysical oscillations or even large spikes in the pressure. Therefore, to prove that this is not an issue for the developed solver, the pressure evolution in time in front of the body is presented in Figure 4.10. This result is obtained by placing a probe adjacent to the IB surface, right in the middle of the front surface of the porous cylinder. This is done for the case at Re=100 only. The pressure coefficient in Figure 4.10 oscillates because of the vortex shedding. Neither spikes nor nonphysical oscillations can be detected. Therefore, it is concluded that the implemented continuous forcing IB solver does not suffer from mass conservation problems for the investigated cases. However, the solver should also be tested for the case of a moving object to make sure that issues do not arise in that case. In fact, for a moving object, the cells close to the IB surface might change state (from inside cells to cut cells for instance) from one time step to the next. This is usually the most critical situation to test for.

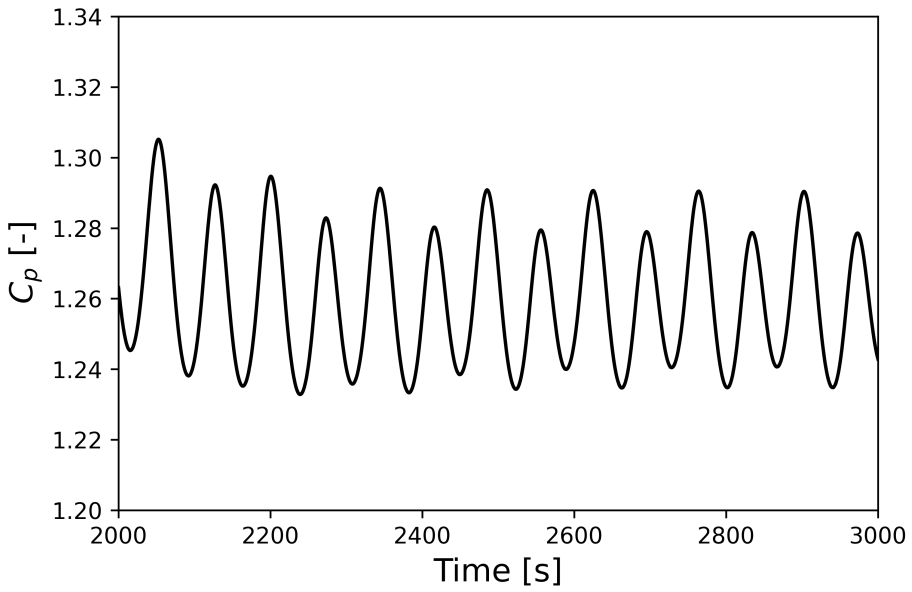


Figure 4.10: Pressure coefficient profile over time for the case Re=100

Chapter 5

Tutorial setup for the porousPimpleIbFoam solver

This chapter briefly presents the steps required to create and run a simple tutorial to test the use of the `porousPimpleIbFoam` solver. At this stage, the reader is expected to be familiar with most of the `OpenFOAM` terminology. The tutorial presented in this chapter can be downloaded here, together with the `porousPimpleIbFoam` solver. The tutorial simulates an immersed porous cylinder in a constant 2D flow at $Re = 40$. A schematic of the domain is presented in Figure 5.1 including the geometric dimensions and some boundary information. The structure tree of the case looks like the following. The `0.org` folder contains initial and boundary conditions for pressure, porosity index and velocity. The `constant` folder contains information regarding the mesh, the IB mask, the porous region properties and fluid transport properties. The `system` folder contains the control dictionary and the finite volume settings to solve the governing equations. Also, it contains the dictionaries used by pre-/post-processing utilities, e.g. `blockMeshDict` and `setFieldsDict`. Finally, an `Allrun` script is provided together with an `Allclean` one. These two scripts run and clean the case.

