Cite as: Yan Toe, Chit: Implementing Immersed Boundary Method for particle representation in OpenFOAM-v2112. In Proceedings of CFD with OpenSource Software, 2023, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2023

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementing Immersed Boundary Method for particle representation in OpenFOAM-v2112

Developed for OpenFOAM-v2112

Author:
Chit Yan Toe
Delft University of Technology
c.yantoe-1@tudelft.nl

Peer reviewed by:
Dr. Saeed Salehi
Dr. Wim Uijttewaal
Rasmus Robertsson
Keivan Afshar Ghasemi

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

January 12, 2024

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:

- Basic knowledge of how the Immersed Boundary Method (IBM) is different from the body-fitted mesh in the representation of particles in the flow domain, and
- How to run tutorials for three-dimensional simulation of flow around a stationary rigid sphere in the rectangular channel using three solvers, namely `porousPimpleIbFoam` developed by Vergassola [1], `pimpleDyMIbFoam` developed by Jasak [2] in `foam-extend-5.0` [3], and `sdfibm` solver developed by Zhang [4] in `OpenFOAM v9`.

The theory of it:

- General background of IBM in the context of particle transport,
- Basic concepts of continuous forcing approach in `porousPimpleIbFoam` solver,
- Basic concepts of discrete forcing approach in `pimpleDyMIbFoam` solver, and
- Basic concepts of discrete forcing approach in `sdfibm` solver.

How it is implemented:

- Implementation of continuous forcing approach in `porousPimpleIbFoam` solver, and discrete forcing approach in `pimpleDyMIbFoam` solver and in `sdfibm` solver.

How to modify it:

- How to reimplement the discrete forcing approach of `sdfibm` of Zhang [4] into `OpenFOAM-v2112` ESI version [5] using a typical incompressible flow solver `pimpleFoam`, and
- After modification, how to simulate a three-dimensional simulation of a stationary rigid sphere in the laminar flow using the reimplemented solver, `sdfIbmESI` solver.

Prerequisites

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

- Basic knowledge of Object-Oriented approach in C++ programming language,
- Fundamental knowledge of Computational Fluid Dynamics (CFD), particularly of Finite Volume Method (FVM),
- Basic knowledge of using `OpenFOAM` and `Linux` operating system (recommended version: Ubuntu 20.04.6 LTS),
- Applied knowledge of compiling procedure to develop a solver in `OpenFOAM-v2112` ESI version.

Contents

Nomenclature

Acronyms

$\mathfrak{Re}$	Reynolds number
`OpenFOAM`	Open-source Field Operation And Manipulation
`sdfIbmESI`	Reimplemented `sdfibm` solver for `OpenFOAM-v2112` version
`sdfibm`	`OpenFOAM` solver using IBM with sdf method
CFD	Computational Fluid Dynamics
DEM	Discrete Element Model
ESI	Engineering System International company
FSI	Fluid-Structure Interaction
FVM	Finite Volume Method
IB	Immersed Boundary
IBM	Immersed Boundary Method
IBS	Immersed Boundary Surface
KC	Keulegan-Carpenter number
PIMPLE	PISO-SIMPLE
PISO	Pressure-Implicit with Splitting of Operators
PVF	Porous Volume Fraction
RHS	Right Hand Side
sdf	Signed Distance Function
SIMPLE	Semi-Implicit Method for Pressure-Linked Equations
STL	Stereolithography
VARANS	Volume-Averaged Reynolds-averaged Navier-Stokes
VOF	Volume-of-Fluid

English symbols

$\tilde{\boldsymbol{u}}^*$	Second predicted velocity vector considering solid particles	m/s
$\tilde{\boldsymbol{u}}$	First predicted velocity vector assuming no solid particles	m/s
$\boldsymbol{F}$	Distributed forcing term	$\mathrm{m/s^2}$
$\boldsymbol{f}$	Localized forcing term	$\mathrm{m/s^2}$
$\boldsymbol{G}$	External forces	$\mathrm{kg \cdot m/s^2}$
$\boldsymbol{g}$	Gravitational acceleration vector	$\mathrm{m/s^2}$
$\boldsymbol{I}$	Inertia tensor	$\mathrm{kg \cdot m^2}$
$\boldsymbol{N}$	External moments	$\mathrm{kg \cdot m^2/s^2}$
$\boldsymbol{n}$	Normal unit vector pointing outwards	
$\boldsymbol{r}$	Position vector	m
$\boldsymbol{T}$	Torque	$\mathrm{kg \cdot m^2/s^2}$
$\boldsymbol{u}$	Velocity vector	m/s
$\boldsymbol{V}$	Velocity vector of the immersed boundary	m/s
$\boldsymbol{v}$	Velocity vector of particle's centroid	m/s
$\boldsymbol{X}$	Lagrangian point	
$\boldsymbol{x}$	Coordinate vector	m

$\hat{\boldsymbol{u}}$	Velocity vector calculated assuming no solid particles	m/s
$\mathbb{R}^n$	Real coordinate space of n dimensions	
a	Coefficient linear porous drag term	
b	Coefficient quadratic porous drag term	
c	Coefficient transient porous drag term	
d	Distance function	
D_{50}	Nominal diameter	m
m	Mass	kg
n	Porosity	
p	Pressure	$\mathrm{kg/m/s^2}$
S	Surface area	$\mathrm{m^2}$
t	Time	s
V	Volume	$\mathrm{m^3}$

Greek symbols

α	Volume fraction	
β	Coefficient for quadratic porous drag term	
$\boldsymbol{\omega}$	Angular velocity	1/s
$\boldsymbol{\tau}$	Stress tensor	$\mathrm{kg/m/s^2}$
Δ	Discretized increment	
δ	Dirac delta function	
Γ	Boundary of the domain	
γ	Volume-of-Fluid	
μ	Dynamic viscosity	kg/m/s
ν	Kinematic viscosity	$\mathrm{m^2/s}$
Ω	Bounded region of the domain	
$\partial\Omega$	Boundary of the domain Ω	
ϕ	Flux	
ρ	Density	$\mathrm{kg/m^3}$
$\tilde{\phi}$	Flux resulted from interpolation of $\tilde{\boldsymbol{u}}$ onto mesh faces	
φ	Signed distance function	
ξ	Coefficient for linear porous drag term	

Superscripts

n	current time step
T	transpose

Subscripts

I	interface
cell	computational cell
corr	correction
f	fluid phase
face	face of the computational cell
p	solid particle
subCell	cut cell internal to the immersed boundary

Chapter 1

Introduction

In this report, we present different implementations of immersed boundary method (IBM) from three `OpenFOAM` solvers, namely `porousPimpleIbFoam`, `pimpleDyMIbFoam` and `sdfibm` solvers. The first solver, `porousPimpleIbFoam` solver, was developed by Vergassola [1] using `OpenFOAM-v2006` for the application of flow through a porous media. The second one, `pimpleDyMIbFoam` solver, was implemented by Jasak [2] in `foam-extend-5.0`. The final solver, `sdfibm`, was developed by Zhang [4] in `OpenFOAM v9`. These three solvers implemented IBM differently from each other in terms of forcing technique. Using these three solvers, simulation of the flow around a solid sphere in the rectangular channel was performed. Results from these simulations are also presented in the report.

The `porousPimpleIbFoam` solver used the continuous forcing technique while the latter two solvers, `pimpleDyMIbFoam` and `sdfibm`, used the discrete forcing technique, in order to impose required force on the immersed boundary surface. Of these two solvers, `sdfibm` solver imposed volume-average discrete force [4]. Among the three solvers, the `sdfibm` solver is emphasized in this report because of its simplicity in code implementation, explaining theoretical background of direct forcing technique in IBM framework. Besides, code implementation of the direct forcing technique in `sdfibm` solver is also explained in Section 2.4 of the report.

As a modification work of this project, IBM method of `sdfibm` solver [4] was reimplemented in `OpenFOAM-v2112` version. The motivation is to show the differences in compiling procedure between two different `OpenFOAM` versions. Brief procedure for modifying the code and compiling the solver is provided in the spirit of `OpenFOAM-v2112`. For this newly reimplemented solver, a benchmark test case that is a two-dimensional simulation of flow around a cylinder was performed. This benchmark test is available in the GitHub repository of Zhang [4] for `sdfibm` solver. Finally, the same simulation of flow around a solid sphere was repeated again using the newly reimplemented solver, the so-called `sdfIbmESI` solver.

As a note to the reader, since the whole process of reimplementing the solver is a long procedure, only important steps are discussed in the report for the sake of brevity. However, the final codes of this new solver `sdfIbmESI` and its tutorial case, which is a simulation of flow around a solid sphere, are provided as the accompanying files in the course repository and GitHub repository `https://github.com/ChitYanToe/sdfIbmESI.git`.

Chapter 2

Background

2.1 Immersed Boundary Method

Under mesh-based framework of CFD simulations, fluid-structure interaction problems can be solved using either (1) time-dependent body-conformal mesh or (2) time-independent structured grid. In the former approach, the computational mesh is *conformed* to the geometry of the structure in the flow domain. Therefore, if the geometry of the structure is changing during the simulation time, the background computational mesh needs to be changed accordingly. This requires re-meshing processes that is computationally complicated throughout the simulation history [6]. However, in the latter method, there is only one structured background mesh. This background mesh occupies not only the flow domain but also the solid or structure domain, therefore requiring no coordinate transformation that is impractical otherwise for complex geometries. In the so-called "Immersed Boundary Method (IBM)" termed by Peskin [7], momentum equation of Navier-Stokes equations is modified as

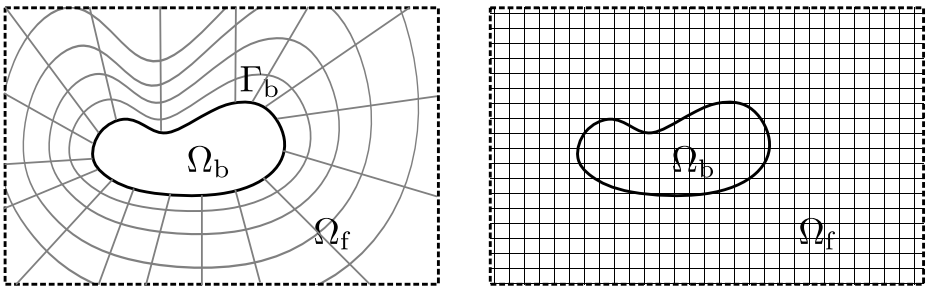
$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{\nabla p}{\rho} + \nu \Delta \boldsymbol{u} + \boldsymbol{f} \tag{2.1}$$

by adding a forcing term $\boldsymbol{f}$ that arises due to the interaction force of the boundary onto the flow domain [8]. In other words, $\boldsymbol{f}$ can be interpreted as the force acted by the particle on the fluid according to Newton's third law. This means that fluid recognizes the presence of the particle correctly. Here, $\boldsymbol{u}$ denotes velocity vector, p denotes pressure, ρ is density of the fluid, and ν kinematic viscosity of the fluid.

Fig. 2.1 shows a body-conformal mesh and a typical mesh commonly used in immersed boundary method. In the body-conformal mesh, the solid body Ω_b is excluded from the background curvilinear mesh on which the fluid domain Ω_f is discretized. On the boundary of solid body Γ_b, the required boundary condition is simply defined without any complication that may encounter in case of IBM. However, in IBM the solid body is immersed inside the fluid domain which is discretized by a simple structured grid. Therefore, IBM does not require the complicated meshing process. Due to the presence of the solid body, a forcing term $\boldsymbol{f}(\boldsymbol{x}, t)$ needs to be added to the Navier-Stokes equation Eq. (2.1). The difficulty lies in how accurately the forcing term should be estimated to obtain the correct representation of the solid boundary. The forcing term generally appears when the fluid velocity differs from the particle velocity.

