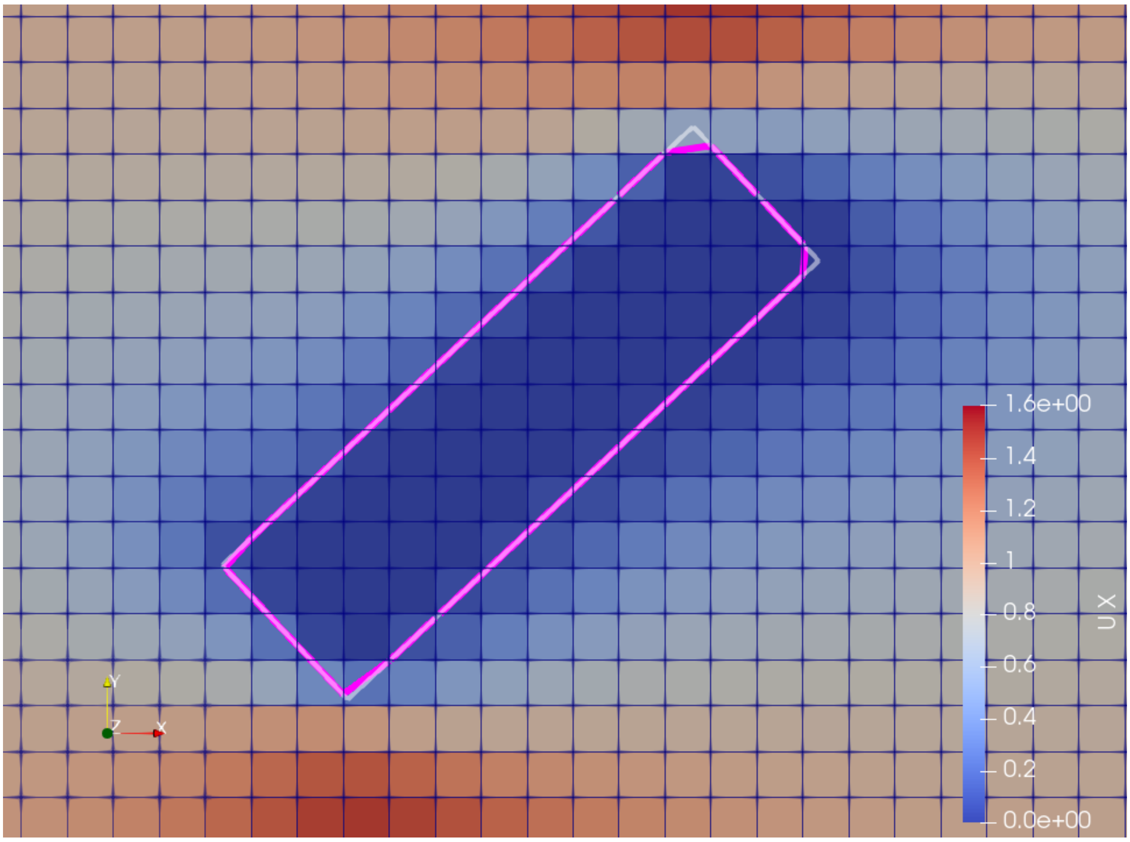


An Analysis of the Immersed Boundary Surface Method in foam-extend

Master's thesis in Applied Mechanics

JAN ERIK DÖHLER

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

An Analysis of the Immersed Boundary Surface Method in foam-extend

JAN ERIK DÖHLER

Department of Mechanics and Maritime Sciences
Division of Fluid Dynamics
Chalmers University of Technology
Gothenburg, Sweden 2022

An Analysis of the Immersed Boundary Surface Method in foam-extend
JAN ERIK DÖHLER

Supervisor: Prof. Håkan Nilsson, Mechanics and Maritime Sciences, Chalmers
Examiner: Prof. Håkan Nilsson, Mechanics and Maritime Sciences, Chalmers

Master's Thesis 2022, 2022:59
Department of Mechanics and Maritime Sciences
Division of Fluid Dynamics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Immersed Boundary cut in the uniform background mesh for a rectangular in a channel.

Typeset in LaTeX
Gothenburg, Sweden 2022

An Analysis of the Immersed Boundary Surface Method in foam-extend
JAN ERIK DÖHLER
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

The Immersed Boundary Surface method is an implementation of the Immersed Boundary method in the latest versions of `foam-extend`, a fork of the free, open-source computational fluid dynamics (CFD) software `OpenFOAM`. Instead of using body-fitted meshing methods, the Immersed Boundary method merges objects and boundaries into a uniform background mesh. While the Immersed Boundary method contains many different merging approaches, the Immersed Boundary Surface method merges objects represented by triangulated surface meshes into the background mesh in a manner similar to the cut-cell approach.

In this thesis, the implementation and limitations of the Immersed Boundary Surface method in `foam-extend 4.1 nextRelease` branch are investigated and analysed. In `foam-extend`, the Immersed Boundary method was already implemented in previous versions using polynomial fitting and based on the discrete ghost-cell approach, but was heavily modified in version 4.1. A detailed description of the newly implemented Immersed Boundary Surface method in `foam-extend 4.1 nextRelease` branch as well as a comparison to the implementation in previous `foam-extend` versions is given. The impact of using the cut-cell approach on the choice of the background mesh is shown in guidelines for mesh refinement. The limitations of the Immersed Boundary Surface method are investigated using simple test cases, focusing on the mass conservation. Furthermore, the implemented Immersed Boundary wall functions are compared to established body-fitted wall functions on different test cases.

Keywords: CFD, Immersed Boundary Method, Immersed Boundary Surface Method, OpenFOAM, foam-extend, Motion fluxes, Wall functions.

Acknowledgements

I'd like to express my deepest gratitude to my supervisor and examiner Professor Håkan Nilsson. He was not only the reason why I worked on this thesis but also a great supervisor. Despite many complications and changes of course, he has been a great help in getting back on track and a great support in every situation. In addition, it has always been a pleasure to discuss problems and widen my knowledge.

I would also like to thank Post doc Saeed Salehi and PhD student Jonathan Fahlbeck who both supported me throughout this thesis and have always been a helping hand. Especially their help in debugging and providing post-process utilities have contributed a great deal to this thesis.

Finally, I would like to thank Hrvoje Jasak for two very informative meetings and interesting discussions.

Jan Erik Döhler, Gothenburg, August 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

BF	Body-Fitted
CFD	Computational Fluid Dynamics
fe41NR	foam-extend 4.1 nextRelease branch (version 11th May 2022 14:47 commit: a6e7082d658f469434beb0f2cd4678557efb29c9)
fe41	foam-extend 4.1 master branch (version 5th July 2021 15:48 commit: 70b064d0f32604f4ce76c9c72cbdf643015a3250)
FVM	Finite Volume Method
GREAT	Operator in OpenFOAM: large number (if float GREAT=1e06 and if double GREAT=1e15)
IB	Immersed Boundary
IBM	Immersed Boundary Method (foam-extend 4.0)
IBS	Immersed Boundary Surface method (foam-extend 4.1)
PDE	Partial Differential Equations
PIMPLE	combination of PISO and SIMPLE
PISO	Pressure-Implicit with Splitting of Operators
RANS	Reynolds-Averaged Navier-Stokes
SCL	Space Conservation Law
SIMPLE	Semi-Implicit Method for Pressure Linked Equations
SMALL	Operator in OpenFOAM: small number (if float SMALL=1e-06 and if double SMALL=1e-15)
STL	STereoLithography / Standard Triangle Language / Standard Tessellation Language - file format of storing the surface geometry of 3D objects
VTK	Visualization ToolKit

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

i,j	Indices for vector notation
f	Index for cell face

Dimensionless Quantities

C_μ	Coefficient for eddy viscosity
C_f	Friction coefficient
C_i	Coefficients for the polynomial fitting in IBM with i standing for numbers 0-4
Re	Reynolds number
$y+$	Non-dimensional distance in y-direction

Greek letters

$\alpha_{\mathbf{u}}$ / $\alpha_{\mathbf{p}}$	Relaxation factor for velocity/pressure equation
δ	Dirac delta function
ϵ	Viscous dissipation
γ	Correction coefficient
μ	Kinematic viscosity
ν	Dynamic viscosity
ω	Specific dissipation
Φ	Face flux
ϕ	Flow variable
ϑ	Velocity scale

Roman letters

$\mathbf{A_u}$	Momentum matrix
$a_{ij}^{\mathbf{u}}$	Discretized momentum matrix coefficients
$\mathbf{F}(\mathbf{x})$	External force
F_1	First blending function in SST k-ω model
F_2	Second blending function in SST k-ω model
$\mathbf{f}$	Area force acting on IB

$\mathbf{H}(\mathbf{u})$	Non-dimensional contribution of momentum matrix $\mathbf{A_u}$
$\mathbf{I}$	Identity matrix
k	Turbulent kinetic energy
l	Length scale
$\mathbf{n}$	Normal vector
P	Steady-mean pressure component
$\tilde{P}_k$	Production limiter for SST k-ω model
p	Pressure
p'	Turbulent pressure component
$\mathbf{r_b}$	Explicitly treated contributions of momentum equation discretization procedure
S	Invariant measure of strain rate
S_f	Face area
$\mathbf{S}_f$	Face area vector
t	Time
$\mathbf{U}$	Steady mean velocity component
$\mathbf{u}$	Velocity
u	Velocity component in x-direction
$\mathbf{u}'$	Fluctuation velocity component
V	Volume
v	Velocity component in y-direction
w	Velocity component in z-direction

Contents

List of Acronyms **ix**

Nomenclature **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Background 1
1.2 Aim 1
1.3 Limitations 2
1.4 Specification of Issue Under Investigation 2

2 Theory **3**
2.1 Governing Equations 3
2.2 Finite Volume Method 3
2.3 Discretization of Momentum Equation 3
2.4 Pressure-Velocity Coupling 4
2.4.1 The PISO Algorithm 4
2.4.2 The SIMPLE Algorithm 5
2.4.3 The PIMPLE Algorithm in OpenFOAM 5
2.5 Turbulence Models 6
2.5.1 Reynolds-Average Navier-Stokes Equations 6
2.5.2 $k - \epsilon$ Model 8
2.5.3 SST $k - \omega$ Model 8
2.5.4 Wall Functions 9
2.6 Immersed Boundary Method 9
2.6.1 The Continuous Forcing Approach 10
2.6.2 The Discrete Forcing Approach 10
2.6.2.1 Ghost-Cell Method 11
2.6.2.2 Cut-Cell Method 12

3 Methodology **15**
3.1 The Immersed Boundary Method in foam-extend 15
3.1.1 IBM in foam-extend 4.0 15

3.1.2 The Immersed Boundary Surface Method (IBS) in foam-extend 4.1 . . . 16
3.2 The Implementation of the IBS in the foam-extend 4.1 nextRelease branch (fe41NR) . . . 19
3.2.1 Immersed Boundary Classes . . . 19
3.2.1.1 calcImmersedBoundary() - The Cutting Process . . . 21
3.2.1.2 calcCorrectedGeometry() - Manipulation and Correction . . . 24
3.2.2 Motion Fluxes . . . 25
3.2.3 Cutting Corrections . . . 26
3.2.4 Pressure and Velocity Boundary Conditions . . . 27

4 IBS Analysis **31**
4.1 Mesh Coarseness . . . 31
4.2 Motion and Mass Fluxes . . . 34
4.2.1 Stationary IB . . . 34
4.2.2 Moving IB . . . 36
4.2.3 Volume-Changing IB . . . 38
4.3 Wall Functions . . . 40
4.3.1 Test Case 1: Backward Facing Step . . . 41
4.3.2 Test Case 2: Forward Facing Step . . . 46
4.3.3 Test Case 3: Cylinder in Channel Flow . . . 51

5 Conclusion **57**

Bibliography **59**

A immersedBoundaryPolyPatch.C **I**

B immersedBoundaryFvPatch.C **XXXI**

C ImmersedCell.C **XXXVII**

D ImmersedFace.C **LI**

List of Figures

2.1 Ghost cell (red cell with center point P) together with extended stencil; inspired by [20] . . . 11
2.2 Prevention of large weighting coefficients in the ghost-cell method through image point (a) and moved immersed boundary (b); inspired by [20] . . . 12
2.3 Cut-cell method - cut cells with centre inside the solid immersed boundary get merged to neighbouring cells, inspired by [21] . . . 13