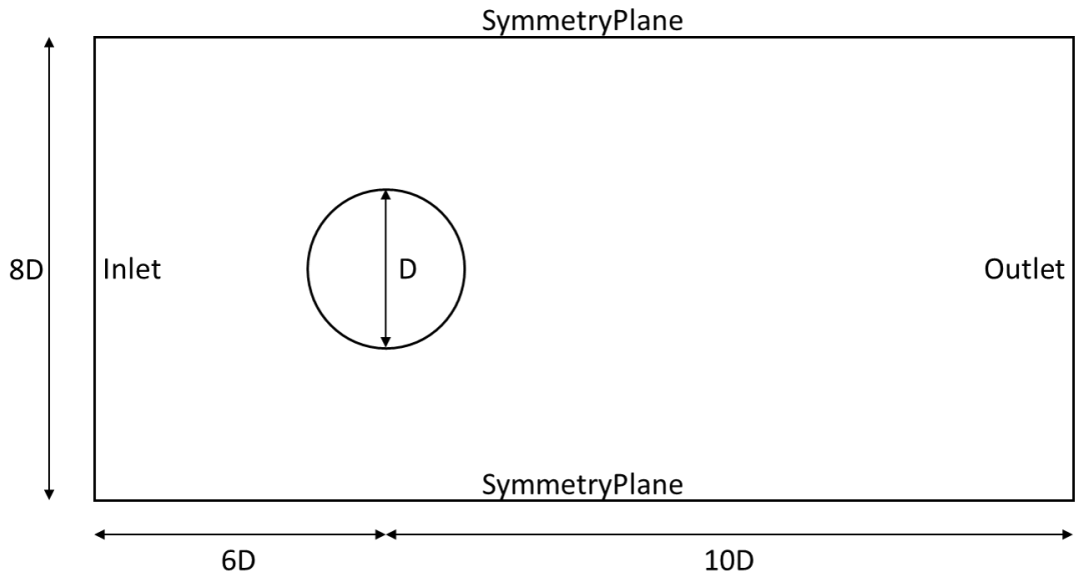


Figure 5.1: Geometric dimensions of the simulation domain including boundary information

porousPimpleIbFoam tutorial tree

```
porousPimpleIbFoamCase/
|--  0.org
|   |-- p
|   |-- porosityIndex
|   |__ U
|-- Allclean
|-- Allrun
|-- constant
|   |-- polyMesh
|   |-- porosityDict
|   |-- porousIbMaskDict
|   |-- transportProperties
|   |-- triSurface
|       |__ ibCylinder.stl
|   |__ turbulenceProperties
|-- system
    |-- blockMeshDict
    |-- controlDict
    |-- decomposeParDict
    |-- fvSchemes
    |-- fvSolution
    |__ setFieldsDict
```

5.1 Mesh generation

The first step of the case setup consists of generating a mesh. For the case of an IBM solver, the base mesh is very simple, therefore, this will be generated with the `blockMesh` utility. Since the mesh generation was not discussed previously in this report, the `blockMeshDict` file is here presented in detail. As the name suggests, `blockMesh` works with blocks objects. Therefore, the first thing to do is to define the vertices of the blocks composing the domain. The domain used in this tutorial has the same shape as the one presented in Figure 4.2.

Geometry definition, `blockMeshDict` file

```
scale   1;

vertices
(
    (-2 -2 -0.1)
    (6  -2 -0.1)
    (6   2 -0.1)
    (-2  2 -0.1)
    (-2 -2  0.1)
    (6  -2  0.1)
    (6   2  0.1)
    (-2  2  0.1)
);
```

The `scale` entry allows to convert the dimension's units, e.g. from m to mm. The vertices are defined as (x, y, z) points. Then, the blocks (only one in this case) are defined as follow.

Blocks definition, `blockMeshDict` file

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (320 160 1) simpleGrading (1 1 1)
);
```

In this extract from the dictionary, the command used to generate a hexahedral block mesh is shown. The first set of entries are the vertices of the block, each block must have a local coordinate system that is right-handed. More information about this can be found in the User Guide of `OpenFOAM`. The second inline entry defines the number of cells in each direction. Since this tutorial simulates a

2D case, the third entry is equal to 1. OpenFOAM always uses 3D meshes also for 2D cases, however, in this case, the third dimension must be described by a single cell. The last part of the command, simpleGrading, defines the distribution of the cells. Once again, the reader is referred to the User Guide for more information about the use of simpleGrading and its alternatives. Finally, the boundary patches can be defined. To define a patch, the user needs to provide a name, e.g. 'inlet' or 'top', a patch type, such as 'patch' or 'wall', and the list of vertices composing the patch. The proper implementation is shown below for a few patches.

Boundaries definition, blockMeshDict file

```
boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }

    ...

    top
    {
        type symmetryPlane;
        faces
        (
            (3 7 6 2)
        );
    }

    ...

    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

Here, some entries have been omitted for clarity. In certain cases, the type entry defines already the boundary condition, this is the case for the symmetryPlane and empty types. The patch type empty is used to let the solver know that the flow won't be resolved in that direction. Once the blockMeshDict is ready, the mesh can be generated by typing the command blockMesh in a terminal window. Always remember to source the OpenFOAM installation first. The resulting mesh should look like the one presented in Figure 4.3b.