Compared to the traditional body-conformal mesh approach, the main disadvantage of IBM is the difficulty of satisfying the exact velocity boundary conditions and mass conservation at interface cells [8]. Another drawback of IBM is that the local grid refinement of boundary mesh cannot be done in preferential direction to resolve the turbulent flow problems, because IBM cannot distinguish whether the grids in computational mesh are oriented normally or tangentially with respect to the boundary surface. However, this is not the case in body-conformal mesh approach.

Different approaches of how to impose the forcing term $\boldsymbol{f}(\boldsymbol{x}, t)$ onto the flow domain give rise to a variety of immersed boundary methods. There are two main groups of IBM that are different



(a) Body-conformal mesh (b) Immersed boundary mesh

Figure 2.1: (a) Body-conformal mesh in which the solid body is excluded from the background curvilinear mesh, (b) the solid body immersed inside the fluid domain which is discretized using structured grid. Illustration inspired from Mittal and Iaccarino [6].

in technique of imposing the forcing term, namely (1) continuous forcing approach, and (2) discrete forcing approach. Review paper of Verzicco [8] is recommended for further details.

2.2 Continuous Forcing Approach

In the continuous forcing approach as applied by Peskin [7], the forcing term $\boldsymbol{f}(\boldsymbol{x},t)$ is added to Eq. (2.1) before it is discretized numerically. The forcing term acts as the source term on the fluid equation at the particular boundary points of the solid body as a localized forcing term. Therefore, the forcing term in Eq. (2.1) can be written as

$$\boldsymbol{f}(\boldsymbol{x},t) = \sum_k \boldsymbol{F}(\boldsymbol{X}_k,t)\delta(|\boldsymbol{x}-\boldsymbol{X}_k|) \tag{2.2}$$

where δ is Dirac delta function, $\boldsymbol{X}_k$ is the k^{th} Lagrangian point located at the boundary of the solid body. Here, $\boldsymbol{F}$ denotes the force acted by the solid boundary onto the fluid. Generally, the boundary node or Lagrangian point does not coincide with the node of the background structured mesh. Therefore, the forcing term of one Lagrangian point distributes among its neighbouring nodes via a smoother distribution function. Different smoother distribution functions can be applied for the transfer of force between Eulerian background mesh and Lagrangian boundary points [6]. Again it is also necessary to transfer back the forces from the background mesh to the Lagrangian points in the next time step of computing its updated position.

Feedback forcing approach developed by Goldstein et al. [9] is a commonly used approach in the framework of continuous forcing approach. In that particular approach, the required forcing term is determined in the feedback-loop manner as the force adapting itself to the local flow field.

In fact, the continuous forcing approach encompasses the forcing term in the governing equations at the continuous level, before discretization. Therefore, the forcing term can be incorporated in the governing equation via a drag force term as explained by Vergassola [1].

2.2.1 `porousPimpleIbFoam` solver

The `porousPimpleIbFoam` solver developed by Vergassola [1] applies the Volume-Averaged Reynolds-averaged Navier-Stokes (VARANS) equations

$$\nabla\cdot\left(\frac{\boldsymbol{u}}{n}\right) = 0, \tag{2.3}$$

$$(1+c)\frac{\partial}{\partial t}\frac{\rho\boldsymbol{u}}{n} + \frac{\boldsymbol{u}}{n}\cdot\nabla\left(\frac{\rho\boldsymbol{u}}{n}\right) = -\nabla p + \rho\boldsymbol{g} + \nabla\cdot\left(\mu\nabla\frac{\boldsymbol{u}}{n}\right) - a\frac{\boldsymbol{u}}{n} - b\frac{\boldsymbol{u}}{n}\left\|\frac{\boldsymbol{u}}{n}\right\|, \tag{2.4}$$

where

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

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

to simulate the flow through the porous media. Here, n denotes porosity of the medium, a denotes coefficient for linear porous drag term, b denotes coefficient quadratic porous drag term, μ is the dynamic viscosity, D_{50} is the nominal diameter of porous material, and KC is Keulegan-Carpenter number. For complete permeability it has $n = 1$ whilst impermeable medium has $n = 0$. Coefficients ξ, β and c require calibration from experimental data [1]. In this formulation, the forcing term is the drag force term of the porous media, and the equations are solved throughout the whole domain including the porous media.

To treat the porosity values at the immersed boundary (IB) cells conveniently, `porosity` n can be rewritten using γ as

$$n_{\mathrm{corr}} = \gamma + n(1-\gamma) \tag{2.7}$$

where γ is Volume-of-Fluid (VOF)-like field defined as Porous Volume Fraction, PVF $= V_{\mathrm{subCell}}/V_{\mathrm{cell}}$. While V_{cell} is the volume of the cell, V_{subCell} is defined by the volume of the cut cell internal to the IB. Inside the immersed body $\gamma = 0$, therefore $n_{\mathrm{corr}} = n$. At the interface, $0 < \gamma < 1$ leads to appropriate n values, Elsewhere $\gamma = 1$, $n_{\mathrm{corr}} = 1$ i.e. perfect permeability. By doing so, the forcing term acts on the computational cells accordingly, without any further modifications.

Directory structure of `porousPimpleIbFoam` is shown below. The solver was designed based on the standard `pimpleFoam` solver of `OpenFOAM-v2006`. For more information on this solver, the report of Vergassola [1] is referred, but not included in the accompanying files. The `porousPimpleIbFoam.C` is the main file for the solver, which includes `UEqn.H` and `pEqn.H` files. To create the immersed boundary mask, the `createPorousIbMask.H` file is necessary and included in `createPorosity.H` file which creates the porosity field in the whole computational domain according to Eq. (2.7).

Directory of `porousPimpleIbFoam` solver

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

Implementation of momentum equation Eq. (2.4) in `UEqn.H` is described in Listing 2.1. As mentioned earlier, `createPorousIbMask.H` handles the creation of the immersed boundary by identifying the cells as `inside`, `outside` and `cut` cells, and finally assigning the `gamma` values, γ. To do so, a stereolithography (STL) geometry file of the immersed body needs to be provided to create the immersed boundary inside the background mesh. It should be noted that since the derivative terms in Eq. (2.4) or Listing 2.1 are divided by `porosity` value n, the solver works well only for porous media, rather than an impermeable body. Otherwise, it can lead to extremely small time steps in the computation.

Listing 2.1: Implementation of momentum equation in `UEqn.H` of `porousPimpleIbFoam` solver

```
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));

    fvOptions.correct(U);
}
```

2.3 Discrete Forcing Approach

In the discrete forcing approach, the forcing term is added to the discretized version of the governing equations. After discretizing the equations, the entries in the matrix are changed according to the location of the immersed boundary in the domain [6]. Due to its nature, discrete forcing approach results in different versions, depending on the discretization method. Among *indirect* and *direct* imposition methods under discrete forcing approach, we will emphasize direct imposition method here because it was implemented in `pimpleDyMIbFoam` solver of `foam-extend-5.0` [2] and `sdfibm` solver of Zhang [4].

2.3.1 Direct Forcing Method

To explain the general procedure of the direct forcing method according to Fadlun et al. [10], we reconsider Eq. (2.1) and discretize it numerically as

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^{n}}{\Delta t} = \mathrm{RHS}^{n+1/2} + \boldsymbol{f}^{n+1/2} \tag{2.8}$$

where $\mathrm{RHS}^{n+1/2}$ includes convective and diffusive terms and the pressure gradient term while $\boldsymbol{f}^{n+1/2}$ is the forcing term that needs to be calculated. If we require that the fluid velocity on the immersed boundary is equal to the velocity of the immersed boundary, such that $\boldsymbol{u}^{n+1} = \boldsymbol{V}^{n+1}$, the necessary forcing term will be

$$\boldsymbol{f}^{n+1/2} = -\mathrm{RHS}^{n+1/2} + \frac{\boldsymbol{V}^{n+1} - \boldsymbol{u}^{n}}{\Delta t}. \tag{2.9}$$

for some selected grid nodes and zero elsewhere [10, 11]. Due to discrepancy in positions between background grid nodes and immersed boundary points, an appropriate interpolation is necessary to calculate the forcing term Eq. (2.9) [10].

2.3.2 IBM implementation in `foam-extend-5.0`

In this subsection, a variant of direct forcing method introduced by Jasak [2], the so-called Immersed Boundary Surface (IBS) method is discussed following the work of Döhler [12]. Differently from previous versions — `foam-extend-3.2` and `foam-extend-4.0`, in this new version any cell intersected by the boundary surface are assumed immersed boundary cells, regardless of the position of the intersected cell's center. These intersected cells are cut by the boundary surface using a linear cut function and the resulting divided cells/faces are categorized as living cells/faces and dead cells/-faces. The living cell is not combined with a neighbouring fluid cell but it will remain a separate entity. Accordingly, the new cell center and face center of this separate cell need to be calculated for calculation of the face flux. At the same time, all dead cells are removed from the discretization matrix [2, 12].

IBS method of foam-extend-5.0 is implemented in immersedBoundary directory shown below. Inside the immersedBoundary sub-directory shown in Listing 2.2 under the immersedBoundary top directory, detailed implementation of immersed boundary surface can be found. Döhler [12] discussed and investigated the limitations of the IBS implementation in foam-extend-4.1 nextRelease.

Directory of immersedBoundary method in foam-extend-5.0

```
./foam-extend-5.0/src/immersedBoundary
|-- CMakeLists.txt
|-- immersedBoundary
|-- immersedBoundaryDynamicMesh
`-- immersedBoundaryTurbulence
```

Listing 2.2: Sub-directory of immersedBoundary in foam-extend-5.0

```
immersedBoundary
|-- immersedBoundaryFvPatch
|-- immersedBoundaryFvPatchFields
|-- immersedBoundaryFvsPatchFields
|-- immersedBoundaryPointPatch
|-- immersedBoundaryPolyPatch
|-- immersedPoly
`-- lnInclude
```

2.3.2.1 pimpleDyMIbFoam solver

pimpleDyMIbFoam solver is an immersed boundary solver of foam-extend-5.0 using PIMPLE algorithm for pressure-velocity coupling. Its directory is shown in Listing 2.3. This solver included the header files immersedBoundaryPolyPatch.H and immersedBoundaryFvPatch.H to use their classes, respectively. Detailed explanation of these two classes is given by the work of Döhler [12].

This solver also requires an STL file for the immersed boundary of the body and assigns it as a boundary patch that is similar to the conventional boundary patch of the flow domain. Therefore, dynamic motion of the immersed boundary can be controlled in the same manner as boundary conditions of dynamic geometry.

Listing 2.3: Directory of pimpleDyMIbFoam solver in foam-extend-5.0

```
pimpleDyMIbFoam/
|-- Make
|-- UEqn.H
|-- correctMeshMotion.H
|-- correctPhi.H
|-- createControls.H
|-- createFields.H
|-- pEqn.H
|-- pimpleDyMIbFoam.C
`-- readControls.H
```

2.4 sdfibm solver

In this section, we will discuss another immersed boundary method solver that applied discrete (direct) forcing method, namely sdfibm solver. The sdfibm solver was developed by Zhang [4] under the framework of OpenFOAM v6 and OpenFOAM v9, variants of OpenFOAM foundation versions [13]. In this project work, we will discuss the updated version of sdfibm solver accessed on commit bb83e36 (HEAD → master, origin/master, origin/HEAD). An interested reader should follow this folder path to download from GitHub repository of sdfibm solver. That solver is required to be compiled with CMake and g++ with C++17.