3.1 IBM method with polynomial fitting by surrounding cells in (a) and the local coordinate system for Neumann boundary condition in (b); inspired by [8] . . . 16
3.2 Cell types for the IBM in `foam-extend 4.0` on the left (a) and for IBS in `foam-extend 4.1` on the right (b), inspired by [7] . . . 17
3.3 The cut cells with new corrected cell and face centres, inspired by [7] 17

4.1 Velocity field in x-direction with STL file of IB block in channel with coarse mesh . . . 32
4.2 Cutting of a rectangular block into a coarse mesh . . . 32
4.3 Cutting of a rectangular block into a fine mesh . . . 32
4.4 Velocity field together with STL file of the two cylinders in the tutorial case `twoIbPatches` at two time steps . . . 33
4.5 Test case stationary IB: domain with stationary IB cylinder. — : wall 34
4.6 Mass flow at inlet and outlet for the stationary IB test case using fe41NR . . . 35
4.7 Mass flow at inlet and outlet for the stationary IB test case using fe41 35
4.8 Velocity vector field for the stationary IB test case using fe41NR at $t = 5s$ . . . 36
4.9 Velocity vector field for the stationary IB test case using fe41 at $t = 5s$ 36
4.10 Test case moving IB: domain with horizontal oscillating IB cylinder, amplitude = 0.5m. — : wall . . . 37
4.11 Mass flow at inlet and outlet for the moving IB test case using fe41NR 37
4.12 Mass flow at inlet and outlet for the moving IB test case using fe41 . 38
4.13 U_x velocity field for the moving IB test case using fe41NR at $t = 8.2s$ 39
4.14 U_x velocity field for the moving IB test case using fe41 at $t = 8.2s$ . . 39
4.15 Test case volume-changing IB: domain with vertical oscillating IB cylinder, amplitude = 1.0m. — : wall . . . 39

4.16 Mass flow at inlet and outlet together with the total mass change per time inside the domain for the volume-changing IB test case using fe41NR 40
4.17 Test case 1: backward facing step with $H = 0.25m$. —— : wall; - - - : symmetry plane 42
4.18 IB cutting for three different meshes in the backward facing step case 43
4.19 Residual plot for IB case with k-Omega-SST model in backward facing step case 44
4.20 Relative velocity profiles at five different locations for 6 different cases 45
4.21 Friction coefficient on the last meter before the backward facing step 46
4.22 Test case 2: forward facing step with $H = 0.25m$. —— : wall; - - - : symmetry plane 47
4.23 Residual plot for IB case with k-Omega-SST model in forward facing step case 48
4.24 Relative velocity profiles at five different locations for 4 different cases with a coarse mesh 49
4.25 Relative velocity profiles at five different locations for 2 IB cases with a fine mesh and 3 BF cases 49
4.26 Friction coefficient on the obstacle behind the forward facing step for the coarse mesh cases 50
4.27 Friction coefficient on the obstacle behind the forward facing step for fine and coarse mesh cases 50
4.28 Test case 3: cylinder in channel. —— : wall; - - - : symmetry plane . . 51
4.29 Test case 3: x-velocity contour plots after 3000 steps for 3 different IB cases and 2 BF cases 53
4.30 Test case 3: IB cutting for the three different meshes 54
4.31 Test case 3: x-velocity contour plots after 3000 steps for two IB cases with fine mesh but different inlet velocities 54
4.32 Test case 3: x-velocity contour plots after 3000 steps for a laminar IB case and a laminar BF case 55

List of Tables

3.1 Classification rules for internal and coupled boundary faces 23
3.2 Settings for basic IB conditions with zero velocity inside IB (no-slip Dirichlet and zero gradient) 29
3.3 Settings for the movingImmersedBoundaryVelocity condition with zero velocity inside IB 30

4.1 Boundary conditions for the second and third mass conservation test case 37
4.2 Numerical schemes used for all wall function test cases 42
4.3 Velocity and pressure boundary conditions for BF and IB case 42
4.4 y+ values for k-Omega-SST cases in backward facing step case at $x/H = -2$ 46
4.5 Highest y+ values for k-Epsilon cases in forward facing step case . . 47
4.6 y+ values for cylinder in channel test cases 52

1 Introduction

1.1 Background

In Computational Fluid Dynamics (CFD), the computational domain is usually decomposed by a body-fitted (BF) method into smaller cells in which the differential equations can be further approximated numerically. However, in more complex geometric domains or domains with moving/rotating solids, the decomposition can lead to high computational costs. In the case of moving or rotating bodies, the mesh of the surrounding flow field must be recalculated every few time steps, since only small deformations of cells can be compensated. In the Immersed Boundary (IB) method, the decomposition is not body fitted, but boundaries are immersed in a uniform background mesh. This has the great advantage that the background mesh is not only uniform but also constant over all time steps and only needs to be modified at the IB.

In `foam-extend`, a branch of the free, open-source CFD software `OpenFOAM`, the IB method was initially implemented based on a discrete ghost-cell forcing approach using polynomial fitting to manipulate field properties. Due to disadvantages of polynomial fitting, the implementation of the IB method was heavily modified and is now based on the cut-cell approach. Instead of using polynomials, the background mesh is cut and the discretization matrix is manipulated to satisfy the boundary condition at the immersed boundary. This new IB implementation, which is implemented in `foam-extend 4.1`, is called the Immersed Boundary Surface (IBS) method and is explained in more detail in chapter 2.6.

1.2 Aim

The aim of this work is to document and investigate the IBS implementation in the `foam-extend 4.1 nextRelease` branch. This work aims to build a basic understanding of the cutting process into the background mesh of the IB as well as the limitations for moving or volume changing, solid objects. For the analysis of the IBS, test cases need to be created that highlight the features under investigation. In addition, this work aims to contribute to the development of the IBS implementation and the preparation of the application of the IB method on rotating stators in the Francis-99 turbine.

1.3 Limitations

- Since the IBS method is still new with very little literature, the documentation of the IBS implementation is focused on the main cutting process and the two IB classes `ImmersedBoundaryFvPatch` and `ImmersedBoundaryPolyPatch`
- This work is focused on solid IBs because the future goal is the application on rotating stators in the Francis-99 turbine
- The work is based on the `foam-extend 4.1 nextRelease` branch (version 11th May 2022 14:47, commit: a6e7082d658f469434beb0f2cd4678557efb29c9
- The analysis on fluxes and wall functions is limited on a few simple test cases

1.4 Specification of Issue Under Investigation

- Differences between the IBM and the IBS
- Integration of IB implementation in `foam-extend`
- Limitations in mesh coarseness using the IBS method
- Transformation of mesh fluxes into mass fluxes
- Translating and rotating objects in the IBS method
- Boundary conditions and wall functions for the IBS method

2

Theory

2.1 Governing Equations

The system of equation for transient, incompressible viscous flow consists of the Continuity equation and the momentum equations. The Continuity equation reads

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \tag{2.1}$$

and can be simplified to

$$\nabla \cdot \mathbf{u} = 0 \tag{2.2}$$

due to constant density. The momentum equations, which are also known as the Navier-Stokes equations can be written as follows:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\frac{1}{\rho}\nabla p + \nabla \cdot (\nu \nabla \mathbf{u}) + S. \tag{2.3}$$

In the transport equations, $\mathbf{u}$ is the velocity vector, p the pressure, ν the kinematic viscosity and S a source term.

2.2 Finite Volume Method

The Finite Volume Method (FVM) is a numerical method for approximating the solution of partial differential equations (PDE). For this purpose, the computational domain is divided into finite volumes, the PDEs are linearized with discretization schemes and volume integrals of the PDE are evaluated over the finite volumes. The FVM is one of the fundamental discretization methods in `OpenFOAM` and is used for all results presented in this thesis.

2.3 Discretization of Momentum Equation

The momentum equation 2.3, simplified and semi-discretized, can be written as follows [6]:

$$\mathbf{A_u}\mathbf{u} = -\nabla p. \tag{2.4}$$

The momentum matrix $\mathbf{A_u}$ in equation 2.4 can be divided into a diagonal and a non-diagonal part. For the non-diagonal contribution in $\mathbf{A_u}$, the linear operator $\mathbf{H}$ was introduced by Jasak [6].

$$a_{ii}^{\mathbf{u}}\mathbf{u}_i = \mathbf{H}(\mathbf{u}) - \nabla p \tag{2.5}$$

$$\mathbf{H}(\mathbf{u}) = \mathbf{r_b} - \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j \tag{2.6}$$

Equation 2.5 can be rewritten for the velocity $\mathbf{u}_i$ at the centre of cell i:

$$\mathbf{u}_i = (a_{ii}^{\mathbf{u}})^{-1}\left[\mathbf{H}(\mathbf{u}) - \nabla p\right]. \tag{2.7}$$

The discretized momentum equation expressed for $\mathbf{u}_i$, substituted into the continuity equation, gives the semi-discrete pressure equation:

$$\nabla \cdot \left[(a_{ii}^{\mathbf{u}})^{-1}\nabla p\right] = \nabla \cdot \left[(a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})\right]. \tag{2.8}$$

Equation 2.8 is called the pressure equation because the continuity equation, which was a velocity equation, now only has the pressure as an unknown in the case where the operator $\mathbf{H}(\mathbf{u})$ is known. This completes the pressure-velocity coupling and the system of equation can be solved.

Using equation 2.7, the discretized equation for the face flux Φ, which is the face normal vector $\mathbf{s}_f$, with $\|\mathbf{s}_f\|$ = face area, times the velocity, can be written as follows:

$$\Phi = \mathbf{s}_f^T\mathbf{u}_f = \mathbf{s}_f\left[(a_{ii}^{\mathbf{u}})^{-1}\left(\mathbf{H}(\mathbf{u}) - \nabla p\right)\right]_f. \tag{2.9}$$

2.4 Pressure-Velocity Coupling

2.4.1 The PISO Algorithm

The PISO algorithm, Pressure-Implicit with Splitting of Operators, was originally conceived by Issa in 1986 for the pressure-velocity treatment of transient flows [4].

1. The discretized momentum equation 2.5 is solved with the pressure field of the previous time step (or guessed initial condition) to obtain a new velocity field. The velocity field is called an intermediate velocity field because only the momentum equation has been solved and not the entire system of equation. $\mathbf{H}(\mathbf{u})$ depends on the flux, which is why the flux is also taken from the previous time step to solve the momentum equation.
2. With the new intermediate velocity field, the new off-diagonal part of the momentum matrix, $\mathbf{H}(\mathbf{u})$, can be calculated and the equation 2.8 can be solved to obtain the new pressure field.
3. With the new velocity and pressure field, the face flux can be updated with equation 2.9 as well.
4. Finally, the velocity field is corrected with the new pressure and flux field, equation 2.7, before the loop starts again with step 2 until the convergence criteria is satisfied.

Although the coefficients in the linear operator $\mathbf{H}(\mathbf{u})$ depend on the flux, the matrix is only updated with the new velocity field, but the coefficients are kept constant throughout the entire correction. Therefore, the PISO algorithm is mainly used for transient flows, as the focus is on treating the pressure-velocity coupling instead of the non-linear coupling.

2.4.2 The SIMPLE Algorithm

The Semi-Implicit Method for Pressure Linked Equations, short SIMPLE, was developed by Patankar in 1972 and is mainly used for the pressure-velocity coupling in steady-state problems [16].