5.2 Porosity setup

The next step consists of defining an IB porous region in the domain. To achieve this, three dictionaries are necessary. The first dictionary is called porosityDict, it is located in the constant folder and it was already discussed in Section 3.1.1. The second dictionary is called setFieldsDict, it is located in the system folder and it is used to assign the values of the porosityIndex field. This dictionary was already presented in Section 3.1.1, however, in this tutorial, it is written differently. Since for the IB solver an STL file is required, here we can take advantage of that. Below, the setFieldsDict for this tutorial is presented. First of all, the default value for the entire domain is defined. The default value for the porosityIndex is 0 since this switch is used only to identify the

porous cells. Then, the `porosityIndex` is set to 1 in the porous region. This region can be identified in many ways, here we take advantage of the STL file to automatically select the correct cells.

`setFieldsDict` file

```
defaultFieldValues
(
    volScalarFieldValue porosityIndex 0
);

regions
(
    surfaceToCell
    {
        file            "<constant>/triSurface/ibCylinder.stl";
        outsidePoints   ((0 -1 0));
        includeCut      false;
        includeInside   true;
        includeOutside  false;
        nearDistance    0.01;
        scale           1.0;
        curvature       -100;
        useSurfaceOrientation true;
        fieldValues
        (
            volScalarFieldValue porosityIndex 1
        );
    }
);
```

The `SurfaceToCell` command allows using an STL file to identify the cells which are inside the surface, outside of it or cut by it. The most important entries are the `include*` ones. These flags tell the utility to include or not certain cells in the selection. The `nearDistance` entry defines the size of the searching box around a surface point, it can improve the searching results. The `scale` defines a surface scaling factor, it is 1 by default. The `curvature` input defines the maximum local surface curvature. It is a sort of switch to include or exclude certain cells. `useSurfaceOrientation` allows using the surface orientation from the STL file. If this flag is active, then the `includeCut` flag must be deactivated. This however does not prevent cut cells from being included. This is the case for this tutorial. Generally speaking, the user has to tune these entries and find what works best for the specific case. It is important to underline that for the —porousPimpleIbFoam— solver to work properly, all the cut cells should be selected at this stage. Otherwise, the porosity in the cut cells won't be corrected properly by the solver. Finally, the last dictionary that needs to be present is the `porousIbMaskDict`. This is located in the `constant` folder, it provides the name of the STL file to the solver together with an outside reference point. This was already discussed in Section 4.2.

5.3 Run and visualize

It is now time to run the simulation. The boundary and initial conditions used are the same as those discussed in Section 4.2 and the `fvSolution` and `fvSchemes` files are provided in the Appendix A. The settings used are standard and any `OpenFOAM` user will find those very familiar. Finally, an adaptive time step with a maximum Courant condition of 0.1 is used. The simulation runs for 2000 s and it outputs results every 100 s. These options are set via the `controlDict` file. An extract of this file is presented below. For brevity, not all the entries are reported. Other than those already mentioned, the `runTimeModifiable` can be seen. When this flag is activated, it allows the user to modify the parameters of this file while the simulation is running. This can be very useful if, for example, a smaller time step is required at the beginning of the simulation. In this case, the condition on the Courant number could be relaxed after a few time steps.

`controlDict` file

```
application     porousPimpleIbFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         2000;

deltaT          0.05;

...

writeInterval   100;

...

runTimeModifiable true;

adjustTimeStep  true;

maxCo           0.1;
```

Finally, the tutorial can be run by typing `./Allrun` in a terminal window. Alternatively, the following command lines can be typed in the terminal. Always make sure to source the `OpenFOAM` installation first.

Run script

```
$ cp -r 0.org 0
$ blockMesh > log.preProcessing
$ setFields >> log.preProcessing
$ decomposePar >> log.preProcessing
$ mpirun -np 4 porousPimpleIbFoam -parallel > log.porousPimpleIbFoam
```

If it is not possible to perform parallel simulations on your machine, then you can skip the `decomposePar` command. Also, the last command should be changed into `porousPimpleIbFoam > log`. On four cores, the tutorial takes around 7 min to finish. In serial, the simulation wall time equals approximately 11 min. Once the simulation is done, the results can be analysed with the visualization software `ParaView` [20]. The results can be opened by typing `paraFoam`, in a terminal window from within the case folder. Alternatively, a file with extension `.foam` can be created, for instance with the command `touch case.foam`, and then this file can be loaded in ParaView. To run ParaView alone, simply type `paraview` in the terminal. In `ParaView`, the velocity field evaluated at the latest time step should look like the one presented in Figure /reffig:resultsTutorial. In case of parallel simulation, please make sure to select the `Decomposed Case` option in the `properties bar` of the `pipeline browser` in `ParaView`.