2.4.1 Theory of sdfibm solver

Numerical algorithm of sdfibm solver was mainly based on work of Kajishima et al. [14]. The forcing method of this approach is volume-average discrete forcing method to consider the presence of the particles in the fluid. The governing equations for the fluid, namely continuity equation and Navier-Stokes equation, read

$$\nabla \cdot \boldsymbol{u}_{\mathrm{f}} = 0 \tag{2.10}$$

and

$$\frac{\partial \boldsymbol{u}_{\mathrm{f}}}{\partial t} + \boldsymbol{u}_{\mathrm{f}} \cdot \nabla \boldsymbol{u}_{\mathrm{f}} = -\frac{1}{\rho_{\mathrm{f}}} \nabla p + \nu_{\mathrm{f}} \nabla \cdot \left[\nabla \boldsymbol{u}_{\mathrm{f}} + (\nabla \boldsymbol{u}_{\mathrm{f}})^T \right] + \boldsymbol{g} \tag{2.11}$$

respectively [14]. Here, the subscript f denotes the fluid phase, ν is kinematic viscosity and $\boldsymbol{g}$ is the gravity vector. For incompressible fluids and single-phase flow applications, we can simplify Eq. (2.11) to get

$$\frac{\partial \boldsymbol{u}_{\mathrm{f}}}{\partial t} + \boldsymbol{u}_{\mathrm{f}} \cdot \nabla \boldsymbol{u}_{\mathrm{f}} = -\frac{1}{\rho_{\mathrm{f}}} \nabla p + \nu_{\mathrm{f}} \nabla^2 \boldsymbol{u}_{\mathrm{f}} \tag{2.12}$$

by incorporating the gravity term into pressure term [15].

Before considering to include the forcing term in the Eq. (2.12), equations of particle dynamics will be stated as

$$\frac{\mathrm{d}(m_{\mathrm{p}} \boldsymbol{v}_{\mathrm{p}})}{\mathrm{d}t} = \int_{S_{\mathrm{p}}} \boldsymbol{\tau} \cdot \boldsymbol{n} \mathrm{d}S + \boldsymbol{G}_{\mathrm{p}} \tag{2.13}$$

for linear momentum and

$$\frac{\mathrm{d}(\boldsymbol{I}_{\mathrm{p}} \boldsymbol{\omega}_{\mathrm{p}})}{\mathrm{d}t} = \int_{S_{\mathrm{p}}} \boldsymbol{r} \times (\boldsymbol{\tau} \cdot \boldsymbol{n}) \mathrm{d}S + \boldsymbol{N}_{\mathrm{p}} \tag{2.14}$$

for angular momentum [14]. Here, m_{p} denotes mass of the particle, $\boldsymbol{v}_{\mathrm{p}}$ is the velocity of the particle's centroid, $\boldsymbol{\tau}$ is stress tensor, $\boldsymbol{n}$ is unit normal vector pointing outwards at the surface, $\mathrm{d}S$ is differential element of the surface area, $\boldsymbol{G}_{\mathrm{p}}$ is external forces, $\boldsymbol{I}_{\mathrm{p}}$ is inertia tensor of the particle, $\boldsymbol{\omega}_{\mathrm{p}}$ is angular velocity of the particle, $\boldsymbol{r}$ is position vector of the particle surface relative to the particle's centroid, and $\boldsymbol{N}_{\mathrm{p}}$ denotes external moments. Integration is performed on the surface of the particle, S_{p}.

Having determined the velocity of a particle's centroid, $\boldsymbol{v}_{\mathrm{p}}$, we can calculate the velocity of the particle as

$$\boldsymbol{u}_{\mathrm{p}} = \boldsymbol{v}_{\mathrm{p}} + \boldsymbol{\omega}_{\mathrm{p}} \times \boldsymbol{r} \tag{2.15}$$

in which the first term of RHS is a contribution by translational motion and the second term is due to rotational motion [14].

To consider the volume-average force arising from the difference between particle velocity and fluid velocity, we define the volume-weighted average of velocity as

$$\boldsymbol{u} = (1 - \alpha)\boldsymbol{u}_{\mathrm{f}} + \alpha \boldsymbol{u}_{\mathrm{p}} \tag{2.16}$$

where α denotes volume fraction $\in [0, 1]$ [14]. Since α is 0 for fluid and 1 for solid fraction respectively, we have $\boldsymbol{u} = \boldsymbol{u}_{\mathrm{f}}$ for computational cells occupied only by fluid whereas $\boldsymbol{u} = \boldsymbol{u}_{\mathrm{p}}$ for the solid cells. At the interface cells where the particle's boundary intersects with the background (fluid domain) mesh, we need to find α for its averaged velocity. In the sdfibm solver, Zhang [4] applied signed distance function to calculate α value accurately.

For the particles whose boundary surface is imposed by *no-slip* and *no-penetration* boundary conditions, there exists a continuity restriction as

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.17}$$

for the volume-weighted average of velocity [14], too. Moreover, we rewrite the momentum equation Eq. (2.12) for the averaged velocity $\boldsymbol{u}$ as

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{H} - \nabla P + \boldsymbol{f}_{\mathrm{p}} \tag{2.18}$$

where

$$\boldsymbol{H} \equiv -\boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nu_{\mathrm{f}} \nabla^2 \boldsymbol{u},$$

$P \equiv p/\rho_{\mathrm{f}}$ and $\boldsymbol{f}_{\mathrm{p}}$ denotes fluid-solid interaction force i.e. the force arisen due to the presence of the particle in the fluid. In other words, $\boldsymbol{f}_{\mathrm{p}}$ is the force required to adjust the single-phase fluid velocity $\boldsymbol{u}_{\mathrm{f}}$ to the averaged velocity $\boldsymbol{u}$ defined by Eq. (2.16) [14].

Using the general scheme of temporal integration of Eq. (2.18), the velocity at time instance $n+1$ can be described as

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t \left(\boldsymbol{H} - \nabla P + \boldsymbol{f}_{\mathrm{p}} \right) \tag{2.19}$$

where n = current time instance and Δt = discrete time interval [14]. If the velocity is predicted as the fluid phase velocity regardless of any phase in the computational cell, we get

$$\hat{\boldsymbol{u}} = \boldsymbol{u}^n + \Delta t \left(\boldsymbol{H} - \nabla P \right) \tag{2.20}$$

in which no interaction force is included. The difference between $\hat{\boldsymbol{u}}$ and $\boldsymbol{u}^{n+1}$ in Eq. (2.19) will be $\boldsymbol{u}^{n+1} - \hat{\boldsymbol{u}} = \Delta t \boldsymbol{f}_{\mathrm{p}}$. For the solid cell, since we require $\boldsymbol{u}^{n+1} = \boldsymbol{u}_p$ i.e. the averaged velocity is of course the particle velocity, the interaction force $\boldsymbol{f}_{\mathrm{p}}$ will be $\left(\boldsymbol{u}_{\mathrm{p}} - \hat{\boldsymbol{u}}\right)/\Delta t$. If the cell is fully occupied by fluid, we will impose simply $\boldsymbol{f}_{\mathrm{p}} = \boldsymbol{0}$. Regarding the interface cells, the interaction force can be approximated as

$$\boldsymbol{f}_{\mathrm{p}} = \alpha \frac{\boldsymbol{u}_{\mathrm{p}} - \hat{\boldsymbol{u}}}{\Delta t} \tag{2.21}$$

where α needs to determined by geometrical functions. Therefore, if we activate the interaction force $\boldsymbol{f}_{\mathrm{p}}$ to Eq. (2.19) depending on the volume fraction α, we will obtain the correct velocity for the next time step.

2.4.1.1 Calculation algorithm

As a first step of IBM in `sdfibm` solver, the fractional step velocity is predicted by excluding the interaction force $\boldsymbol{f}_{\mathrm{p}}$ and pressure gradient term ∇P to get

$$\tilde{\boldsymbol{u}} = \boldsymbol{u}^n + \frac{\Delta t}{2} \left(3\boldsymbol{H}^n - \boldsymbol{H}^{n-1} \right) \tag{2.22}$$

using the Adams-Bashforth method [14]. Zhang [4] used the predicted velocity $\tilde{\boldsymbol{u}}$ in evaluating $\nabla^2 \boldsymbol{u}$, and so the resulting discretized equation can be written as

$$\tilde{\boldsymbol{u}} = \boldsymbol{u}^n + \frac{\Delta t}{2} \left[\left(-3\boldsymbol{u}^n \cdot \nabla \boldsymbol{u}^n \right) + \left(\boldsymbol{u}^{n-1} \cdot \nabla \boldsymbol{u}^{n-1} \right) + \nu_{\mathrm{f}} \left(\nabla^2 \tilde{\boldsymbol{u}} + \nabla^2 \boldsymbol{u}^n \right) \right]. \tag{2.23}$$

It should be noted that the influence of the particle is not yet considered in this step.

For illustration of deriving the pressure Poisson equation, let's assume that the predicted velocity $\tilde{\boldsymbol{u}}$ is modified by adding the interaction force $\boldsymbol{f}_{\mathrm{p}}^n$ to obtain the second predicted velocity

$$\tilde{\boldsymbol{u}}^* = \tilde{\boldsymbol{u}} + \Delta t \boldsymbol{f}_{\mathrm{p}}^n \tag{2.24}$$

where $\boldsymbol{f}_{\mathrm{p}}^n = \alpha(\boldsymbol{u}_{\mathrm{p}}^n - \tilde{\boldsymbol{u}})/\Delta t$ i.e., the interaction force arisen due to the difference between particle's velocity $\boldsymbol{u}_{\mathrm{p}}^n$ and the predicted velocity $\tilde{\boldsymbol{u}}$. Therefore, we rewrite the momentum equation Eq. (2.19) as

$$\boldsymbol{u}^{n+1} = \tilde{\boldsymbol{u}}^* - \Delta t \nabla P \tag{2.25}$$

using the second predicted velocity, $\tilde{\boldsymbol{u}}^*$. If this equation is substituted into continuity equation, the pressure Poisson equation is obtained as

$$\nabla^2 P = \frac{1}{\Delta t} \nabla \cdot \tilde{\boldsymbol{u}} + \nabla \cdot \boldsymbol{f}_{\mathrm{p}}^n \tag{2.26}$$

after substituting $\tilde{\boldsymbol{u}}^*$ by Eq. (2.24) as well. It is noted that in the numerical procedure the velocity field $\tilde{\boldsymbol{u}}$ is interpolated onto mesh faces to create a flux field $\tilde{\phi}$ as per requirement of FVM discretization

in OpenFOAM. Then, the resulting flux is used for calculation of $\nabla \cdot \tilde{\boldsymbol{u}}$ by summing up the flux through mesh faces, $\tilde{\phi}$.

After solving the pressure Poisson equation, the fractional step velocity is updated by adding the pressure term to the predicted velocity in order to get the pressure-corrected velocity

$$\hat{\boldsymbol{u}} = \tilde{\boldsymbol{u}} - \Delta t \nabla P. \tag{2.27}$$

Similarly, the flux field correction reads

$$\phi^{n+1} = \tilde{\phi} - \Delta t \nabla_{\text{face}} P \, \mathrm{d}S_{\text{face}} \tag{2.28}$$

where ∇_{face} represents gradient evaluated at the face, and $\mathrm{d}S_{\text{face}}$ is differential surface area of the face.