1. As in the PISO algorithm, the pressure field and face fluxes are taken by the previous steps, but the discretized momentum equation is manipulated with an implicit under-relaxation factor $\alpha_{\mathbf{u}}$:

$$\frac{1}{\alpha_{\mathbf{u}}} a_{ii}^{\mathbf{u}} \mathbf{u}_i + \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}} \mathbf{u}_j = \mathbf{r_b} - \nabla p^{(k-1)} + \frac{1 - \alpha_{\mathbf{u}}}{\alpha_{\mathbf{u}}} a_{ii}^{\mathbf{u}} \mathbf{u}_i^{(k-1)} \tag{2.10}$$

2. After solving equation 2.10, $\mathbf{H}(\mathbf{u})$ can be updated. With the new velocity field a pressure correction field can be calculated from equation 2.8, which will be used to compute an under-relaxed pressure field of the new time step:

$$p^{(k)} = (1 - \alpha_p) p^{(k-1)} + \alpha_p p^* \tag{2.11}$$

   with p^* as the pressure correction from the pressure equation 2.8.

3. In a next step, the face flux for the new time step is computed with equation 2.9.

4. With the corrected pressure and the newly calculated flux, the velocity field can be corrected to satisfy the continuity equation. Again the under-relaxed factor is used.

$$\mathbf{u}^{(k)} = \alpha_{\mathbf{u}} \left(\frac{1}{a_{ii}^{\mathbf{u}}} \mathbf{H}(\mathbf{u}^*) - \frac{1}{a_{ii}^{\mathbf{u}}} \nabla p \right) + (1 - \alpha_{\mathbf{u}}) \mathbf{u}^* \tag{2.12}$$

   with $\mathbf{u}^*$ from the first step and equation 2.10.

A loop is performed over all steps, as long as the tolerance of the convergence criteria is not reached.

2.4.3 The PIMPLE Algorithm in OpenFOAM

The PIMPLE algorithm is a combination of the PISO and SIMPLE method, where a loop over the momentum equation (as in SIMPLE) and over the pressure equation (as in PISO) is done. With two coefficients, `nCorrector` and `nOuterCorrector`, the number of correction loops can be chosen explicitly. The coefficient `nOuterCorrector` determines how often the entire system of equation is solved and the fields corrected (number of outer corrections), as in the SIMPLE algorithm. The coefficient `nCorrector` chooses the number of corrections of the pressure equation inside the outer loop. This means, for example, if the coefficient `nOuterCorrector` is set to one and `nCorrector` to four, the PIMPLE algorithm matches the PISO algorithm with four corrections. In the same way, but with `nOuterCorrector` greater than one and `nCorrector` equal to one, the PIMPLE algorithm becomes a SIMPLE algorithm.

2.5 Turbulence Models

Some test cases and simulations are carried out in laminar flow without considering turbulent behaviour. Turbulence models are therefore not needed in these cases. However, in order to be able to investigate the physics near walls with immersed boundaries, some turbulence models need to be introduced and explained beforehand. Hence, this section focuses on introducing the necessary models and theories to further investigate wall functions.

2.5.1 Reynolds-Average Navier-Stokes Equations

To compare the turbulent behaviour near walls between body-fitted cases and immersed boundary cases, it is sufficient to look at time-averaged properties and do not resolve turbulent fluctuations. Instead of the previously presented Navier-Stokes equations, the Reynolds-averaged Navier-Stokes equations are used together with turbulence models.

After the Reynolds decomposition, the velocity can be divided into a steady, mean part $\mathbf{U}$ and its fluctuating component $\mathbf{u}'$.

$$\mathbf{u}(\mathbf{x}, t) = \mathbf{U}(\mathbf{x}, t) + \mathbf{u}'(\mathbf{x}, t) \tag{2.13}$$

The decomposed velocity is inserted into the governing equations, which are time-averaged afterwards. The continuity equations can be simplified to:

$$\nabla \cdot (\mathbf{U} + \mathbf{u}') = \nabla \cdot \mathbf{U} + \nabla \cdot \mathbf{u}' = \nabla \cdot \mathbf{U}. \tag{2.14}$$

To simplify the time-averaged Navier-Stokes equation with decomposed velocity and decomposed pressure, the system of equation is separated into the x-, y- and z-momentum equation. The simplification is only shown for the x-momentum equation, as it can be done in the same way for the y- and z-momentum equation. The non-simplified, incompressible, time-averaged x-momentum equation is written in equation 2.15. For simplicity, the additional source term is omitted in the following equations.

$$\overline{\frac{\partial u}{\partial t}} + \overline{\nabla \cdot (u\mathbf{u})} = \overline{-\frac{1}{\rho}\frac{\partial p}{\partial x}} + \overline{\nabla \cdot (\nu \nabla u)}. \tag{2.15}$$

Each term in equation 2.15 can further be simplified [12]:

$$\overline{\frac{\partial u}{\partial t}} = \frac{\partial U}{\partial t} \tag{2.16}$$
$$\overline{\nabla \cdot (u\mathbf{u})} = \nabla \cdot (U\mathbf{U}) + \nabla \cdot (\overline{u'\mathbf{u}'}) \tag{2.17}$$
$$\overline{-\frac{1}{\rho}\frac{\partial p}{\partial x}} = -\frac{1}{\rho}\frac{\partial P}{\partial x} \tag{2.18}$$
$$\overline{\nabla \cdot (\nu \nabla u)} = \nabla \cdot (\nu \nabla U) \tag{2.19}$$

with $U = u - u'$ and $P = p - p'$. All simplifications put together, the Reynolds-averaged x-momentum equation reads:

$$\frac{\partial U}{\partial t} + \nabla \cdot (U\mathbf{U}) = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \nabla \cdot (\nu \nabla U) - \nabla \cdot (\overline{u'\mathbf{u}'}). \tag{2.20}$$

When simplifying the time-averaged convection term, a new term with fluctuating velocities, equation 2.17, appears. This term is moved to the right hand side, because it is associated with convective momentum transfer through turbulent eddies [12]. Together with the terms from the y- and z-momentum equation, these additional momentum transfer terms are called Reynolds stresses. Equation 2.20 together with the Reynolds-averaged y- and z-momentum equation gives the Reynolds-averaged Navier-Stokes (RANS) equation system:

$$\frac{\partial U}{\partial t} + \nabla \cdot (V\mathbf{U}) = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \nabla \cdot (\nu \nabla V) - \nabla \cdot (\overline{u'\mathbf{u}'}) \tag{2.21}$$
$$\frac{\partial V}{\partial t} + \nabla \cdot (V\mathbf{U}) = -\frac{1}{\rho}\frac{\partial P}{\partial y} + \nabla \cdot (\nu \nabla V) - \nabla \cdot (\overline{v'\mathbf{u}'}) \tag{2.22}$$
$$\frac{\partial W}{\partial t} + \nabla \cdot (W\mathbf{U}) = -\frac{1}{\rho}\frac{\partial P}{\partial z} + \nabla \cdot (\nu \nabla W) - \nabla \cdot (\overline{w'\mathbf{u}'}) \tag{2.23}$$

Due to new unknown terms in the RANS (the Reynolds stresses), turbulence models are needed to close the system of equation. One of the most common turbulence models is the $k - \epsilon$ model and the $k - \omega$ model, which add two additional transport equations to the three RANS equations 2.21-2.23. In the $k - \epsilon$ and $k - \omega$ model, the Boussinesq hypothesis proposed in 1877 is used, which states that the Reynolds stresses are proportional to the mean deformation rates and can be described for incompressible flows as follows:

$$-\rho\overline{\mathbf{u}' \otimes \mathbf{u}'} = \mu_t \left[\nabla\mathbf{U} + \nabla(\mathbf{U})^T\right] - \frac{2}{3}\rho k \mathbf{I} \tag{2.24}$$

with the turbulent kinetic energy per unit mass $k = \frac{1}{2}\left(\overline{(u')^2} + \overline{(v')^2} + \overline{(w')^2}\right)$ and the eddy viscosity μ_t. In the $k - \epsilon$ and $k - \omega$ model transport equations, the turbulent kinetic energy k and the rate of viscous dissipation ϵ and the rate of specific dissipation ω, respectively, are used to predict the kinetic energy k and the turbulent viscosity μ_t and finally the Reynolds stresses.

2.5.2 $k - \epsilon$ Model

The kinetic energy k and the rate of viscous dissipation ϵ are used to define velocity scale ϑ and length scale l of the large-scale turbulence [12].

$$\vartheta = k^{1/2}, \quad l = \frac{k^{3/2}}{\epsilon} \tag{2.25}$$

With the velocity and length scale, the eddy viscosity μ can be described with k and ϵ.

$$\mu_t = C\rho\vartheta l = \rho C_\mu \frac{k^2}{\epsilon}, \tag{2.26}$$

with C_μ as a dimensionless constant.

It is possible to derive exact transport equations for k and ϵ, but both have too many unknown terms and are therefore not feasible for the numerical application of the $k - \epsilon$ model. Therefore, the standard $k - \epsilon$ model by Launder and Spalding, 1974, uses simplified transport equations [11]:

$$\frac{\partial \rho k}{\partial t} + \nabla \cdot (\rho k \mathbf{U}) = \nabla \cdot \left[\frac{\mu_t}{\sigma_k}\nabla k\right] + \mu_t \left[\nabla \mathbf{U} + \nabla(\mathbf{U})^T\right] \nabla \mathbf{U} - \rho\epsilon \tag{2.27}$$

$$\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot (\rho \epsilon \mathbf{U}) = \nabla \cdot \left[\frac{\mu_t}{\sigma_\epsilon}\nabla \epsilon\right] + C_{1\epsilon}\frac{\epsilon}{k}\mu_t \left[\nabla \mathbf{U} + \nabla(\mathbf{U})^T\right] \nabla \mathbf{U} - C_{2\epsilon}\rho\frac{\epsilon^2}{k}. \tag{2.28}$$

2.5.3 SST $k - \omega$ Model

The SST $k - \omega$ model by Menter 1993 belongs to the $k - \omega$ models using the Shear Stress Transport formulation and is as the $k - \epsilon$ model a two-equation eddy-viscosity model. The $k - \omega$ SST model implementation is based on the formulation by Menter et al. from 2003, which reads [13]:

$$\frac{\partial \rho k}{\partial t} + \nabla \cdot (\rho k \mathbf{U}) = \tilde{P}_k - \beta^* \rho k \omega + \nabla \cdot [(\mu + \sigma_k \mu_t)\nabla k] \tag{2.29}$$

$$\frac{\partial \rho \omega}{\partial t} + \nabla \cdot (\rho \omega \mathbf{U}) = \alpha \rho S^2 - \beta \rho \omega^2 + \nabla \cdot [(\mu + \sigma_\omega \mu_t)\nabla \omega] + 2(1 - F_1)\rho\sigma_{\omega 2}\frac{1}{\omega}\nabla k \nabla \omega, \tag{2.30}$$

with a production limiter $\tilde{P}_k$:

$$\tilde{P}_k = min\left\{\mu_t \nabla U \left[\nabla U + \nabla(U)^T\right], 10\beta * \rho k \omega\right\}. \tag{2.31}$$

The blending function F_1 is given by

$$F_1 = tanh\left\{\left\{min\left[max\left(\frac{\sqrt{k}}{\beta^* \omega y}, \frac{500\nu}{y^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}y^2}\right]\right\}^4\right\} \tag{2.32}$$

with y as the distance to the nearest wall and

$$CD_{k\omega} = max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\nabla k \nabla \omega, 10^{-10}\right). \tag{2.33}$$

Inside the free-stream, the blending function F_1 gets equal to zero, which results in using the $k-\epsilon$ model, while inside the boundary layer it becomes one, which results in using the $k-\omega$ model approach by Wilcox.

The kinematic eddy viscosity given by:

$$\nu_t = \frac{a_1 k}{max(a_1\omega, SF_2)} \tag{2.34}$$

with S as the invariant measure of the strain rate and the second blending function F_2, which is similar to F_1 and defined as:

$$F_2 = tanh\left\{\left[max\left(\frac{2\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right)\right]^2\right\}. \tag{2.35}$$

According to Menter et al. 2003 [13] the coefficients in equation 2.29 - 2.35 are a combination of the $k-\epsilon$ and $k-\omega$ model and given as follows:

$$\beta^* = 0.09,\ \alpha_1 = \frac{5}{9},\ \beta = \frac{3}{40},\ \sigma_k = 0.85,\ \sigma_\omega = 0.5,$$
$$\alpha_2 = 0.44,\ \beta_2 = 0.0828,\ \sigma_{\omega 2} = 0.856. \tag{2.36}$$

2.5.4 Wall Functions

Wall functions are empirical equations used to model the near-wall region. Instead of using a highly refined mesh near walls to resolve the viscous-affected regions ($y+$ < 5), wall functions can be used to model the physics near the wall. This allows the use of a coarser mesh with cell heights in the range of the log-law region ($30 < y+$ < 200).