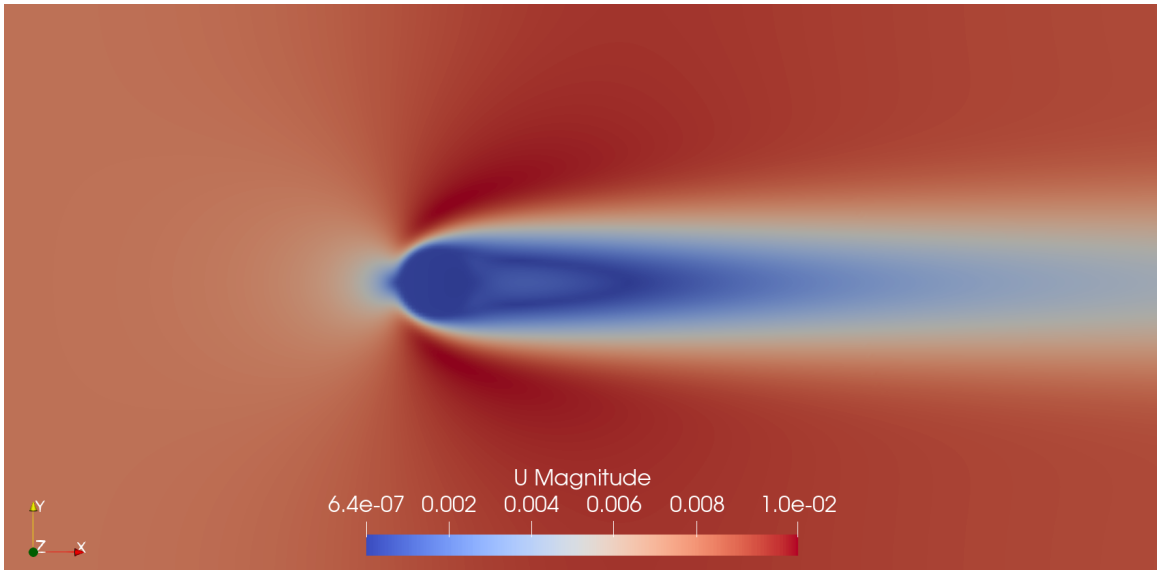


Figure 5.2: Velocity field at t= 2000s

Bibliography

[1] "CFD with OpenSource Software." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/. Accessed: 2020-11-21.

[2] A. Feichtner, E. Mackay, G. Tabor, P. Thies, and L. Johanning, "Comparison of macro-scale porosity implementations for cfd modelling of wave interaction with thin porous structures," *Journal of Marine Science and Engineering*, vol. 9, p. 150, 02 2021.

[3] P. Higuera, J. L. Lara, and I. J. Losada, "Three-dimensional interaction of waves and porous coastal structures using openfoam®. part ii: Application," *Coastal Engineering*, vol. 83, pp. 259–270, 2014.

[4] A. Feichtner, E. Mackay, G. Tabor, P. Thies, L. Johanning, and D. Ning, "Using a porous-media approach for cfd modelling of wave interaction with thin perforated structures," *Journal of Ocean Engineering and Marine Energy*, vol. 7, pp. 1–23, 02 2021.

[5] P. Higuera, "olaflow: CFD for waves [Software].," 2017.

[6] H. Jasak, D. Rigler, and Z. Tukovic, "Design and implementation of immersed boundary method with discrete forcing approach for boundary conditions," *Proc. of the 6th European Conference on Computational Fluid Dynamics*, vol. 1, pp. 5319–5332, 2014.

[7] T.-J. Hsu, T. Sakakiyama, and P.-F. Liu, "A numerical model for wave motions and turbulence flows in front of a composite breakwater," *Coastal Engineering*, vol. 46, no. 1, pp. 25–50, 2002.

[8] M. del Jesus, J. L. Lara, and I. J. Losada, "Three-dimensional interaction of waves and porous coastal structures: Part i: Numerical model formulation," *Coastal Engineering*, vol. 64, pp. 57–72, 2012.

[9] P. Higuera, J. L. Lara, and I. J. Losada, "Three-dimensional interaction of waves and porous coastal structures using openfoam®. part i: Formulation and validation," *Coastal Engineering*, vol. 83, pp. 243–258, 2014.