As the next step, the interaction force is calculated as

$$\boldsymbol{f}_{\text{p}} = \alpha^n \frac{\boldsymbol{u}_{\text{p}}^n - \hat{\boldsymbol{u}}}{\Delta t} \tag{2.29}$$

where α and $\boldsymbol{u}_{\text{p}}$ are values at the current time step. Afterwards, the pressure-corrected velocity is modified by adding the interaction force $\boldsymbol{f}_{\text{p}}$ to get the velocity at time step $n+1$

$$\boldsymbol{u}^{n+1} = \hat{\boldsymbol{u}} + \Delta t \boldsymbol{f}_{\text{p}}. \tag{2.30}$$

After updating the fluid velocity, the force and torque acting on the particle are calculated using volume integration

$$\boldsymbol{F}_{\text{p}}^n = -\rho_{\text{f}} \int_{V_{\text{p}}} \boldsymbol{f}_{\text{p}} \mathrm{d}V \tag{2.31}$$

and

$$\boldsymbol{T}_{\text{p}}^n = -\rho_{\text{f}} \int_{V_{\text{p}}} \boldsymbol{r} \times \boldsymbol{f}_{\text{p}} \mathrm{d}V. \tag{2.32}$$

Before we perform the particle's advection, the linear and angular velocities are updated for the new time step using the linear multistep method [4] using Eqs. (2.33)

$$\begin{aligned} \boldsymbol{v}_{\text{p}}^{n+1} &= \boldsymbol{v}_{\text{p}}^n + \left(\frac{3}{2} \boldsymbol{F}_{\text{p}}^n - \frac{1}{2} \boldsymbol{F}_{\text{p}}^{n-1} \right) \frac{\Delta t}{m} \\ \boldsymbol{\omega}_{\text{p}}^{n+1} &= \boldsymbol{\omega}_{\text{p}}^n + \left(\frac{3}{2} \boldsymbol{T}_{\text{p}}^n - \frac{1}{2} \boldsymbol{T}_{\text{p}}^{n-1} \right) \frac{\Delta t}{\boldsymbol{I}_{\text{p}}}. \end{aligned} \tag{2.33}$$

Finally, the particle's position $\boldsymbol{x}_{\text{p}}$ is moved according to Crank-Nicolson method [14, 4]

$$\boldsymbol{x}_{\text{p}}^{n+1} = \boldsymbol{x}_{\text{p}}^n + \frac{\Delta t}{2} \left(\boldsymbol{v}_{\text{p}}^n + \boldsymbol{v}_{\text{p}}^{n+1} \right). \tag{2.34}$$

2.4.1.2 Determination of alpha

To determine the volume fraction, alpha α, in the forcing term, the signed distance function (sdf) is used in combination with Pyramid decomposition method [4].

Signed distance functions are functions to represent the surface implicitly, which are a subset of implicit functions [16]. A signed distance function is defined by φ such that $|\varphi(\boldsymbol{x})| = d(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathbb{R}^n$ with $d(\boldsymbol{x}) = \min(|\boldsymbol{x} - \boldsymbol{x_I}|)$ for all interface points $\boldsymbol{x}_I \in \partial\Omega$ [16]. This means that for all points existing on the boundary $\boldsymbol{x}_I$ we have $\varphi(\boldsymbol{x}) = d(\boldsymbol{x}) = 0$. For the points inside the boundary surface, it holds $\varphi(\boldsymbol{x}) = -d(\boldsymbol{x})$ while for the outside points, there exists $\varphi(\boldsymbol{x}) = d(\boldsymbol{x})$ [16].

Fig. 2.2 shows the sdf of a boundary surface indicated by the $\varphi(\boldsymbol{x_I}) = 0$ contour, also with $\varphi(\boldsymbol{x}) > 0$ for the region outside the boundary. The inside region of the boundary surface is described by $\varphi(\boldsymbol{x}) < 0$ contours.

The advantage of sdf approach is that it describes information not only of the surface itself but also of its relation with the entire space i.e. the distance between every point and the boundary

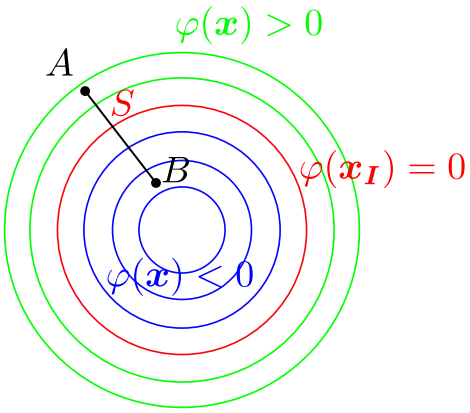


Figure 2.2: Example of signed distance function for a circle indicated by the red line ($\varphi = 0$ contour). The region outside the circle is shown in green lines ($\varphi > 0$) while for the inside region is $\varphi < 0$.

surface can be retrieved easily [4]. For instance, let us consider a line segment AB with point A located inside the surface S and point B outside S shown in Fig. 2.2, the fraction of AB lying within the surface S is simply the ratio: $-\varphi_A/(\varphi_B - \varphi_A)$ [4], which shall be used to calculate the volume fraction in `sdfibm` solver. It should be noted that points A and B are close enough to S such that local sdf is approximately linear [4]. In such a case, finding an intersection point is not necessary anymore, therefore avoiding a complicated step.

Another advantage of sdf approach is that different functions can be combined in boolean operations to obtain a composite geometry. Therefore, if two shapes described by φ_1 and φ_2 are unioned, we can easily take $\min(\varphi_1, \varphi_2)$ rather than finding their intersections explicitly [4].

Another ingredient of calculating α is Pyramid decomposition method that is used especially for the case of the boundary surface is intersected with the cubic mesh or n-face polyhedron cell. In brief, the intersected cell is divided into n pyramids, and the solid fraction for each face α_f is calculated. Summing up α_f-weighted solid volumes for all pyramids will give the total solid volume fraction at this cell [4]. Detailed explanation is given by the paper of Zhang et al. [17].

2.4.2 Implementation of `sdfibm` solver in `OpenFOAM v9`

In this subsection, implementations of `sdfibm` solver will be explained briefly in line with procedures of Subsection 2.4.1.1. Detailed algorithm of particle collision, shape generation, material generation and motions will not be discussed here because our emphasis is just to reimplement `sdfibm` solver in another variant of `OpenFOAM`.

Sub-directories and file contents of the `sdfibm` solver are shown below. Compilation of the source code is instructed in the `CMakeLists.txt` file. The `src` sub-directory contains all `C++` source codes that handle IBM calculations including particle collision, shape, motion and material. The `tool_vof` sub-directory includes the source codes for extra functionality that applies sdf approach to initialize the field of volume-of-fraction. That functionality is not related to IBM calculation of the solver, therefore it will not be discussed here.

Sub-directories and files of `sdfibm` solver

```
|-- CMakeLists.txt
|-- LICENSE.txt
|-- README.md
|-- examples
|-- figs
|-- src
`-- tool_vof
```

The `src` folder contains `main.cpp` file in which the main algorithm of the solver is implemented, and four sub-directories for libraries of collision, material, motion, and shape, respectively. The files

under the src directory are shown below, omitting some source code files and header files for the sake of clarity. The main.cpp file includes solidcloud.h header file for the IBM while the conventional OpenFOAM header files are included for solving fluid equations. The solidcloud.cpp is used for the IBM calculation and creating solid objects that will be explained later.

Sub-directories and files in src directory of sdfibm solver

```
src/
|-- CMakeLists.txt
|-- libcollision
|-- libmaterial
|-- libmotion
|-- libshape
|-- main.cpp
|-- solid.cpp
|-- solid.h
|-- solidcloud.cpp
|-- solidcloud.h
`-- utils.h
```

Listing 2.4 shows the code snippet of main.cpp that calls the solidcloud.h header file and creates a solidcloud object using sdfibm::SolidCloud class via sdfibm namespace. In order to create a solidcloud object, the required solidDict file is checked whether it exists in the case directory or latest time directory, or not. Then, the initial condition of solidcloud object is written out in cloud.out file according to the properties defined in solidDict file.

Listing 2.4: Creation of solidcloud object in main.cpp of sdfibm solver

```
#include "fvCFD.H"
#include "solidcloud.h"

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.h"
    #include "initContinuityErrs.H"

    std::string dictfile;

    // if start-time > 0, read from start-time-folder for solidDict, otherwise read from case root
    if(runTime.time().value() > 0)
    {
        if(!Foam::Pstream::parRun())
            dictfile = mesh.time().timeName() + "/solidDict";
        else
            dictfile = "processor0/" + mesh.time().timeName() + "/solidDict";
    }
    else
    {
        dictfile = "solidDict";
    }

    sdfibm::SolidCloud solidcloud(args.path() + "/" + dictfile, U, runTime.value());
    solidcloud.saveState();  // write the initial condition
```

Implementation for solving the momentum and pressure equations are shown in Listing 2.5. It is noted that though Zhang [4] did not discuss temperature or tracer equation in his paper, the sdfibm solver included this equation as a convection-diffusion type equation and solved it. Before solving the equations, the conditional testing is performed whether the solver runs for "fluid-structure interaction" (FSI) or "Discrete-element Model" (DEM). In the latter case, solving equations of fluid phase is disabled in calculation. If on_fluid flag is found as 1 in solidDict file, the momentum and pressure equations will be solved.

Listing 2.5: Solving momentum and pressure equations in main.cpp of sdfibm solver

```
30      while (runTime.loop())
31      {
32          Foam::Info << "Time = " << runTime.timeName() << Foam::endl;
33
34          #include "CourantNo.H"
35          Foam::dimensionedScalar dt = runTime.deltaT();
36
37          if(solidcloud.isOnFluid())
38          {
39              Foam::fvVectorMatrix UEqn(
40                  fvm::ddt(U)
41                + 1.5*fvc::div(phi, U) - 0.5*fvc::div(phi.oldTime(), U.oldTime())
42                ==0.5*fvm::laplacian(nu, U) + 0.5*fvc::laplacian(nu, U));
43              UEqn.solve();
44
45              phi = linearInterpolate(U) & mesh.Sf();
46              Foam::fvScalarMatrix pEqn(fvm::laplacian(p) == fvc::div(phi)/dt - fvc::div(Fs));
47              pEqn.solve();
48
49              U   = U   - dt*fvc::grad(p);
50              phi = phi - dt*fvc::snGrad(p)*mesh.magSf();
51
52              Foam::fvScalarMatrix TEqn(
53                  fvm::ddt(T)
54                + fvm::div(phi, T)
55                ==fvm::laplacian(alpha, T));
56              TEqn.solve();
57          }
```

In Listing 2.5, UEqn is constructed as fvVectorMatrix according to Eq. (2.23). To solve implicit part of Eq. (2.23) which is $\nabla^2\tilde{\boldsymbol{u}}$ in RHS, fvm::laplacian(nu,U) was used instead of using fvc::laplacian(nu,U). The latter one was used for solving the explicit part $\nabla^2\boldsymbol{u}^n$ of Eq. (2.23). It should be noted that Eq. (2.23) used second-order time discretization [4, 14] and the same manner was implemented in Listing 2.5. Then, flux field phi is created by interpolating the velocity field onto the mesh face that will be used in solving pressure Poisson equation.

To solve pressure Poisson equation Eq. (2.26), pEqn is constructed as fvScalarMatrix in which phi variable is used for the first divergence term of Eq. (2.26) that is operated dot product with the predicted velocity. Here, it should be noted that -fvc::div(Fs) is included in the code Listing 2.5 whereas it is not found in the original manuscript of Zhang et al. [17]. In fact, Eq. (2.26) was derived independently and checked with formulation of Breugem [18]. Nevertheless, although the forcing term is found in the code of Zhang [4], this term is subtracted in Listing 2.5 whereas it is added in Eq. (2.26). The reason why the minus sign appears in his code is due to implementation of forcing term in solidcloud.cpp file shown in Listing 2.6. Zhang [4] implemented (uf - us) to calculate the forcing term instead of (us - uf) as used in Eq. (2.21) where us is velocity of the solid particle and uf is the fluid velocity.