The dimensionless distance parameter $y+$, which is used to divide the near-wall region into the viscous sublayer, buffer layer and logarithmic area, is defined by:

$$y+ = \frac{u_\tau y}{\nu} \tag{2.37}$$

with the friction velocity $u_\tau = \sqrt{\tau_w/\rho}$ and the wall shear stress τ_w.

2.6 Immersed Boundary Method

The Immersed Boundary (IB) Method is a numerical approach in which solid boundaries are not accounted for by the discretized domain but by manipulating the governing equations. The solid boundaries defined in Lagrangian coordinates are immersed in the mesh defined in Eulerian coodinates. Therefore, the governing equations, which do not take into account the immersed boundaries, must be manipulated with an additional momentum forcing.

Since the manipulation of the governing equations can be done both before and after the discretization, the IBM can generally be divided into two main categories, namely the continuous and the discrete forcing approaches. Many different IB methods have been developed over the last 50 years, but in this work the focus is on the general introduction of the IB method and the theoretical background for its implementation in `foam-extend`.

2.6.1 The Continuous Forcing Approach

In the continuous forcing approach, the source term with the additional momentum forcing is added to the continuous governing equations before discretization. This is the main difference to the discrete forcing approach (explained later), where the manipulation is done after the discretization [10]. The continuous IBM approach was first introduced in 1972 by Peskin, who used the IB method to simulate the blood flow around a flexible leaflet of a human heart valve. The interaction of the blood flow with the flexible leaflet could not be adequately solved with body-fitted meshes, which was the main reason behind developing an IB method [17]. The moving boundary is replaced by a force field acting on the Cartesian background mesh to simulate the impact of the leaflet on the blood stream.

For elastic boundaries, such as the leaflet, the external forces acting on the surrounding fluid at the location of the immersed boundary can be modelled with the following equation:

$$\mathbf{F}(\mathbf{x}) = \int_B \mathbf{f}(s)\delta(\mathbf{x} - \mathbf{x}(s))da. \tag{2.38}$$

The force $\mathbf{f}(s)$ acts on the immersed boundary B and is multiplied with the Dirac delta function δ, which is zero over the entire domain except at the location of the boundary where the value of δ is one. Since the location of the immersed boundary, defined in Lagrangian approach, generally does not align with the nodal points of the Cartesian background mesh, the sharp two-dimensional impulse function becomes a smoother distribution function that affects not only the location of the boundary but also neighbouring cells in the discretized mesh. While the modelling of the moving boundary by equation 2.38 is suitable for elastic boundaries, rigid boundaries could not be treated sufficiently. Hence, the continuous forcing approach is suitable for elastic boundaries and has the advantage of being independent of the discretization method, which simplifies the numerical implementation, but leads to stiff differential equations and numerical instabilities for rigid bodies [9]. Several extensions of the continuous forcing approach, e.g. Goldstein et al. [3] or Saiki and Biringen [18], improved the modelling of rigid boundaries, but are still limited to low Reynolds number flows.

2.6.2 The Discrete Forcing Approach

In contrast to the continuous forcing approach, the momentum equation are discretized without the immersed boundaries for the discrete forcing approach and the additional momentum forcing added after the discretization. While the continuous approach was suitable for elastic boundaries, the discrete approach finds its advantages in rigid boundaries because numerical accuracy, stability and discrete conservation properties of the solver can be influenced by modifying boundary conditions in the discretized equation system [9]. In general, discrete forcing approaches can be differed into two main groups in terms of how the boundary condition is determined: the indirect forcing approach and the direct forcing approach. Since this work focuses on the implementation of the IB methods in `foam-extend`, only

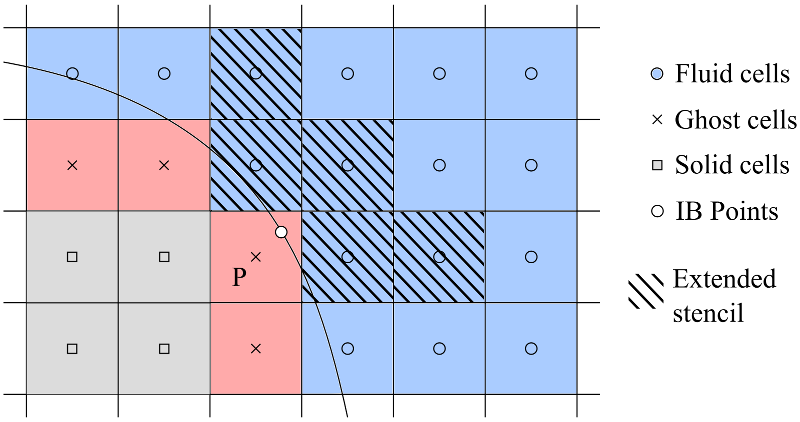


Figure 2.1: Ghost cell (red cell with center point P) together with extended stencil; inspired by [20]

two direct discrete forcing approaches are introduced on which the IB methods in `foam-extend` is based.

2.6.2.1 Ghost-Cell Method

The Ghost-cell method belongs to the discrete forcing methods because the solution is locally reconstructed at the boundary to satisfy the immersed boundary condition. As with continuous forcing approaches, the immersed boundary generally does not coincide with the nodal points and requires additional manipulations. While in the continuous approach the distribution function was smoothed over the vicinity of the immersed boundary, the discretized solution is modified for intersected cells to satisfy the boundary condition in the discrete forcing methods. Therefore, a new type of cell, called "ghost-cell", is introduced. While all cells in the physical region of the domain have their cell centre within the flow region, the ghost-cells are the first cells having their cell centre inside the solid boundary. In figure 2.1 ghost-cells are marked with an "x" as their centre, while cells in the physical region have an empty circle as their centre point.[20]

The flow variables ϕ in the centres of the ghost-cells are then manipulated and calculated with polynomials so that the field matches the boundary condition. In a two-dimensional case, the simplest approach would consist of a triangle stencil between the ghost-cell P and the two nearest fluid cells X1 and X2, figure 2.2 (a). For a linear polynomial, a Dirichlet boundary condition could be introduced through equation 2.39.

$$\phi = a_0 + a_1 x + a_2 y \tag{2.39}$$

The flow variables are therefore weighted with the neighbouring nodes X1 and X2, where x and y are the distances between P and X2 and P and X1, respectively. For the special case that the distance between the immersed boundary and a fluid cell node is comparatively small, the extrapolation can lead to large negative weighting coefficients and thus to numerical instabilities. For such cases, the polynomial fitting has to be adjusted [20]. Two possible approaches are shown in figure 2.2. The first

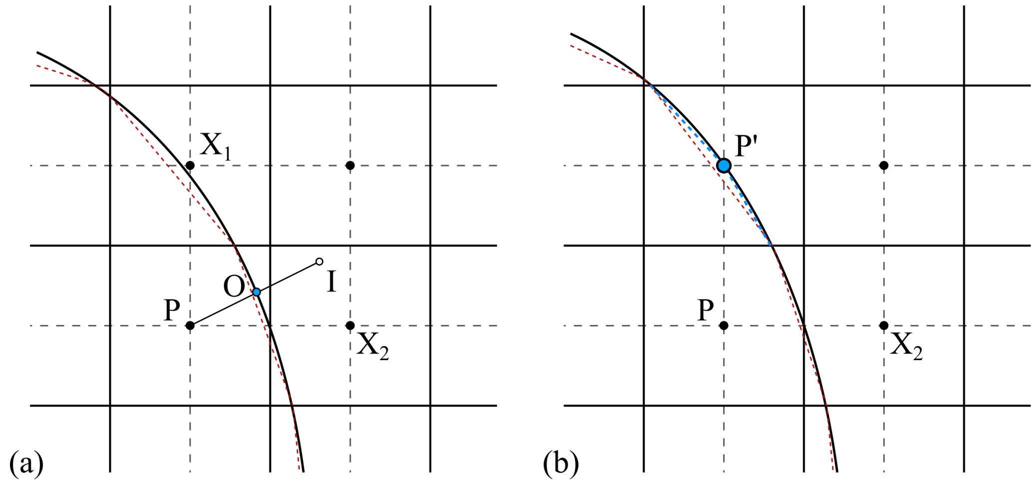


Figure 2.2: Prevention of large weighting coefficients in the ghost-cell method through image point (a) and moved immersed boundary (b); inspired by [20]

approach uses an image of the ghost-cell node through the boundary to extrapolate the flow variable. The value ϕ at P is then calculated with

$$\phi_P = 2\phi_O - \phi_I. \tag{2.40}$$

A second alternative would be to move the piecewise linear boundary to the physical node that is close to the boundary to avoid polynomial extrapolation and large negative coefficients.With this approach, if the distance between the immersed boundary and the fluid node is less than 10% of the cell size, the accuracy errors are negligible [20].

2.6.2.2 Cut-Cell Method

The cut-cell method, first published in 1986 by Clarke et al. [1] under the name Cartesian grid method, cuts intersected cells at the location of the immersed boundary to conserve mass and momentum near the IB. Cells that are intersected by the IB and whose centre lies inside the fluid domain are cut and reshaped by discarding all "dead" parts that lie inside the solid boundary. Cut parts that lie within the fluid and belonged to an intersected cell whose centre lies inside the solid, are merged with an adjacent fluid cell, as shown in figure 2.3 [21]. For the discretization of the momentum equation, mass, convective and diffusive fluxes as well as pressure gradients have to be evaluated on the cell faces. Due to the reshape of intersected cells into trapezoidal cells, this prediction is not as straight forward any longer and has to be treated specially [21]. The evaluation of fluxes on the cell faces is not of importance for the IBM in `foam-extend` and will therefore not be investigated further. Interested readers are referred to [21].

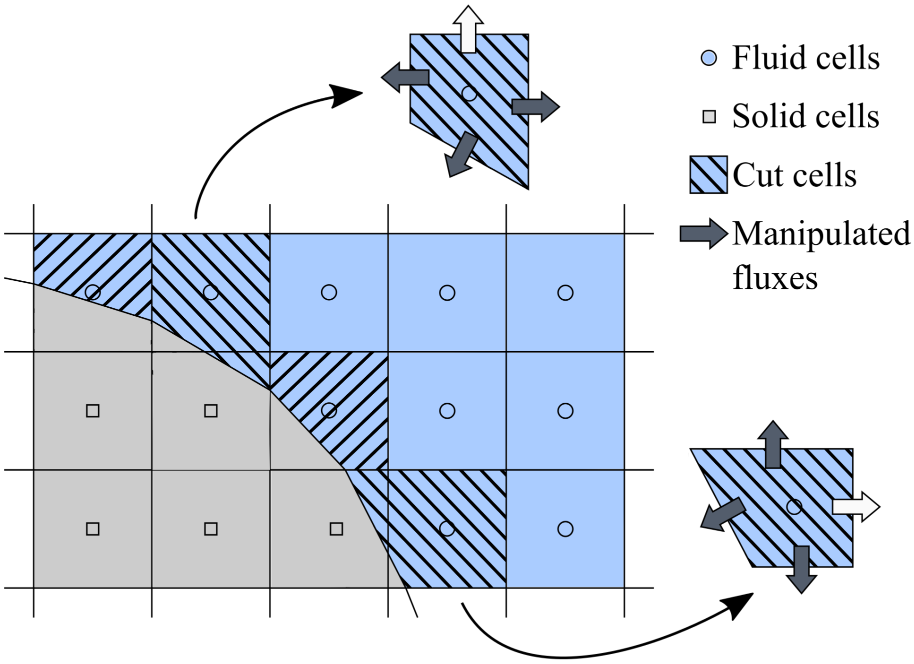


Figure 2.3: Cut-cell method - cut cells with centre inside the solid immersed boundary get merged to neighbouring cells, inspired by [21]

3

Methodology

3.1 The Immersed Boundary Method in foam-extend

In the following, the methodology and implementation of the Immersed Boundary method in the extension `foam-extend` of the open-source CFD toolbox Open Source Field Operation and Manipulation (`OpenFOAM`) is presented, which was first introduced in 2014 by Jasak et al. [5]. Since the methodology and implementation of the IB method has changed with newer releases, two different methodologies are presented, but the implementation only of the latest version.