[10] F. Cimolin and M. Discacciati, "Navier–stokes/forchheimer models for filtration through porous media," *Applied Numerical Mathematics*, vol. 72, pp. 205–224, 2013.

[11] M. R. A. Van Gent, "Wave interaction with permeable coastal structures," in *International Journal of Rock Mechanics and Mining Sciences and Geomechanics Abstracts*, vol. 6, p. 277A, 1995.

[12] C. S. Peskin, "Flow patterns around heart valves: a digital computer method for solving the equations of motion," *IRE transactions on medical electronics*, vol. BME-20, no. 4, pp. 316–317, 1973.

[13] U. Senturk, D. Brunner, H. Jasak, N. Herzog, C. Rowley, and A. Smits, "Benchmark simulations of flow past rigid bodies using an open-source, sharp interface immersed boundary method," *Progress in Computational Fluid Dynamics An International Journal*, vol. 1, 03 2018.

[14] A. Viré, J. Xiang, F. Milthaler, P. Farrell, M. Piggott, J.-P. Latham, D. Pavlidis, and C. Pain, "Modelling of fluid-solid interactions using an adaptive-mesh fluid model coupled with a combined finite-discrete element model," *Ocean Dynamics*, vol. 62, pp. 1487–1501, 12 2012.

[15] M.-C. Lai and C. S. Peskin, "An immersed boundary method with formal second-order accuracy and reduced numerical viscosity," *Journal of Computational Physics*, vol. 160, no. 2, pp. 705–719, 2000.

[16] K. Khadra, P. Angot, S. Parneix, and J. Caltagirone, "Fictitious domain approach for numerical modelling of navier–stokes equations," *International Journal for Numerical Methods in Fluids*, vol. 34, pp. 651–684, 12 2000.

[17] J. Mohd-Yusof, "Combined immersed-boundary/b-spline methods for simulations of flow in complex geometries," *Center for turbulence research annual research briefs*, vol. 161, no. 1, pp. 317–327, 1997.

[18] J. Kim, D. Kim, and H. Choi, "An immersed-boundary finite-volume method for simulations of flow in complex geometries," *Journal of Computational Physics*, vol. 171, no. 1, pp. 132–150, 2001.

[19] E. Fadlun, R. Verzicco, P. Orlandi, and J. Mohd-Yusof, "Combined immersed-boundary finite-difference methods for three-dimensional complex flow simulations," *Journal of Computational Physics*, vol. 161, no. 1, pp. 35–60, 2000.

[20] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *Visualization Handbook*, 01 2005.

Study questions

1. Which is an important difference between the porous models implemented in `olaFlow` and `OpenFOAM`? Why is this difference important?
2. Which type of immersed boundary method is used in the `porousPimpleIbFoam` and why?
3. How is the porosity field corrected at the immersed boundary? And why isn't there a porosity field for the solid region and one for the fluid region?

Appendix A

Solver validation set-up details

A.1 Boundary and initial conditions solver validation

p file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            fixedValue;
        value           $internalField;
    }
    top
    {
        type            symmetryPlane;
    }
    bottom
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type empty;
    }
```

```
}

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

U file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0];

internalField   uniform (0.008 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    outlet
    {
        type            pressureInletOutletVelocity;
        value           $internalField;
    }
    top
    {
        type            symmetryPlane;
    }
    bottom
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type empty;
    }
}

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

porosityIndex file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
```

```
    format      ascii;
    class       volScalarField;
    location    "0";
    object      porosityIndex;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    top
    {
        type            symmetryPlane;
    }
    bottom
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            empty;
    }
}


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

A.2 fvSchemes solver validation

`fvSchemes` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwind grad(U);
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
    div((phi|interpolate(porosity)),U) Gauss limitedLinearV 1;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}


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

A.3 fvSolution solver validation

`fvSolution` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.1;
        smoother        GaussSeidel;
    }

    pFinal
    {
        $p;
        tolerance       1e-06;
        relTol          0;
    }

    pcorrFinal
    {
        $p;
        tolerance       1e-06;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        tolerance       1e-05;
        relTol          0;
    }

    UFinal
    {
    $U;
    tolerance 1e-06;
    relTol 0;
    }
}

PIMPLE
{
    momentumPredictor yes;
    nOuterCorrectors 1;
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
```

```
64      pRefCell        0;
65      pRefValue       0;
66  }
67
68  // ************************************************************************* //
```