Listing 2.6: Implementation of forcing term in solidcloud.cpp of sdfibm solver

```
365         vector us = solid.evalPointVelocity(cc[icur]);
366         vector uf = m_Uf[icur];
367         vector localforce = alpha*cv[icur]*(uf - us)*dtINV;
368         force    += localforce;
369         torque   += (cc[icur]-solid.getCenter()) ^ localforce;
370         m_Fs[icur] += localforce/cv[icur];
```

If we go back to Listing 2.5, we see that pressure Poisson equation is solved by pEqn.solve(). Afterwards, the fractional step velocity U is corrected by adding the pressure term -dt*fvc::grad(p) according to Eq. (2.27). In a similar manner, flux term phi is corrected by pressure contribution as shown in Eq. (2.28). Then, the temperature or tracer T equation was constructed as a fvScalarMatrix and solved using TEqn.solve().

Now we will move on with the calculation of fluid-particle interaction as shown in Listing 2.7. At first, an interact() function of the solidcloud object is called to solve the fluid-particle interaction

problem. Detailed implementation of interaction can be found in Listing A.1. After obtaining interaction forces `Fs` from `solidFluidInteract` function (shown in Listing A.2), the velocity field `U` is updated according to Eq. (2.30) as well as the flux field `phi`. The reason why the minus sign appears in the code implementation instead of the plus sign was already explained above.

Listing 2.7: Updating for fluid-particle interaction in `main.cpp` of `sdfibm` solver

```
60          solidcloud.interact(runTime.value(), dt.value());
61
62          if(solidcloud.isOnFluid())
63          {
64              U = U - Fs*dt;
65              phi = phi - dt*(linearInterpolate(Fs) & mesh.Sf());
66
67              U.correctBoundaryConditions();
68              adjustPhi(phi, U, p);
69
70              T = (1.0 - As)*T + Ts;
71              T.correctBoundaryConditions();
72
73              #include "continuityErrs.H"
74          }
```

After updating the flux field, the boundary fields of `U` are corrected and to satisfy the mass continuity, flux balances are also adjusted. Then, temperature or tracer field `T` is updated and its boundary conditions are corrected. The `continuityErrs.H` header file is included to calculate and print the continuity errors in Listing 2.7.

Listing 2.8 performs the evolution of the solid particles in `solidcloud` object using its `evolve()` function. The `evolve` method of `SolidCloud` class includes computing particle-particle interaction by using `solidSolidInteract()` method and updating the particles' motion via `move` method of `solid` class. Listing A.3 shows implementation of `evolve` method in `solidcloud.cpp`.

Listing 2.8: Updating for fluid-particle interaction in `main.cpp` of `sdfibm` solver

```
76          solidcloud.evolve(runTime.value(), dt.value());
77          solidcloud.saveState();
78
79          if(solidcloud.isOnFluid())
80          {
81              solidcloud.fixInternal(dt.value());
82          }
```

To obtain forces and torques around the surface of a solid particle, Eqs. (2.31) and (2.32) are implemented inside the `solidFluidInteract` method shown in Listing A.2. The resulting force and torque values are used to update the velocity $\boldsymbol{v}_\mathrm{p}$ and angular velocity $\boldsymbol{\omega}_\mathrm{p}$ of the particles. To update as such, `addMidFluidForceAndTorque` method implements Eqs. (2.33) in Listing A.4. In Listing A.3, particle-particle interaction is handled by `solidSolidInteract` method of `SolidCloud` class.

As a next step in Listing 2.8, the particles are advected by using updated velocities according to Eq. (2.34). This action is implemented in Listing A.5. Afterwards, `fixInternal` method of `solidcloud` object corrects the fluid velocity inside the cells that are fully covered by the solid particles.

Finally, saving output files is managed by implementation of Listing 2.9. Moreover, `sdfibm` solver takes care of saving the latest `solidDict` file in the latest time directory that will be used to restart simulation.

Listing 2.9: Saving output files in `main.cpp` of `sdfibm` solver

```
84          if(runTime.outputTime())
85          {
86              runTime.write();
87
88              if(Foam::Pstream::master())
```

```
            {
                std::string file_name;
                if(Foam::Pstream::parRun())
                    file_name = "./processor0/" + runTime.timeName() + "/solidDict";
                else
                    file_name = "./" + runTime.timeName() + "/solidDict";
                solidcloud.saveRestart(file_name);
            }
        }
    }

    Foam::Info << "DONE\n" << endl;
    return 0;
}
```

2.5 Simulation of laminar flow around a sphere

Flow around a sphere was simulated to testify the working condition of three different solvers, namely `porousPimpleIbFoam`, `pimpleDyMIbFoam` and `sdfibm` solvers, in a three-dimensional setting. Since the `porousPimpleIbFoam` solver was intended to simulate the flow through the porous region and its governing equations contain division by n, the sphere in our simulation was required to be porous as well. Therefore, porosity of the sphere was set to 0.1 in the `constant/porosityDict` file. In the simulation, inflow velocity was set to 0.008 m/s to obtain Reynolds number $\mathfrak{Re} = 16$ for a laminar case. The dimensions of the background computational mesh are shown in Fig. 2.3, showing the names of the patches defined in `blockMeshDict` file.

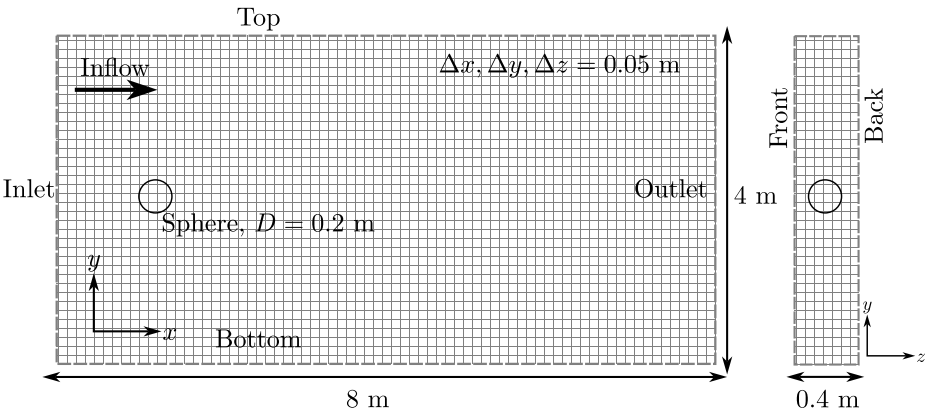
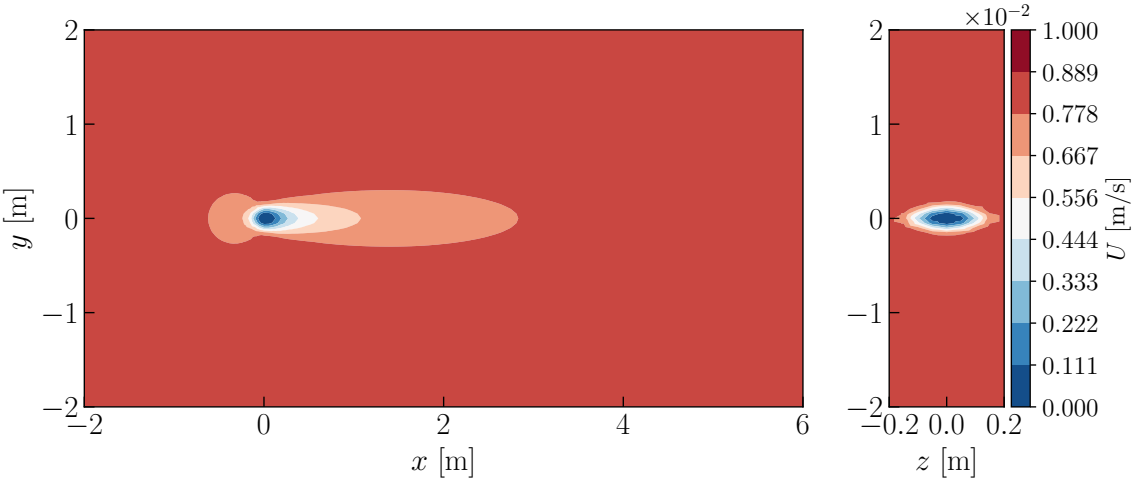


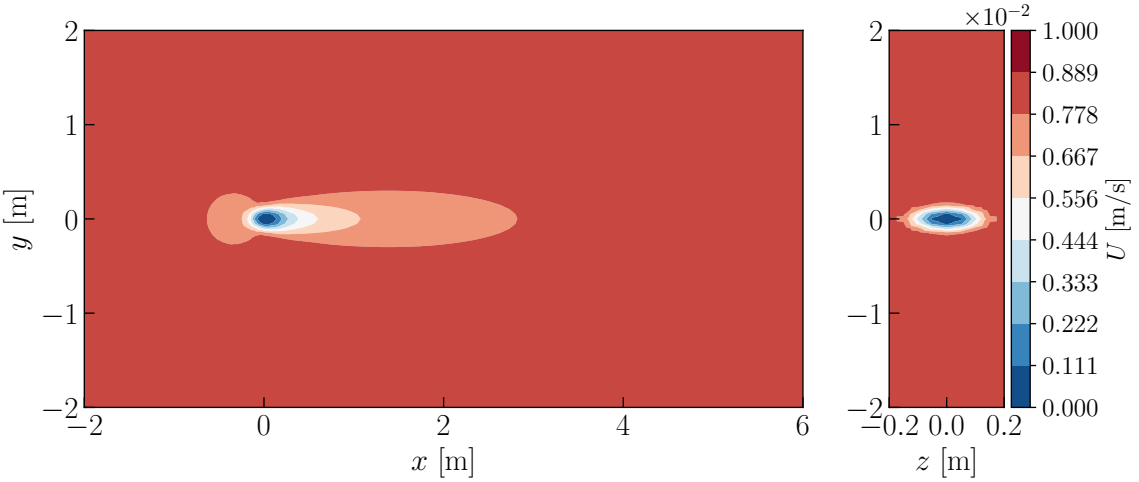
Figure 2.3: Geometry of computational mesh indicating the names of the patches defined in `blockMeshDict` file.