3.1.1 IBM in foam-extend 4.0

The IB methods's first implementations in `foam extend 3.2` and `foam extend 4.0` were based on the discrete ghost-cell forcing approach with a weighted least square interpolation and called IBM [19]. In `OpenFOAM`, the Immersed Boundary surface mesh is added as an STL file (Standard Triangle Language) and immersed in a uniform Cartesian background mesh. As with the original ghost-cell method, the cells in the IBM in `foam-extend 4.0` are separated into three categories after the intersection: fluid, solid and IB cells, see figure 3.1 (a). The IB cells are the counterpart to the ghost-cells, with the only difference being that IB cells are intersected cells with their cell centre inside the fluid domain. While IB and fluid cells contribute to the solution, the discretized system of equation is not solved for the flow inside the solid cells. For the Dirichlet boundary condition of a fluid variable ϕ at the IB, a quadratic polynomial is used.

$$\begin{aligned}\phi_P = \phi_{IB} + C_0(x_P - x_{IB}) + C_1(y_P - y_{IB}) + C_2(x_P - x_{IB})(y_P - y_{IB}) \\ + C_3(x_P - x_{IB})^2 + C_4(y_P - y_{IB})^2 \end{aligned} \tag{3.1}$$

The polynomial in equation 3.1 uses the global coordinates of the IB cell centre P and the corresponding point on the immersed boundary IB, as shown in figure 3.1 (a). The coefficients C_i are calculated from the weighted least square interpolation fit of the neighbouring fluid and IB cells, shown as marked cells in figure 3.1 (a). In a local coordinate system, but in a similar manner, the Neumann boundary condition is calculated with a quadratic polynomial in the following equation.

$$\phi_P = C_0 + [\mathbf{n}_{IB} \cdot (\nabla\phi)_{IB}]x'_P + C_1 y'_P + C_2 x'_P y'_P + C_3(x'_P)^2 + C_4(y'_P)^2 \tag{3.2}$$

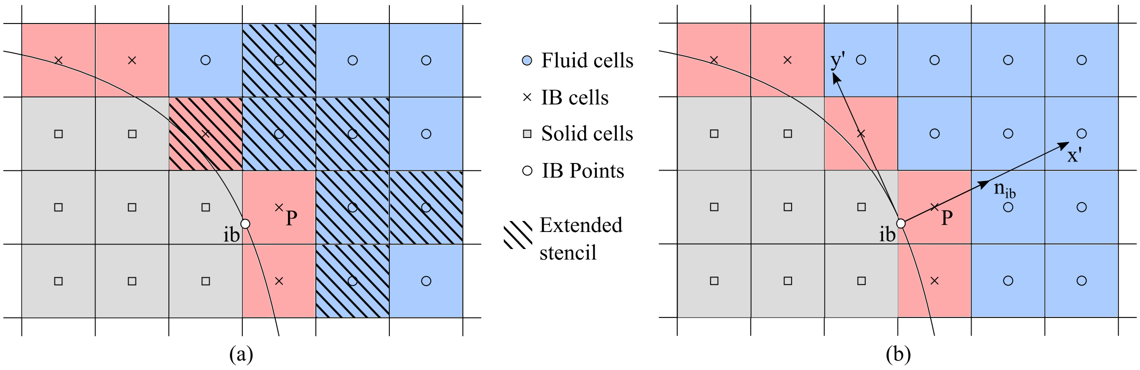


Figure 3.1: IBM method with polynomial fitting by surrounding cells in (a) and the local coordinate system for Neumann boundary condition in (b); inspired by [8]

For the solution of the momentum equation, the pressure values at the IB faces and in the IB cells are needed. After the calculation of the pressure equation, which does not require the pressure boundary condition, the Neumann boundary condition in equation 3.2 is used to calculate the pressure at the IB faces and in the IB cells. To apply for the immersed boundary, the pressure equation 2.8 must be modified and reads in discretized form:

$$\sum_f \left(\frac{1}{a_P}\right)_f \mathbf{n}_f \cdot (\nabla p)_f S_f = \sum_f \mathbf{n}_f \left(\frac{\mathbf{H}_P}{a_P}\right)_f S_f + \sum_f \mathbf{n}_{f_{ib}} \cdot \mathbf{v}_{f_{ib}} S_{f_{ib}}. \tag{3.3}$$

The letter f represents all the faces, where f_{ib} stands in particular for the faces between fluid and IB cells. $\mathbf{n}_f$ is the normal vector of the faces, S_f is the area. The velocity $\mathbf{v}_{f_{ib}}$ is calculated with equation 3.4 and scaled so that the net mass flux through the IB faces is set to zero.

$$\mathbf{v}_{f_{ib}} = \frac{1}{2}(\mathbf{v}_P + \mathbf{v}_{N_{ib}}) \tag{3.4}$$

$\mathbf{v}_P$ is here the velocity at the cell centre which lies directly outside the IB and $\mathbf{v}_{N_{ib}}$ of the first cell centre inside the IB.

3.1.2 The Immersed Boundary Surface Method (IBS) in foam-extend 4.1

In `foam-extend 4.1`, the IB approach and its implementation have been changed entirely. The new so-called Immersed Boundary Surface method (IBS) no longer resembles the ghost-cell method, but rather the cut-cell approach. Due to drawbacks with the idea of polynomial fitting in `foam-extend 3.2` and `foam-extend 4.0`, a different approach was chosen in the newer version `foam-extend 4.1`. This method is called Immersed Boundary Surface method (IBS) and is essentially based on the idea of the discrete cut-cell IB approach.

In the new IBS method, there are still three different types of cells: solid (dead) cells, intersected cells and fluid (live) cells. The difference is that not only intersected cells whose centre is in the fluid region are IB cells, but all intersected cells. This

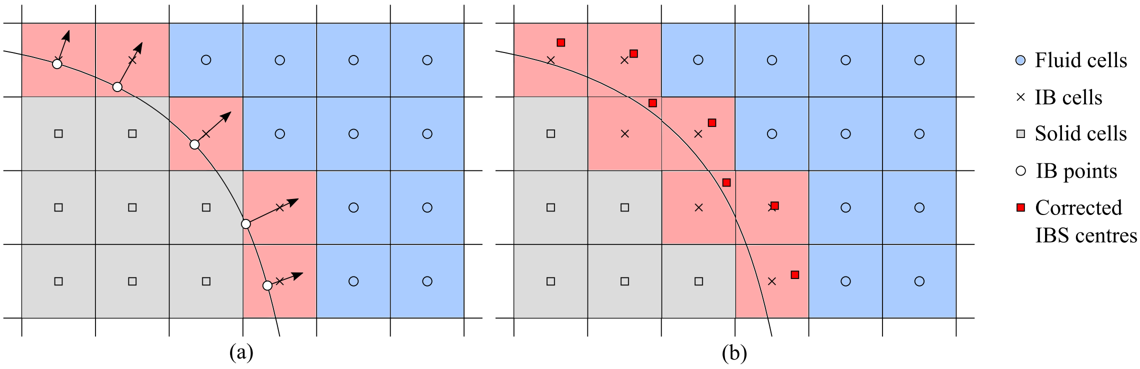


Figure 3.2: Cell types for the IBM in `foam-extend 4.0` on the left (a) and for IBS in `foam-extend 4.1` on the right (b), inspired by [7]

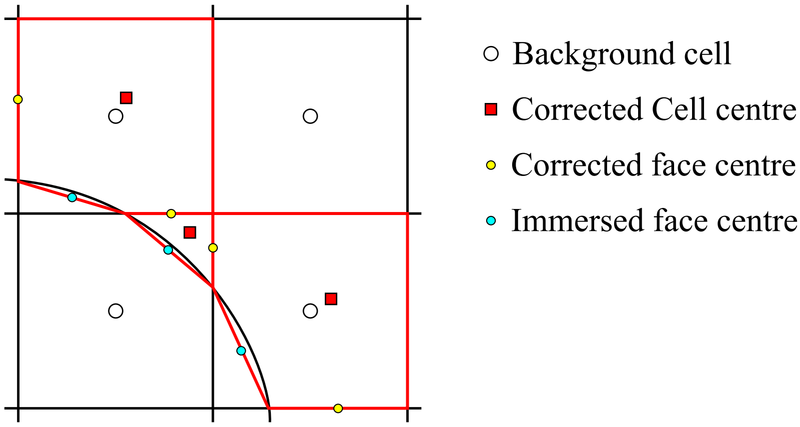


Figure 3.3: The cut cells with new corrected cell and face centres, inspired by [7]

can be clearly seen in figure 3.2. Unlike the IBM, which used polynomial fitting, the intersected cells are now cut by the IB. The cutting itself is done by a simple linear cut between the intersection points for each intersected cell. This divides the intersected cells into living and dead volumes, as well as the surfaces of theses cells into living and dead faces. The living part of the intersected cells is not added to the neighbouring fluid cell, as in the cut-cell approach in 2.6.2.2, but becomes a fluid cell on its own. Therefore, a new cell centre and cell volume must be calculated for the living part of the cells, as well as a new face area, face centre and face area vector for the cut faces and the new IB face, see figure 3.3. All dead cells and faces are excluded from the discretization matrix [7]. While all dead cells are excluded, the newly calculated geometry data of the live part of the cut-cells replaces the old data of the cut-cells. This has the advantage that the influence by the IB can be added as a usual BF boundary condition on the living part and the conventional finite volume method discretization can be used without modification [7].

In case of inaccurate intersection between the STL surface mesh and the background mesh, e.g. when the surface mesh coincides with background points or faces, geometrically open cells may exist, leading to robustness issues. To ensure closed cells after cutting, the Marooney Manoeuvre is used in `foam-extend 4.1`. For normal cells, the summation over all faces should be zero:

$$\sum_C S_f = 0. \tag{3.5}$$

In the special case of degenerated intersections equation 3.5 has to be manipulated by the Marooney Manoeuvre:

$$\sum_C \gamma_f S_f + S_{fIB} = 0 \tag{3.6}$$

The old surfaces S_f are corrected with the face correction γ_f and the corrected immersed boundary face area S_{fIB} is added to the summation. S_{fIB} can therefore be calculated with the following equation:

$$S_{fIB} = -\sum_C \gamma_f S_f. \tag{3.7}$$

Moving immersed boundary

For arbitrary moving boundaries with the velocity $\mathbf{u}_b$, the integral form of the transport equations for the moving mesh FVM can be written as in the following equation [7].

$$\int_V \frac{\partial \phi}{\partial t} dV + \oint_S \phi[\mathbf{n} \cdot (\mathbf{u} - \mathbf{u}_b)]dS - \oint_S \gamma(\mathbf{n} \cdot \nabla\phi)dS = \int_V q_v dV \tag{3.8}$$

For incompressible flows, the only difference is that the relative velocity has to be used for moving grids, which means that the solution of the transport equations for moving immersed boundaries is generally not more complicated than for static ones. However, if the conservation fluxes are calculated with the relative velocities, mass conservation, for example, cannot be automatically guaranteed. To obtain mass conservation, the space conservation law (SCL) is relied on instead, since mass conservation can be obtained by fulfilling the SCL for incompressible flows [2]. The space conservation equation can be written as:

$$\int_V \frac{\partial V}{\partial t} - \oint_S (\mathbf{n} \cdot \mathbf{u}_b)dS = 0 \tag{3.9}$$

Looking at the mass conservation in equation 3.10, it can be seen that the mass is also conserved if the space conservation applies.

$$\int_V \frac{\partial V}{\partial t} + \oint_S [\mathbf{n} \cdot (\mathbf{u} - \mathbf{u}_b)]dS = \int_V \frac{\partial V}{\partial t} - \oint_S (\mathbf{n} \cdot \mathbf{u}_b)dS + \oint_S (\mathbf{n} \cdot \mathbf{u})dS = 0 \tag{3.10}$$

The SCL in equation 3.10 can be found in the first two terms of the mass conservation equation. If the SCL is fulfilled, equation 3.10 reduces to

$$\oint_S (\mathbf{n} \cdot \mathbf{u})dS = 0. \tag{3.11}$$

Therefore, the space conservation law has to be fulfilled for the special case of moving immersed boundaries (or static IB with moving mesh) to conserve mass and prevent introduced artificial mass sources.

The space conservation equation 3.9 discretized with first-order accuracy reads:

$$\frac{V^n - V^0}{\Delta t} - \sum_f F_b = 0. \tag{3.12}$$

The first term is the linear approximation of the volume change, while the second one is the summation of the mesh motion fluxes, $F_b = \mathbf{S}_f \cdot \mathbf{u}_b$, over all faces. If the SCL in equation 3.12 for a cell is not satisfied after moving the IB, the old volume of this cell is corrected with the Motion Flux manoeuvre [7]:

$$V^0 = V^n - \Delta t \sum_f F_b > 0. \tag{3.13}$$

If the old volume cannot be corrected, since the corrected volume would be zero or negative, the new volume is corrected instead.

$$V^n = V^0 + \Delta t \sum_f F_b. \tag{3.14}$$

The reason why the old volume is manipulated and the space conservation law is not fulfilled in many cases without manipulation is that the IB usually moves further than to the next cell node in a time step. In other words, the IB movement not only changes the volume of the intersected cells, but also that of the previously intersected cells that are dead in the new time step. The volume change of the dead cells must be taken into account, which is why the newly intersected cell is enlarged in the old time step. Through this manipulation, the IB moves within a larger, fictive cell. As already stated, this manipulation is not performed on old intersected cells that are completely dry in the new time step, since V^n is zero.

3.2 The Implementation of the IBS in the foam-extend 4.1 nextRelease branch (fe41NR)

In this section, the implementation of the Immersed Boundary Surface method is analyzed. The focus is mainly on two IB classes which handle the cutting and manipulation of the discretization matrix. The entire analysis is done in the `foam-extend 4.1 nextRelease` branch, version of the 11th May 2022 14:47
(git commit: a6e7082d658f469434beb0f2cd4678557efb29c9), which will be named fe41NR.

3.2.1 Immersed Boundary Classes

In spring 2022, the `foam-extend 4.1 nextRelease` branch has the IBS method implemented such that simple static mesh solvers such as `laplacianFoam`, `simpleFoam` or `icoFoam` can include and handle immersed Boundaries. However, for dynamic meshes an additional IB solver is needed. Therefore, a solver for dynamic meshes and immersed Boundaries using the PIMPLE method, called `pimpleDyMIbFoam`, is implemented.

Regardless of which solver is used, the manipulation of the discretization matrix is done mainly in two IB classes:

- `immersedBoundaryPolyPatch`
- `immersedBoundaryFvPatch`

The implementation of immersed boundaries follows the structure of `OpenFOAM` with the structure of the "fvPatch", "polyPatch", and "primitivePatch" for the boundary discretization containing the geometric information of all boundaries [14]. In general, for the immersed boundaries, this means that the cutting and manipulation of the geometrical information is done at the level of the polyPatch data with the `immersedBoundaryPolyPatch` class, while the `immersedBoundaryFvPatch` class is the connection between the immersed boundary condition and the finite volume discretization.

Since the implementation of boundary conditions, such as the IB, is very complex, the implementation and interaction of the immersed boundary classes is only explained and analyzed with a few examples. The idea is to visualize the interactions and integration as well as possible, but the interested reader is referred to the source code of fe41NR for more information.

As already described in the theory section, the internal field and the background mesh initially know nothing about the immersed boundary. This means that the internal field is first discretized with the immersed boundary and is only manipulated when the solver loops over all boundary patches. Functions like `makeC` or `makeSf` in the class `fvMesh` are used to calculate and memorize the cell centres and face surface areas. In order to be able to use different `makeC` and `makeSf` functions explicitly for the immersed boundary, the functions in `fvMesh` have been changed to virtual functions. This change allows protected member functions of the class `immersedBoundaryFvPatch` to be called with an object of this specific class. The object of this class is the IB patch, which is one of the boundary patches that the solver iterates during the discretization process.

In the `immersedBoundaryFvPatch` class, the protected member functions `makeCf`, `makeSf`, `makeC` and `makeV` are implemented. While `makeC` and `makeV` do nothing because the cell centre and cell volume are not needed for the discretization process, the functions for the face centre, `makeCf`, and the face area, `makeSf`, manipulate the discretization matrix entries of this specific boundary patch by calling a function of a private data reference to the `immersedBoundaryPolyPatch` class. The public member function `ibPolyPatch()` returns the private data `ibPolyPatch_`, which is a reference to the immersed boundary patch implemented in the `immersedBoundaryPolyPatch` class. Further details of the manipulation and cutting process in the `immersedBoundaryPolyPatch` class are explained in detail below.

As already explained, functions are implemented in the `immersedBoundaryFvPatch` class that return a polyPatch data from the `immersedBoundaryPolyPatch` class. One of these is, for example, the function `ibPolyPatch()`, which returns the data ibPolyPatch_. On this immersedBoundaryPolyPatch object, a public member function called ibPatch() of the class immersedBoundaryPolyPatch is executed before calling another function faceCentres() which returns the face centres of the calculated immersed boundary patch, see source code below.

Listing 3.1: immersedBoundaryFvPatch.C (Appendix B)

```
// * * * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * * * //

void Foam::immersedBoundaryFvPatch::makeCf(slicedSurfaceVectorField& Cf) const
{
    // Insert the patch data for the immersed boundary
    // Note: use the face centres from the stand-alone patch within the IB
    // HJ, 30/Nov/2017
    // Inserting only local data
    Cf.boundaryField()[index()].UList::operator=
    (
        ibPolyPatch().ibPatch().faceCentres()
    );
}
```

The function ibPatch() triggers the function calcImmersedBoundary(), one of two main implemented functions for the cutting process inside the immersedBoundaryPolyPatch class. The function ibPatch() is just one of many functions that can trigger the cutting process, but only does so when the cutting process is not yet done and pointers not yet active. Since it is crucial for optimizing a code to save unnecessary computational processes, it is important to perform the cutting operations as infrequently as possible. In addition, the returned pointers point to the location of the data that is being asked for.

In the following, the two main functions calcImmersedBoundary() and calcCorrectedGeometry() of the class immersedBoundaryPolyPatch are explained below. The calcImmersedBoundary() function generally executes the cutting process, while the calcCorrectedGeometry() function modifies the vector and scalar fields of the IB cells and faces and corrects cutting errors with the Marooney Manoeuvre.

3.2.1.1 calcImmersedBoundary() - The Cutting Process

After defining references for the mesh and it's geometry data within the function calcImmersedBoundary(), a small comment in the code summarizes very well what the most important steps are within this function.

Listing 3.2: immersedBoundaryPolyPatch.C (Appendix A)

```
    // Algorithm
    // Initialise the search by marking the inside points using calcInside
    // Based on inside points addressing, check intersected faces and cells
    // For all intersected cells, calculate the actual intersection and
    // - calculate the (cell) intersection face, its centre, and area vector
    // - adjust the cell volume and centre
    // - adjust the face area and face centre
```

First, the intersected cells and faces are detected and the actual intersection is calculated. Then a new centre point and cell volume are calculated for all intersected cells and a new centre point and area for all intersected faces. The newly calculated vector and scalar fields are then stored to be retrieved by the function `calcCorrectedGeometry()`.

To identify which cells and faces are intersected by the immersed boundary, all points that lie inside the triangular STL surface are marked, appendix A line 186-239. The idea behind this is first to separate all points into inner and outer points, second to divide all cells into wet, dry and cut cells by checking the property of all points inside a cell, and finally to divide all faces into wet, dry and cut faces as well. After all points have been separated into inner and outer points, a loop is executed over all cells of the background mesh. Within this loop, each cell is assigned to wet-cells, dry-cells or cut-cells, depending on the classification of the points - inner only, outer only or both. For the cut-cells, it is also checked whether the nearest triangle of the STL patch is within the bounding box of this cell, to exclude unexpected cases due to bad STL files or other reasons, appendix A line 241-271. In the latter case, the neighbouring cells are checked, appendix A line 277-347.

After classifying all cells, the actual cutting is performed by creating an object `cutCell` from the class `ImmersedCell`.

Listing 3.3: immersedBoundaryPolyPatch.C (Appendix A)

```
// Calculate the intersection
ImmersedCell<triSurfaceDistance> cutCell
(
    cellI,
    mesh,
    dist
);
```

How the linear cutting in cut-cells is implemented in the file `ImmersedCell.H` and `ImmersedCell.C` is not shown here. It should only be mentioned that the cut-cell is divided into two sub-cells and the wet-cell data is returned, as explained in section 3.1.2. Also, the intersection is limited to a maximum of two faces protruding the surface of the immersed boundary. This is a limitation that is investigated in section 3.2.3.

The cutting data returned by the `ImmersedCell` class to the `cutCell` object is then stored in various lists, scalar and vector fields. In order to store the data correctly, the procedure is implemented differently depending on the type of the cut. Besides the regular cut, which means that faces of the intersected cell have been cut, appendix A line 430-473, the intersection between the immersed boundary and the background mesh can lie exactly on a face of the background mesh. This results in no cell cut at this specific location and the intersection is called direct face cut. Nevertheless, at least one face and two points are saved, and either the neighbour or the owner of that face is marked as wet or dry respectively, appendix A line 478-575. For the direct face cut, the special case of coupled boundaries, as for processor boundaries, has to be taken into account, appendix A line 577-717. After all points and faces are stored, duplicate points and faces are removed to optimise memory usage, and a stand-alone patch, or in other words a `primitivePatch`, of the cell intersected immersed boundary faces is created.

Table 3.1: Classification rules for internal and coupled boundary faces

Owner	Neighbour WET	Neighbour DRY	Neighbour CUT
WET	WET	DRY	WET
DRY	DRY	DRY	DRY
CUT	WET	DRY	CUT

Listing 3.4: immersedBoundaryPolyPatch.C (Appendix A)

```
// Build stand-alone patch
// Memory management
{
    unmergedPoints.shrink();

    pointField ibPatchPoints;
    labelList pointMap;

    mergePoints
    (
        unmergedPoints,
        1e-6,                 // mergeTol.  Review.  Do not like the algorithm
        false,                // verbose
        pointMap,
        ibPatchPoints
    );

    // Renumber faces after point merge
    faceList ibPatchFaces(unmergedFaces.size());

    forAll (unmergedFaces, faceI)
    {
        // Get old and new face
        const face& uFace = unmergedFaces[faceI];
        face& rFace = ibPatchFaces[faceI];
        rFace.setSize(uFace.size());
        forAll (uFace, pointI)
        {
            rFace[pointI] = pointMap[uFace[pointI]];
        }
    }

    // Create IB patch from renumbered points and faces
    ibPatchPtr_ = new standAlonePatch(ibPatchFaces, ibPatchPoints);
```

The stand-alone patch data is written to VTK files at each output-time. A list of all dry-cells (dead-cells) and a variable with the amount of dry-cells are then created, appendix A line 797-829.

In the first part of the function `calcImmersedBoundary()`, the actual cutting and classifying of all cells is done. In addition, a `primitivePatch` is created and a new IB face added during the cutting. In the second part, the function takes care of all faces and their classification. All faces are assigned to wet, dry or cut faces, as with cells. To classify the faces, the classes of the owner and neighbour cells of their faces must be checked. The rules in table 3.1 are implemented for all internal and coupled boundary faces.

Example: If a face has a wet-cell as owner and a cut-cell as neighbour, the face is a wet-face. Since it does not matter for the classification whether it is owner or neighbour, the table is symmetrical. The implemented rules from table 3.1 are checked inside the function for internal and boundary faces, but only internal and coupled boundary faces have an owner and a neighbour cell. Ordinary boundary faces do not have a neighbour cell, which means that the face is assigned to the same class as the owner cell. First, all wet- and dry-faces are detected, appendix A line 847-1047, and afterwards all the faces that are still unknown are checked for intersection with inner and outer points, appendix A line 1050-1151.