2.5.1 Simulation results from three different solvers

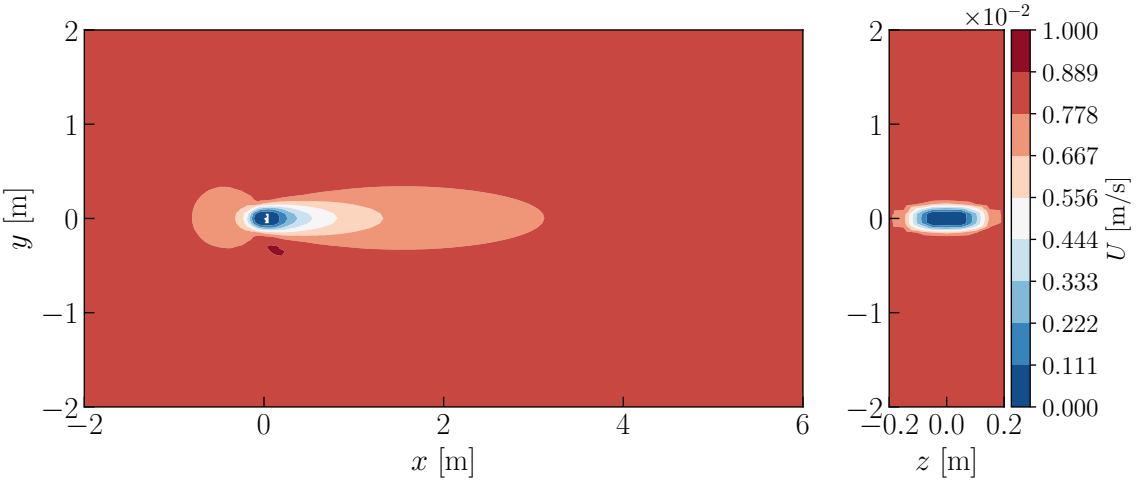
Figs. 2.4a, 2.4b and 2.4c show simulation results of flow velocity around a sphere using the three different solvers — `porousPimpleIbFoam`, `pimpleDyMIbFoam` and `sdfibm` solvers, respectively. As mentioned earlier, `porousPimpleIbFoam` solver was used for flow around a porous sphere while the other two solvers were used for flow around an impermeable sphere. Fig. 2.5 shows detailed comparison of longitudinal velocity U at $z = 0$ plane along the $x-$axis for $y = 0$, and along the $y-$axis for $x = 0$, respectively. In general, simulation results show a good agreement between three solvers although flow velocity of `sdfibm` solver differs slightly from the other two solvers, especially near the surface of the sphere shown in Fig. 2.5. It is worth to mention that we applied the same numerical settings for each simulation case. Hence, this difference might be due to different methods of immersed boundary implementation in each solver. However, we did not investigate the difference in this report since it is beyond the scope of the project.



(a) `porousPimpleIbFoam` solver



(b) `pimpleDyMIbFoam` solver



(c) `sdfibm` solver

Figure 2.4: Velocity results for flow around a sphere using three different solvers: (left) longitudinal transect, (right) transverse transect.

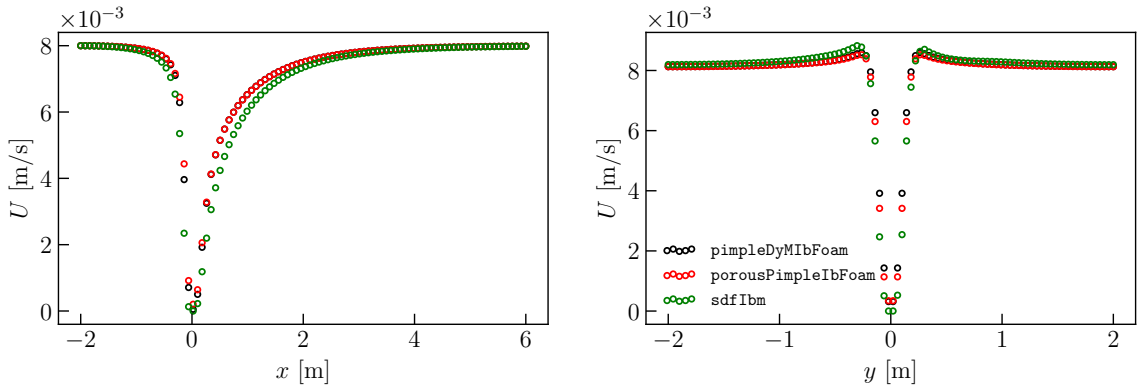


Figure 2.5: Comparison of longitudinal velocity U at $z = 0$ plane using three different solvers: (left) along the line $x-$axis for $y = 0$, (right) along the line $y-$axis for $x = 0$.

2.6 Conclusion

We briefly discussed a general idea of IBM in CFD simulations, and presented different approaches for imposing the forcing term in the hydrodynamic equations. Continuous forcing approach was applied in `porousPimpleIbFoam` solver and direct forcing approach was used in `pimpleDyMIbFoam` solver and `sdfibm` solver. The main difference between these two forcing approaches is that the former one adds the forcing term to the hydrodynamic equations before numerical discretization while the latter one adds after discretization. Regarding the representation of a particle in the flow domain, an STL file was required for its geometry in `porousPimpleIbFoam` and `pimpleDyMIbFoam` solvers. However, this file is not necessary in `sdfibm` solver because the solver uses standard predefined geometry shapes and can also easily combine these shapes to form a complex geometry, thanks to the sdf method.

Among the three solvers, `sdfibm` solver was emphasized, providing detailed explanation of its numerical algorithm and code implementation in `OpenFOAM v9` version. Moreover, a brief explanation of sdf method was presented for calculation of volume fraction. Implementation of IBM in `sdfibm` solver was found relatively simple compared to implementation in `pimpleDyMIbFoam` solver.

Finally, we simulated a test case for laminar flow ($\mathfrak{Re} = 16$) around a sphere using these three solvers. Longitudinal flow velocity was compared for each solver and results of `sdfibm` solver slightly differ from the other two solvers' results near the surface of the sphere. However, a good agreement was observed generally.

Chapter 3

Reimplementing sdfIbmESI solver

In Chapter 2, basic ideas of different IBM methods were introduced together with their implementations in different OpenFOAM solvers. Among these solvers, implementation of main.cpp in sdfibm solver was explained in more detail since we will reimplement this solver in OpenFOAM-v2112 ESI version. In this chapter, a brief procedure for reimplementing the solver is presented. The newly reimplemented solver is named sdfIbmESI solver as this solver is based on sdfibm solver and aligned with code and compilation standards of OpenFOAM-v2112.

3.1 sdfIbmESI solver

Philosophy of sdfIbmESI solver is just to introduce functionalities of sdfibm solver originally developed by Zhang [4] into OpenFOAM-v2112. Therefore, detailed code implementations will remain the same, except for some minor changes. Major changes or modifications need to be done in the compilation procedure because of a big difference in the compiling method between two OpenFOAM versions.

3.1.1 Creating sdfIbmESI directory

First of all, a new directory called sdfIbmESI is created by the Linux command mkdir sdfIbmESI under the directory $WM_PROJECT_USER_DIR/applications/solvers/incompressible/. Copy the files of any single-phase, incompressible solver e.g. pimpleFoam solver to the newly created directory sdfIbmESI. It is important to rename the file as sdfIbmESI.C and executable file name inside Make/files. Also, the executable file of the solver after compilation should be under the user's binary directory $(FOAM_USER_APPBIN). Typical directory of the sdfIbmESI solver should be similar to the one as shown below.

Typical directory of sdfIbmESI solver

```
sdfIbmESI/
|-- Make
|-- UEqn.H
|-- correctPhi.H
|-- createFields.H
|-- pEqn.H
|-- sdfIbmESI.C
`-- setRDeltaT.H
```

3.1.2 Creating the required library files

Before editing the contents of sdfIbmESI.C file, the required library files should be created first. The required library directories are libcollision for particle-particle collision, libmaterial for

generating material of the particle, libmotion for handling motion of the particle and libshape for generating shape of the particle.

To create the required libraries, a separate directory was created for each library under the top directory of sdfIbmESI. Under the directory of each library, Make directory was created along with two files which are files and options. Moreover, the required header files should be included in Make/options file to link with corresponding files via lnInclude. Such an example is shown in Listing 3.1 that shows the linkage of libcollision with other necessary libraries such as libshape. For more information on developing the library in OpenFOAM, the reader is referred to lecture slides on "OpenFOAM user directory organization and compilation" delivered by Chalmers University of Technology.

Listing 3.1: options file under Make directory of libcollision library

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/OpenFOAM/lnInclude \
    -I../libshape/lnInclude \
    -I../libmotion/lnInclude \
    -I../libmaterial \
    -I..

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

After successful compilation of library files, one should see three libFILENAME.so files, namely libcollision.so, libmotion.so and libshape.so, under their respective directories. These shared libraries need to be linked with the sdfIbmESI solver using environment variable LD_LIBRARY_PATH that is a dynamic link loader. This process can be automated by using the source command and exportFile.sh file that is provided in the accompanying files.

Here, it is worth mentioning that there is a difference in including method of header files between two solvers. In our newly reimplemented sdfIbmESI solver, header files were included by *only* indicating their file names as #include "FILENAME.H" while in the original solver these were included using relative path as #include "../FILENAME.H". The exact location of these header files were, however, directed in the corresponding Make/options file of our sdfIbmESI solver.

3.1.3 Adding other necessary files

It is also necessary to add other necessary files, for instance, solid.C, solidcloud.C and types.H in the top directory of sdfIbmESI solver. Therefore, one should see the final directory as below.

Final directory of sdfIbmESI solver

```
sdfIbmESI/
|-- Make
|-- cellenumerator.C
|-- cellenumerator.H
|-- createFields.H
|-- geometrictools.C
|-- geometrictools.H
|-- libcollision
|-- libmaterial
|-- libmotion
|-- libshape
|-- logger.C
|-- logger.H
|-- meshinfo.C
|-- meshinfo.H
|-- sdfIbmESI.C
|-- setRDeltaT.H
|-- solid.C
```

```
|-- solid.H
|-- solidcloud.C
|-- solidcloud.H
|-- types.H
`-- utils.H
```

Here, some minor changes need to be highlighted that arise due to version difference of two solvers. As a first example, type conversion was performed for `meta.lookup("gravity")` and `solid.lookup("pos")` by using `vector()` function in `solidcloud.C` file. Another example is also type conversion for `transportProperties.lookup("rho")` by using `readScalar` function in `solidcloud.C` file. Last example is inclusion of `IFstream.H` file by using `#include "IFstream.H"` in `solidcloud.C` file to be able to read `dictfile` file which is a user-input file e.g. `solidDict` file.

3.1.4 Editing `sdfIbmESI.C` file

Now, we will edit the contents of `sdfIbmESI.C` file by including `solidcloud.H` header file that handles immersed boundary method. By including this header file, we could create a `SolidCloud` object using the properties defined in `dictfile` file that will be provided in the case directory. The rest of the `sdfIbmESI.C` file were edited accordingly as in `main.cpp` of original `sdfibm` solver. The final version of `sdfIbmESI.C` is shown in Listing B.1.

3.1.5 Compiling `sdfIbmESI` solver

To compile the solver using `wmake` command, `Make` directory consisting of `files` file shown in Listing 3.2 and `options` file shown in Listing 3.3 are required. Executable output file after the compilation will be saved as `sdfIbmESI` under the user's binary directory `$FOAM_USER_APPBIN`.