If a face falls under both cases, points that are on the inside and points that are on the outside, the face is cut with the class `ImmersedFace`, and a so-called `cutFace` object with two sub-faces is created, appendix A line 1112-1118. This object `cutFace` is then used to report the modified properties to vector and scalar fields for immersed boundary faces.

Listing 3.5: immersedBoundaryPolyPatch.C (Appendix A)

```
1130 // Real intesection.  Check cut. Rejection on thin cut is
1131 // performed by ImmersedFace.  HJ, 13/Mar/2019
1132 const scalar faceFactor =
1133     cutFace.wetAreaMag()/mag(S[faceI]);
1134
1135 // True intersection.  Collect data
1136 intersectedFace[faceI] = immersedPoly::CUT;
1137
1138 // Get intersected face index
1139 ibFaces[nIbFaces] = faceI;
1140
1141 // Get wet centre
1142 ibFaceCentres[nIbFaces] = cutFace.wetAreaCentre();
1143
1144 // Get wet area, preserving original normal direction
1145 ibFaceAreas[nIbFaces] = faceFactor*S[faceI];
```

The last step of the function `calcImmersedBoundary()` is to store the amount of dead faces and cell number corresponding to the dead faces, appendix A line 1160-1191. At the end of this function, the cutting process has been carried out and all cells and faces have been assigned as wet, dry or cut. The properties of IB cells and faces have been changed, but the cutting has not yet been corrected nor have the geometry fields. These two steps are performed in the following function `calcCorrectedGeometry()`.

3.2.1.2 calcCorrectedGeometry() - Manipulation and Correction

As explained before, the main idea of this function is to correct unwanted cuttings and manipulate the `polyMesh` geometry fields.

At the beginning of the function `calcCorrectedGeometry()` new reference variables for the mesh geometry of the `polyMesh` are defined. The important detail is that it is not just a reference to the constant `polyMesh` scalar and vector fields, but that these fields are also made mutable.

Listing 3.6: immersedBoundaryPolyPatch.C (Appendix A)

```
1231 // Get mesh reference
1232 const polyMesh& mesh = boundaryMesh().mesh();
1233
1234 // Get mesh geometry from polyMesh.  It will be modified
1235 vectorField& C =
1236     const_cast<vectorField&>(boundaryMesh().mesh().cellCentres());
```

```

vectorField& Cf =
    const_cast<vectorField&>(boundaryMesh().mesh().faceCentres());

scalarField& V =
    const_cast<scalarField&>(boundaryMesh().mesh().cellVolumes());

vectorField& Sf =
    const_cast<vectorField&>(boundaryMesh().mesh().faceAreas());
```

This allows the mesh geometry to be modified with the recalculated cell and face data from function `calcImmersedBoundary()`, appendix A line 1248-1299. For all dead cells, the volume is multiplied by a very small number, as is the face area for all dead faces.

Listing 3.7: immersedBoundaryPolyPatch.C (Appendix A)

```
forAll (dc, dcI)
{
    // Scale dead volume to small
    V[dc[dcI]] *= SMALL;
}
```

Listing 3.8: immersedBoundaryPolyPatch.C (Appendix A)

```
forAll (df, dfI)
{
    // Scale dead area to small
    Sf[df[dfI]] *= SMALL;
}
```

By this manipulation, the non-diagonal components of all dead cells in the discretization matrix approach zero and thus have no effect on live cells. To avoid very small diagonal values and to achieve zero velocity inside the IB, for the special case of zero flow inside, the diagonal of all dead cells is multiplied by a very large value (in OpenFOAM implemented through GREAT). This second manipulation is done in the function `setDeadValues` in the class `immersedBoundaryFieldBase`, which is called from every IB condition, as `mixedIB`.

In the last part of this function, the Marooney Manoeuvre for open cells is implemented, which is explained in section 3.1.2. For the Marooney Manoeuvre, a default threshold of the name `closedThreshold_` is used, which is set to 1e-6 in the file `primitiveMeshCheck.C`.

3.2.2 Motion Fluxes

Besides the manipulation of the discretization matrix, the computation of the motion fluxes, the correction of the delta coefficient and the non-orthogonal correction vectors are implemented in `immersedBoundaryFvPatch.C`, appendix B. The Motion flux manoeuvre is implemented in B line 235-269.

Due to a limited time budget for this work, it is not possible to go into more detail about the implementation of the computation of motion fluxes with immersed boundaries and the interested reader is referred to the source code of in appendix B. On the other hand, a brief analysis of motion fluxes is given later in section 4.2.

3.2.3 Cutting Corrections

In the previous section it is said that the linear cutting process in `ImmersedCell.C` and `ImmersedFace.C` is not explained in detail. However, in order to better understand why some cells or faces are not cut even though they intersect with the IB, cutting limitations are further investigated.

In order to have a diagonal dominant discretization matrix and thus a stable solver, the mesh quality is important for most of CFD solvers. When the uniform background mesh is cut by the IB, uniformity and aspect-ratio close to unity are no longer guaranteed and high non-diagonal values may appear in the discretization matrix $\mathbf{A_u}$. Therefore, the cutting process must be limited and very small faces and cells corrected. The limitations are checked in `ImmersedCell.C` and in `ImmersedFace.C`.

In `ImmersedCell.C` and in `ImmersedFace.C` it is first checked whether the cell or face cut is significant, appendix C line 493-525 and appendix D line 321-369. For this check, the distance between each point of the respective cell or face and the IB is evaluated. If one of the following if-statements is true, all points are classified as wet or dry respectively.

$$\text{if:} \quad max(h) < TOL \quad \Rightarrow \text{All points are wet} \tag{3.15}$$

$$\text{if:} \quad min(h) > -TOL \quad \Rightarrow \text{All points are dry} \tag{3.16}$$

h is the depth of each point to the STL surface and is negative if the point lies outside the STL (inside the flow field). The absolute tolerance TOL is computed by multiplying the shortest edge of the cell or face by a tolerance factor `immersedPoly::tolerance()` of the class `immersedPoly` defined as 1e-4.

$$TOL = min(edgeLength) \cdot 10^{-4} \tag{3.17}$$

If 3.15 and 3.16 are not fulfilled, the cut is significant and valid.

After the cut is initiated, the distance to the STL is checked again in a different way in both `ImmersedCell.C` and `ImmersedFace.C`. The intersection point between STL and background mesh on each edge should not be too close to the start or end point of the corresponding edge. Therefore, four if-statements are implemented.

Listing 3.9: ImmersedCell.C (Appendix C)

```
104 if
105 (
106     depth_[start]*depth_[end] < 0
107  && edgeLength > SMALL
108  && mag(depth_[start]) > edgeLength*immersedPoly::tolerance_()
109  && mag(depth_[end]) > edgeLength*immersedPoly::tolerance_()
110 )
```

This check is implemented in appendix C line 104-110 and appendix D line 78-84 and uses the tolerance factor `immersedPoly::tolerance()`=1e-4.

If all three statements are fulfilled, in other words the intersection point is not too close to the corners, the cutting continues with the newly calculated intersection point. If one of these statements is not fulfilled, the corner point is used as intersection point instead of the newly calculated point. Then the cutting process is carried out in ImmersedCell.C and the sub-faces for the wet- and dry-parts are created in ImmersedFace.C. At the end of the cutting process in ImmersedCell.C, output messages are implemented to inform whether the newly calculated sub-volumes and sub-areas either have a negative volume/area or are larger than the uncut volume/-face. If such a bad cut, with negative volume/area or unreasonable values occurs, the info "Bad cell cut" is written to the log file for cells, appendix D line 795-830, and "Bad cell face cut" for faces, appendix D line 685-727.

3.2.4 Pressure and Velocity Boundary Conditions

According to the standard BF boundary conditions for walls, three basic boundary conditions are implemented for IB:

- fixedValueIb
- zeroGradientIb
- mixedIb

These three boundary conditions allow either a Dirichlet, a Neumann or a mixed boundary condition to be used. As can be seen in the code, the fixedValueIb boundary condition is the Dirichlet condition for immersed boundaries and uses the fixedValue condition used in BF methods.

Listing 3.10: fixedValueIbFvPatchField.C

```
template<class Type>
Foam::fixedValueIbFvPatchField<Type>::fixedValueIbFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<Type>(p, iF),   // Do not read mixed data
    immersedBoundaryFieldBase<Type>
    (
        p,
        Switch(dict.lookup("setDeadValue")),
        pTraits<Type>(dict.lookup("deadValue"))
    ),
    triValue_("triValue", dict, this->ibPatch().ibMesh().size())
{
    // Since patch does not read a dictionary, the patch type needs to be read
    // manually.  HJ, 6/Sep/2018
    this->readPatchType(dict);

    if (!isType<immersedBoundaryFvPatch>(p))
    {
        FatalIOErrorInFunction(dict)
            << "\n    patch type '" << p.type()
            << "' not constraint type '" << typeName << "'"
            << "\n    for patch " << p.name()
            << " of field " << this->dimensionedInternalField().name()
            << " in file " << this->dimensionedInternalField().objectPath()
            << exit(FatalIOError);
    }

    // Re-interpolate the data related to immersed boundary
    this->updateIbValues();
```

```
    fixedValueFvPatchField<Type>::evaluate();
}
```

Listing 3.11: fixedValueIbFvPatchField.C

```
218 template<class Type>
219 void Foam::fixedValueIbFvPatchField<Type>::evaluate
220 (
221     const Pstream::commsTypes
222 )
223 {
224     this->updateIbValues();
225
226     // Set dead value
227     this->setDeadValues(*this);
228
229     // Evaluate fixed value condition
230     fixedValueFvPatchField<Type>::evaluate();
231 }
```

In the main constructor of the `fixedValueIb` boundary condition, two main functions are called, the private member function `updteIbValue()` and public member function `evaluate()`. The function `evaluate()` is found in most of the boundary conditions. Therefore, this function is shown for all three boundary conditions to roughly illustrate their implementation.

The function `evaluate()` in line 218-230 of the file `fixedValueIbFvPatchField.C` is basically an extension of the boundary condition `fixedValue`, since the function `evaluate()` of the `fixedValue` boundary condition is called at the end in line 230. In addition to the `evaluate()` function of the class `fixedValueFvPatchField`, the IB boundary condition must take care of the values at the IB surface and the internal field. The private member function `updateIbValues()` interpolates the values from the triangular surface and can trigger the IB cutting, which is explained in section 3.2.1.1. The internal field values are set in `setDeadValues(intField)`.

Listing 3.12: fixedValueIbFvPatchField.C

```
template<class Type>
void Foam::fixedValueIbFvPatchField<Type>::updateIbValues()
{
    // Interpolate the values from tri surface using nearest triangle
    const labelList& nt = this->ibPatch().ibPolyPatch().nearestTri();

    Field<Type>::operator=(Field<Type>(triValue_, nt));
}
```

The same procedure can be seen for the `zeroGradientIb` and the `mixedIb` boundary conditions. Again, the functions `updateIbValues()` and `evaluate()` are called in the constructor, and the latter is an extension of the function `evaluate()` of the `zeroGradientFvPatchField` and `mixedFvPatchField` class respectively.

The use of all three boundary conditions for a typical no-slip Dirichlet boundary condition or a zero-gradient boundary condition with zero flow inside the IB is shown in table 3.2.

While the keywords `type` and `patchType` specify which boundary condition is used, `triValue` and `triGradient` specify the values of the Dirichlet and Neumann boundary condition. Both are used with the boundary condition `mixedIb` and therefore a

Table 3.2: Settings for basic IB conditions with zero velocity inside IB (no-slip Dirichlet and zero gradient)

keywords	Boundary conditions		
type	fixedValueIb	zeroGradientIb	mixedIb
patchType	immersedBoundary	immersedBoundary	immersedBoundary
triValue	uniform (0, 0, 0)	-	uniform (0, 0, 0)
triGradient	-	-	uniform (0, 0, 0)
triValueFraction	-	-	uniform 1
setDeadValue	yes	yes	yes
deadValue	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
value	uniform (0, 0, 0)	uniform (0, 0, 0)	uniform (0, 0, 0)

third parameter, `triValueFraction`, is required to specify whether the Dirichlet or Neumann condition is used. For the `zeroGradientIb` boundary condition, none of these parameters are required because it is a zero gradient condition. For all three boundary conditions, it must be specified whether a constant and uniform value is given inside the IB and if so, what value.