Listing 3.2: `files` file under `Make` directory of `sdfIbmESI` solver

```
solidcloud.C
solid.C
meshinfo.C
logger.C
geometrictools.C
cellenumerator.C
sdfIbmESI.C

EXE = $(FOAM_USER_APPBIN)/sdfIbmESI
```

Listing 3.3: `options` file under `Make` directory of `sdfIbmESI` solver

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/functionObjects/field/lnInclude \
    -Ilibshape/lnInclude \
    -Ilibmotion/lnInclude \
    -Ilibcollision/lnInclude \
    -Ilibmaterial

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lincompressibleTransportModels \
```

```
    -Llibshape \
    -lshape \
    -Llibmotion \
    -lmotion \
    -Llibcollision \
    -lcollision
```

3.1.6 Testing with the benchmark case

To test the ability of the `sdfIbmESI` solver, a benchmark case was simulated which is a two-dimensional simulation of the flow around a cylinder in $\mathfrak{Re} = 200$. Fig. 3.1 shows the velocity contours for the flow around a cylinder around which vortex shedding was observed. This observation is indeed a typical feature of flow around the cylinder as shown by Zhang [4].

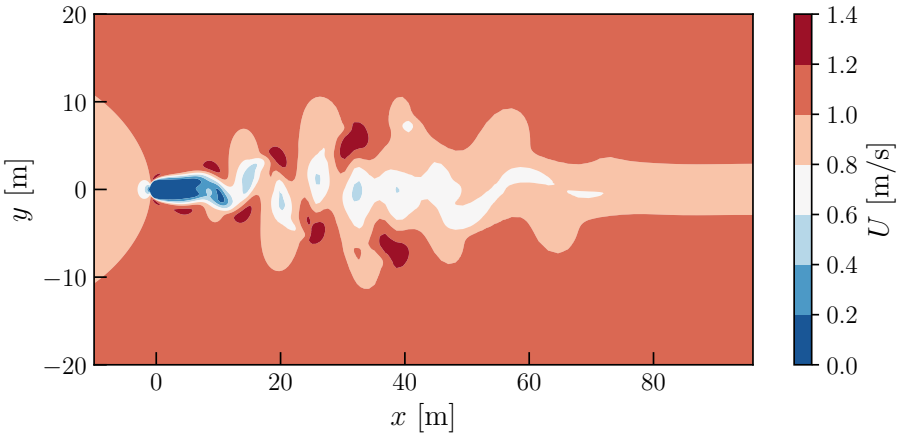


Figure 3.1: Velocity result for flow around a cylinder for $\mathfrak{Re} = 200$.

Chapter 4

Tutorial case for sdfIbmESI solver

To demonstrate the ability of the newly reimplemented **sdfIbmESI** solver, we repeat the same simulation of flow around a solid sphere as explained in Subsection 2.5.

4.1 Case Directory

Required files for the simulation are shown below and provided as the accompanying files in the course repository. In `0.orig` directory, `T` file denotes initial condition of temperature (or) tracer field (though in this case, it is not our main objective), `U` denotes initial condition for flow velocity, and `p` is the initial condition for pressure. More importantly, `solidDict` file is required in the top directory for creating a `solidcloud` object in `sdfIbmESI` solver. In case of `parallel` run, the `solidDict` file needs to be copied in each `processorID` directory. Material properties, geometrical shape, types of the motion, and initial position need to be defined of the solid object in the `solidDict` file.

Required files for simulation of flow around a solid sphere with `sdfIbmESI` solver

```
sdfIbmESISphereLaminar/
|-- 0.orig
|   |-- T
|   |-- U
|   `-- p
|-- Allclean
|-- Allrun
|-- constant
|   |-- transportProperties
|   `-- turbulenceProperties
|-- solidDict
`-- system
    |-- blockMeshDict
    |-- controlDict
    |-- fvSchemes
    `-- fvSolution
```

Listing 4.1 shows the definition of a sphere of radius 0.1 m using `Sphere` keyword and `radius` keyword in `solidDict` file. This sphere is named `sphere1`. To define different kinds of `shapes` e.g. circle, ellipse, it is explained in the paper of Zhang [4].

Listing 4.1: Definition of a sphere in `solidDict` file for `sdfIbmESI` solver

```
shapes
{
    shape1
    {
        name sphere1;
        type Sphere;
        radius 0.1;
```

```
    }
}
```

Also, definition of the material of the solid particle in the `solidDict` file is shown in Listing 4.2. For `material1` material, its density is defined as 1.0 kg/m^3 by the keyword `rho`. This defined material is termed as `mat1`.

Listing 4.2: Definition of material in `solidDict` file for `sdfIbmESI` solver

```
materials
{
    material1
    {
      name mat1;
      type General;
      rho 1.0;
    }
}
```

In Listing 4.3, allowable type of motion for the solid particle is prescribed in the `solidDict` file using the keywords `type` and `mask`. In this case, `mask` entry `b000000` means that six degrees of freedom of the solid body is frozen i.e. the solid sphere is fixed in space. This defined motion is named `static1`.

Listing 4.3: Definition of material in `solidDict` file for `sdfIbmESI` solver

```
motions
{
    motion1
    {
      name static1;
      type Motion01Mask;//Motion000000;
      mask b000000;
    }
}
```

Finally, a solid sphere is created in the `solidDict` file shown in Listing 4.4 using the defined properties of the particle, which are `sphere1`, `mat1`, `static1`, and initial position `pos` keyword and velocity `vel` keyword. In this case, the sphere is fixed at the origin `(0.0 0.0 0.0)` with initial velocity vector `(0.0 0.0 0.0)`.

Listing 4.4: Definition of a solid sphere `solid1` in `solidDict` file for `sdfIbmESI` solver

```
solids
{
    solid1
    {
      shp_name sphere1;
      mot_name static1;
      mat_name mat1;
      pos (0.0 0.0 0.0);
      vel (0.0 0.0 0.0);
    }
}
```

Properties of the fluid are defined in `transportProperties` file under `constant` directory as shown in Listing 4.5. The entries of the dictionary are self-explanatory.

Listing 4.5: Fluid properties defined in `transportProperties` file for `sdfIbmESI` solver

```
transportModel  Newtonian;

nu              [ 0 2 -1 0 0 0 0 ] 1e-04;// kinematic viscosity of the fluid
alpha           [ 0 2 -1 0 0 0 0 ] 0.01; // coefficient of diffusion
rho             1000;            // density of the fluid
```

Type of simulation is defined in `turbulenceProperties` file of `constant` directory, described in Listing 4.6. In this tutorial, the `laminar` flow simulation was applied.

Listing 4.6: Flow properties defined in `turbulenceProperties` file for `sdfIbmESI` solver

```
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType laminar;

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

In the `system` directory, `blockMeshDict` file specifies the geometry of the computational domain and mesh sizes. Also, under this `system` directory, `controlDict` file defines the computational time step, name of the solver and other options for saving output files. Discretization scheme and linear solver for the field variables need to be defined in `system/fvSchemes` and `system/fvSolution` files, respectively.

Two script files, `Allrun` and `Allclean`, are intended to automate the procedure of the simulation. Otherwise, `blockMesh` command needs to be used to create the computational domain, and `sdfIbmESI` command can be used in order to run the case. Therefore, in order to simulate the tutorial case, one can execute the commands shown in Listing 4.7.

Listing 4.7: Commands to run the tutorial case using `sdfIbmESI` solver.

```
cp -r 0.orig 0
blockMesh
sdfIbmESI
```

4.2 Simulation results

Fig. 4.1 shows longitudinal velocity of the flow around the solid sphere. As expected, the result from the `sdfIbmESI` solver is similar to the simulation result from the original `sdfibm` solver shown in Fig. 2.4c.

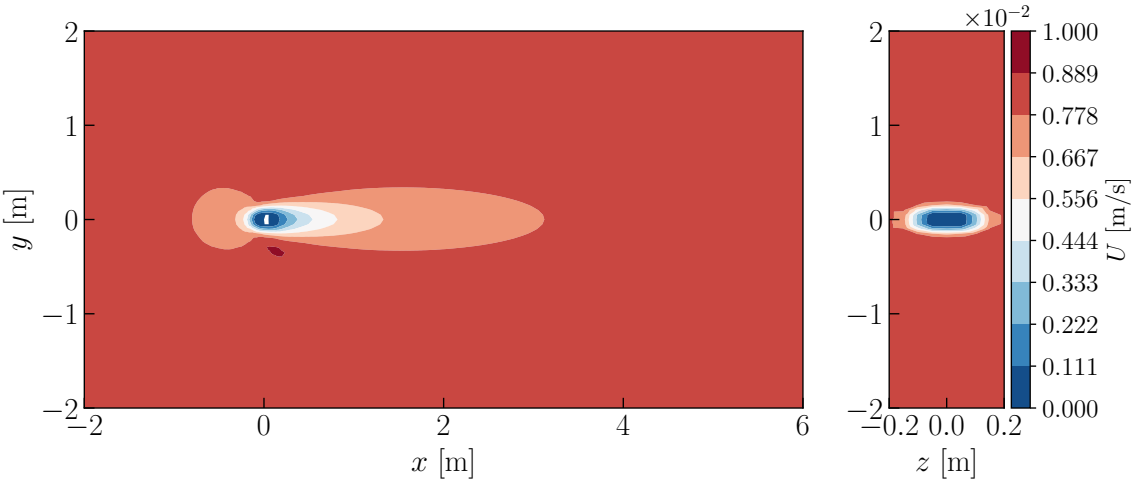


Figure 4.1: Longitudinal velocity of the flow around a solid sphere using `sdfibmESI` solver: (left) longitudinal transect, (right) transverse transect.

Detailed study of simulation results and their differences is beyond the scope of this project. Nevertheless, this reimplemented `sdfIbmESI` solver can provide targeted ability of the original `sdfibm`

solver. Moreover, one can find the `tool_vof` directory in the accompanying folder. The `tool_vof` is not directly related to IBM method, but can be used to initialize the VOF field of two-phase flow simulations. In this report, this aspect is not discussed for the sake of simplicity.

Bibliography

[1] M. Vergassola, "A continuous forcing immersed boundary approach to solve the varans equations in a volumetric porous region," in *In Proceedings of CFD with OpenSource Software, 2021*, Edited by Nilsson. H, 2021.

[2] H. Jasak, "Immersed boundary surface method in foam-extend," in *Workshop OpenFOAM in Hydraulic Engineering*, vol. 21, 2018, p. 22.

[3] "foam-extend-5.0," https://sourceforge.net/p/foam-extend/foam-extend-5.0/ci/master/tree/, accessed: 2023-10-23.

[4] C. Zhang, "sdfibm: A signed distance field based discrete forcing immersed boundary method in openfoam," *Computer Physics Communications*, vol. 255, p. 107370, 2020.

[5] "OpenFOAM® v2112," https://develop.openfoam.com/Development/openfoam/-/blob/maintenance-v2112/doc/Build.md?ref_type=heads, accessed: 2022-04-12.

[6] R. Mittal and G. Iaccarino, "Immersed boundary methods," *Annu. Rev. Fluid Mech.*, vol. 37, pp. 239–261, 2005.

[7] C. S. Peskin, "Flow patterns around heart valves: A numerical method," *Journal of Computational Physics*, vol. 10, no. 2, pp. 252–271, 1972. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999172900654

[8] R. Verzicco, "Immersed boundary methods: Historical perspective and future outlook," *Annual Review of Fluid Mechanics*, vol. 55, pp. 129–155, 2023.

[9] D. Goldstein, R. Handler, and L. Sirovich, "Modeling a no-slip flow boundary with an external force field," *Journal of computational physics*, vol. 105, no. 2, pp. 354–366, 1993.

[10] E. A. 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.

[11] M. Uhlmann, "An immersed boundary method with direct forcing for the simulation of particulate flows," *Journal of computational physics*, vol. 209, no. 2, pp. 448–476, 2005.

[12] J. E. Döhler, "An analysis of the immersed boundary surface method in foam-extend," Master's thesis, Chalmers University of Technology, 2022.

[13] O. Foundation. (2024) Openfoam and the openfoam foundation. [Online]. Available: https://openfoam.org

[14] T. Kajishima, S. Takiguchi, H. Hamasaki, and Y. Miyake, "Turbulence structure of particle-laden flow in a vertical plane channel due to vortex shedding," *JSME International Journal Series B Fluids and Thermal Engineering*, vol. 44, no. 4, pp. 526–535, 2001.

[15] J. H. Ferziger, M. Perić, and R. L. Street, *Computational methods for fluid dynamics*. springer, 2019.

[16] S. Osher and R. P. Fedkiw, *Level set methods and dynamic implicit surfaces*. Springer New York, 2005, vol. 1.

[17] C. Zhang, C. Wu, and K. Nandakumar, "Effective geometric algorithms for immersed boundary method using signed distance field," *Journal of Fluids Engineering*, vol. 141, no. 6, p. 061401, 2019.

[18] W.-P. Breugem, "A second-order accurate immersed boundary method for fully resolved simulations of particle-laden flows," *Journal of Computational Physics*, vol. 231, no. 13, pp. 4469–4498, 2012.

Study questions

How to use it:

1. What is the main difference between IBM and body-fitted mesh in terms of governing equations?
2. For a complex computational geometry in which detailed meshing is almost impossible, which approach should be preferred among IBM and body-fitted mesh?

The theory of it:

1. Describe the main difference between continuous forcing approach and discrete forcing approach regarding the forcing term.
2. Which *specific* method of discrete forcing approach is applied in `sdfibm` solver?

How it is implemented:

1. When used `sdfIbmESI` or `sdfibm` solver, an STL file is not required to generate the immersed boundary or surface. How is it possible in these two solvers, while it is not in `porousPimpleIbFoam` solver?
2. Which text file is essential to generate the immersed body in `sdfIbmESI` or `sdfibm` solver?

How to modify it:

1. How many libraries need to be linked in `Make/options` file for successful compilation of `sdfIbmESI` solver?
2. How can a solid body be created which shape is a circle of radius 1.0 m and its motion is free in all six degrees of freedom in the `sdfIbmESI` solver?

Appendix A

Codes of `sdfibm` solver

Listing A.1: Implementation of `interact` method in `solidcloud.cpp` of `sdfibm` solver

```
397 void SolidCloud::interact(scalar time, scalar dt)
398 {
399     // reset solid field, which are source terms for the fluid solver
400     m_ct = 0;
401     m_As = 0.0;
402     m_Fs = Foam::dimensionedVector("zero", Foam::dimAcceleration, Foam::vector::zero);
403     m_Ts = Foam::dimensionedScalar("zero", Foam::dimTemperature, 0.0);
404     using namespace std::chrono;
405     high_resolution_clock::time_point t1 = high_resolution_clock::now();
406
407     for (Solid& solid : m_solids)
408         solidFluidInteract(solid, dt);
409
410     checkAlpha();
411
412     high_resolution_clock::time_point t2 = high_resolution_clock::now();
413     duration<double> t_elapse = duration_cast<duration<double>>(t2 - t1);
414
415     if (Foam::Pstream::master())
416     {
417         std::ostringstream msg;
418         msg << "t = " << std::setw(6) << time
419             << " [FSI took " << std::left << std::setprecision(3) << std::setw(6)
420             << 1000*t_elapse.count() << " ms]";
421         LOG(msg.str());
422     }
423
424     m_As.correctBoundaryConditions();
425     m_Fs.correctBoundaryConditions();
426     m_Ts.correctBoundaryConditions();
427 }
```

Listing A.2: Implementation of `solidFluidInteract` method in `solidcloud.cpp` of `sdfibm` solver

```
333 void SolidCloud::solidFluidInteract(Solid& solid, scalar dt)
334 {
335     m_geotools.clearCache();
336     const Foam::vectorField& cc = m_mesh.cellCentres();
337     const Foam::scalarField& cv = m_mesh.V();
338
339     scalar dtINV = 1.0/dt;
340     vector force  = vector::zero;
341     vector torque = vector::zero;
342
343     m_cellenum.SetSolid(solid);
344
345     int numInsideCell = 0;
```

```
346     int numBorderCell = 0;
347     int insideType = solid.getID() + 4;
348     scalar alpha = 0.0;
349     while (!m_cellenum.Empty())
350     {
351         int icur = m_cellenum.GetCurCellInd();
352         if (m_cellenum.GetCurCellType() == CellEnumerator::ALL_INSIDE)
353         {
354             ++numInsideCell;
355             alpha = 1.0;
356             m_ct[icur] = insideType;
357         }
358         else
359         {
360             m_ct[icur] = m_cellenum.GetCurCellType();
361             ++numBorderCell;
362             alpha = m_geotools.calcCellVolume(icur, solid, m_ON_TWOD)/cv[icur];
363         }
364 
365         vector us = solid.evalPointVelocity(cc[icur]);
366         vector uf = m_Uf[icur];
367         vector localforce = alpha*cv[icur]*(uf - us)*dtINV;
368         force    += localforce;
369         torque   += (cc[icur]-solid.getCenter()) ^ localforce;
370         m_Fs[icur] += localforce/cv[icur];
371         m_As[icur] += alpha;
372         m_Ts[icur] += alpha;
373 
374         m_cellenum.Next();
375     }
376 
377     if (Foam::Pstream::parRun())
378     {
379         Foam::reduce(numInsideCell, Foam::sumOp<Foam::scalar>());
380         Foam::reduce(numBorderCell, Foam::sumOp<Foam::scalar>());
381     }
382 
383     Foam::Info << "Solid " << solid.getID() << " has " << numInsideCell << '/'
384                << numBorderCell << " internal/boundary cells.\n";
385 
386     force  *= m_rhof;
387     torque *= m_rhof;
388 
389     if (Foam::Pstream::parRun())
390     {
391         Foam::reduce(force,  Foam::sumOp<Foam::vector>());
392         Foam::reduce(torque, Foam::sumOp<Foam::vector>());
393     }
394     solid.setFluidForceAndTorque(force, torque);
395 }
```

Listing A.3: Implementation of `evolve` method in `solidcloud.cpp` of `sdfibm` solver

```
483 void SolidCloud::evolve(scalar time, scalar dt)
484 {
485     m_time = time;
486     static label N_SUBITER = 20;
487     if (m_solids.size() == 1) N_SUBITER = 1;
488     scalar dt_sub = dt / N_SUBITER;
489     for (int i = 0; i < N_SUBITER; ++i)
490     {
491         // clear all forces
492         for (Solid& solid : m_solids)
493             solid.clearForceAndTorque();
494         // NOW Fn = 0.0
495 
496         for (Solid& solid : m_solids)
497         {
```

```
498             solid.addMidFluidForceAndTorque();
499         }
500         // NOW Fn = F_f
501
502         this->addMidEnvironment();
503         // NOW Fn = F_f + m*g
504
505         this->solidSolidInteract();
506         // NOW Fn = F_f + F_c + m*g
507
508         for (Solid& solid : m_solids)
509         {
510             solid.move(time, dt_sub);
511         }
512     }
513
514     // appear plane has no env force and no solid-solid interactoin TODO
515
516     for (Solid& solid : m_solids)
517         solid.storeOldFluidForce();
518 }
```

Listing A.4: Implementation of `addMidFluidForceAndTorque` method in `solid.h` of `sdfibm` solver

```
152     inline void addMidFluidForceAndTorque()
153     {
154         force  += (1.5*fluid_force  - 0.5*fluid_force_old);
155         torque += (1.5*fluid_torque - 0.5*fluid_torque_old);
156     }
```

Listing A.5: Implementation of `move` method in `solid.h` of `sdfibm` solver

```
163     void move(const scalar& time, const scalar& dt)
164     {
165         // motion = velocity & omega
166         // temporarily store motion at time n
167         vector velocity_old = velocity;
168         vector omega_old    = omega;
169
170         // update motion to n+1 using force at time n+1/2
171         velocity += force*mass_inv*dt;        // velocity updated to t + dt
172         tensor R = orientation.R();
173         tensor moi_inv_world = R & moi_inv & R.T();
174         omega += (moi_inv_world & torque)*dt; // omega updated to t + dt
175
176         // constrain motion
177         if(ptr_motion != nullptr)
178             ptr_motion->constraint(time, velocity, omega);
179
180         // position & orientation updated AFTER constraint
181         center      += 0.5*(velocity + velocity_old)*dt;
182         orientation += 0.5*quaternion(0.5*(omega + omega_old))*orientation*dt;
183         orientation.normalise(); // no need to normalise every step, but cheap anyway
184     }
```

Appendix B

Main file of `sdfIbmESI` solver

Listing B.1: `sdfIbmESI.C` of `sdfibmESI` solver

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "pimpleControl.H"
#include "fvOptions.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"

#include "solidcloud.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Transient solver for incompressible, turbulent flow"
        " of Newtonian fluids on a moving mesh."
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H" // chit added
    #include "initContinuityErrs.H"
    #include "createFields.H"
    #include "createUfIfPresent.H"
    #include "CourantNo.H"

    std::string dictfile;

    turbulence->validate();

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

    // if start-time > 0, read from start-time-folder for solidDict, otherwise read from case root
    if(runTime.time().value() > 0)
    {
        dictfile = mesh.time().timeName() + "/solidDict";
    }
    else
    {
        dictfile = "solidDict";
    }

    sdfibm::SolidCloud solidcloud(args.path() + "/" + dictfile, U, runTime.value()); // chit
```

```
87      solidcloud.saveState();  // write the initial condition edited by chit
88
89      Info<< "\nStarting time loop\n" << endl;
90
91      while (runTime.loop())
92      {
93          Info<< "Time = " << runTime.timeName() << nl << endl;
94      Foam::dimensionedScalar dt = runTime.deltaT();
95
96      // --- Pressure-velocity PIMPLE corrector loop
97          if(solidcloud.isOnFluid())
98      {
99          Foam::fvVectorMatrix UEqn(
100                 fvm::ddt(U)
101               + 1.5*fvc::div(phi, U) - 0.5*fvc::div(phi.oldTime(), U.oldTime())
102               ==0.5*fvm::laplacian(nu, U) + 0.5*fvc::laplacian(nu, U));
103             UEqn.solve();
104
105         phi = linearInterpolate(U) & mesh.Sf();
106         Foam::fvScalarMatrix pEqn(fvm::laplacian(p) == fvc::div(phi)/dt - fvc::div(Fs));
107             pEqn.solve();
108
109         U   = U   - dt*fvc::grad(p);
110             phi = phi - dt*fvc::snGrad(p)*mesh.magSf();
111
112             Foam::fvScalarMatrix TEqn(
113                 fvm::ddt(T)
114               + fvm::div(phi, T)
115               ==fvm::laplacian(alpha, T));
116             TEqn.solve();
117     }
118
119     solidcloud.interact(runTime.value(), dt.value());
120
121     if(solidcloud.isOnFluid())
122         {
123         U = U - Fs*dt;
124                 phi = phi - dt*(linearInterpolate(Fs) & mesh.Sf());
125
126             U.correctBoundaryConditions();
127                 adjustPhi(phi, U, p);
128
129             T = (1.0 - As)*T + Ts;
130         T.correctBoundaryConditions();
131         #include "continuityErrs.H"
132     }
133
134     solidcloud.evolve(runTime.value(), dt.value());
135     solidcloud.saveState();
136
137     if(solidcloud.isOnFluid())
138     {
139         solidcloud.fixInternal(dt.value());
140     }
141
142     if(runTime.outputTime())
143         {
144             runTime.write();
145
146             if(Foam::Pstream::master())
147             {
148                 std::string file_name;
149                 if(Foam::Pstream::parRun())
150                 {
151                     for (int i=0; i<Pstream::nProcs() ;i++)
152                     {
153                         file_name = "./processor"+ std::to_string(i) + "/" + runTime.timeName()+"/
    solidDict";
```

```
154                         solidcloud.saveRestart(file_name);
155                     }
156                 }
157                 else
158                 {
159                     file_name = "./" + runTime.timeName() + "/solidDict";
160                     solidcloud.saveRestart(file_name);
161                 }
162             }
163         }
164 
165     }
```

Index