For the special case of a moving IB, none of the above boundary conditions can be used. The transfer from mesh motion fluxes into mass fluxes is not covered by the basic boundary condition and requires special treatment. Therefore, a fourth boundary condition is implemented for immersed boundaries.

- `movingImmersedBoundaryVelocity`

The `movingImmersedBoundaryVelocity` boundary condition is a fixed value condition for moving immersed boundaries. In the function `evaluate()` of the class `movingImmersedBoundaryVelocityFvPatchVectorField`, the function `evaluate()` of the class `fixedValueFvPatchVectorField` is called to set a no-slip Dirichlet boundary condition. As with the basic IB boundary conditions, the values inside the IB are set in the function `evaluate()`.

Listing 3.13: movingImmersedBoundaryVelocityFvPatchVectorField.C

```
void Foam::movingImmersedBoundaryVelocityFvPatchVectorField::evaluate
(
    const Pstream::commsTypes
)
{
    // Set dead value
    this->setDeadValues(*this);

    // Evaluate mixed condition
    fixedValueFvPatchVectorField::evaluate();
}
```

Unlike the other basic boundary conditions, the boundary velocity of the IB must be taken into account in the function `updateIbValues()` and an additional function is needed to take into account the mesh motion fluxes. This function is called `updateCoeffs()` and is an extension of the function `updateCoeffs()` of the class `fixedValueFvPatchVectorField`.

Table 3.3: Settings for the movingImmersedBoundaryVelocity condition with zero velocity inside IB

keywords	Boundary condition
type	movingImmersedBoundaryVelocity
patchType	immersedBoundary
setDeadValue	yes
deadValue	(0, 0, 0)
value	uniform (0, 0, 0)

Listing 3.14: movingImmersedBoundaryVelocityFvPatchVectorField.C

```
void Foam::movingImmersedBoundaryVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvMesh& mesh = dimensionedInternalField().mesh();

    if (mesh.changing())
    {
        const fvPatch& p = patch();

        // Get wall-parallel mesh motion velocity from immersed boundary
        vectorField Up = this->ibPatch().ibPolyPatch().motionDistance()/
            mesh.time().deltaT().value();

        const volVectorField& U =
            mesh.lookupObject<volVectorField>
            (
                dimensionedInternalField().name()
            );

        scalarField phip =
            p.patchField<surfaceScalarField, scalar>(fvc::meshPhi(U));

        // Warning: cannot use patch normal but the real face normal
        // THEY MAY NOT BE THE SAME!  HJ, 28/Mar/2019
        vectorField n = p.Sf()/(p.magSf());

        const scalarField& magSf = p.magSf();
        scalarField Un = phip/(magSf + VSMALL);

        // Adjust for surface-normal mesh motion flux
        vectorField::operator=(Up + n*(Un - (n & Up)));
    }

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

For the use of the `movingImmersedBoundaryVelocity` boundary condition only the dead values keywords have to be specified, table 3.3.

The implementation of turbulence boundary conditions and wall functions is not explained here, but a comparison between IB and BF wall function is presented in section 4.3.

4

IBS Analysis

4.1 Mesh Coarseness

For the Immersed Boundary Surface method, the mesh quality is a very important component. Assuming an STL file with sufficient quality, the mesh quality defines how well the IB is cut into the background mesh. Cells that are too large compared to the size of the IB can lead to large differences in geometry or even wrong cuts.

In the function `calcImmersedBoundary()` of the class `immersedBoundaryPolyPatch`, where the cutting process is implemented, all points which lie inside the triangular surface are marked, appendix A [185-233], as prescribed earlier. If no point is marked, the entire cutting process is skipped and no IB is merged into the background mesh. Hence, the cells of the background mesh must not be so large that no mesh point lies within the IB. If this special case occurs, as in figure 4.1, the velocity field is calculated as for the case that there is no object at all inside the channel flow. In figure 4.1 the STL file of the IB is marked white to illustrate the size compared to the cells. The velocity field in x-direction is constant because the background mesh is not aware of the IB.

Other problems occur when the points of the background mesh lie within the surface mesh of the IB, but the cells are still too large, so that more than one cut per cell have to be made. The IBS in `foam-extend 4.1` is implemented with the limitation of one cut per cell and boundary patch. In other words, only two intersection points between the STL and the background mesh per cell are allowed to define a clear cut. If there are more than two intersection points per cell, it is not clear between which points the linear cutting must be performed. This leads to wrong immersed boundaries and should be avoided. Such a case can be seen in figure 4.2. On the left side in figure 4.2a it can be seen that not only the top and bottom of the block are not cut at all, but also that the cutting in the middle cell is done incorrectly. The velocity in x-direction is even negative because the cut object is open on the right side and does not represent a rectangular with closed walls. A much better representation of the latter case can be seen in figure 4.3. The mesh is fine enough that neither a cell has more than one cut per cell, nor does the volume of the immersed object differs much from the STL. In figure 4.3b, the grey highlighted intersection surfaces are not as high as the black surrounding box of the STL, because this is a quasi-2D case and the immersed block is higher than the domain. In addition, small gaps can be seen in the walls of the cut faces in figure 4.3b, which are due to cutting correction (direct face cut), as explained in section 3.2.3. When looking at the IB tutorial

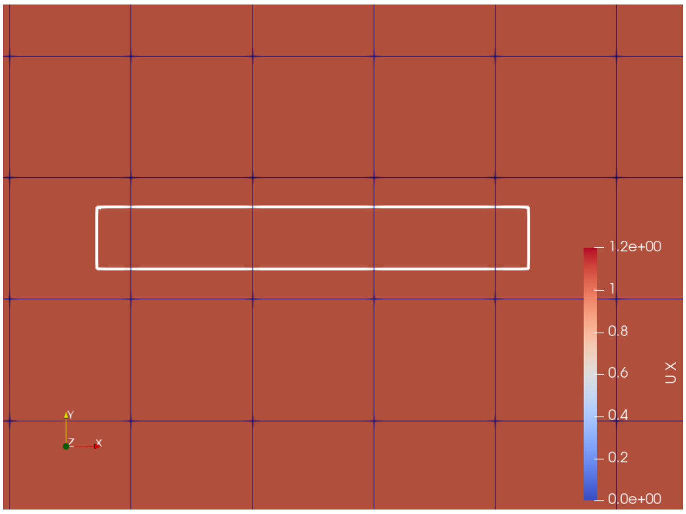
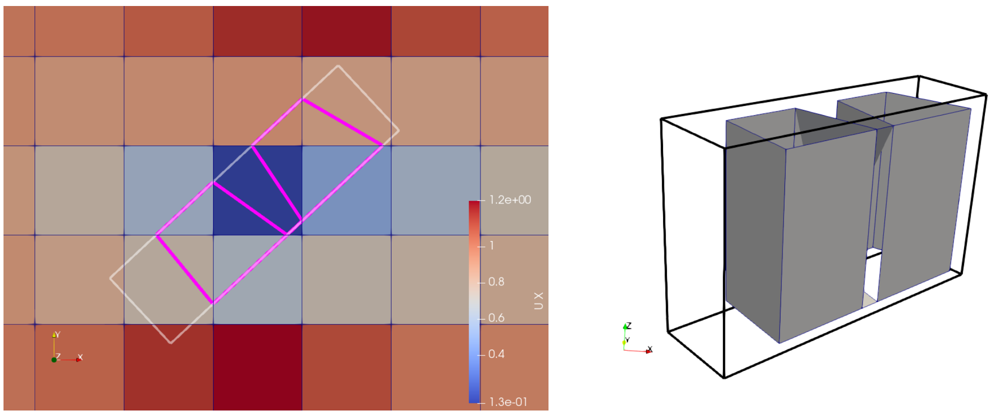


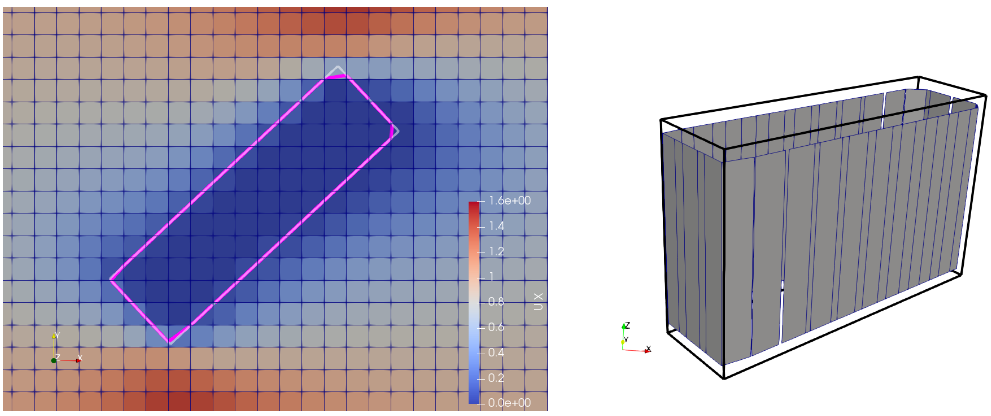
Figure 4.1: Velocity field in x-direction with STL file of IB block in channel with coarse mesh



(a) U_x field with STL file in white and actual cutting in magenta

(b) Surface plot of cut faces inside the block of the STL file

Figure 4.2: Cutting of a rectangular block into a coarse mesh



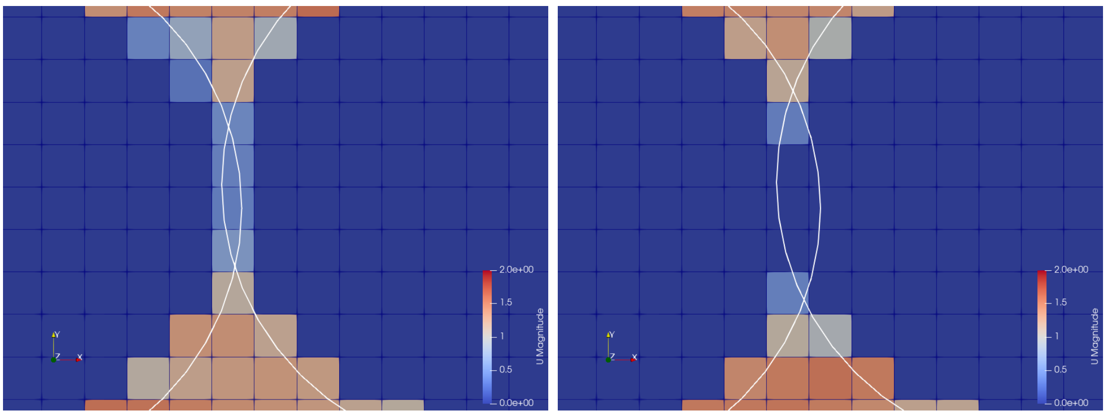
(a) U_x field with STL file in white and actual cutting in magenta

(b) Surface plot of cut faces inside the block of the STL file

Figure 4.3: Cutting of a rectangular block into a fine mesh

case `twoIbPatches`, where two cylinders move in one channel, another important behaviour of IB with implications for mesh refinement becomes clear. In the tutorial case, one cylinder oscillates in x-direction while the other oscillates in y-direction. After only 0.18 seconds, the cylinders touch and slide over each other. Since the IBs are not solid walls but only triangular surfaces, they cannot collide in the usual sense but merge into one large solid object. For the computational domain, it does not matter whether a points lies in one or two solid IB. Collisions are therefore not yet possible in `foam-extend`.

But before the two cylinders touch or "collide", the special case of two IB cuttings in one cell occurs, which was already explained above. The difference this time is that one cell is cut by two different patches and the cut, made for each patch separately, is unaware of the other cut. Therefore, the cell is cut twice, the cell parameters manipulated and changed twice and the second cut overwrites the first. Figure 4.4a shows the magnitude of the velocity field between the two cylinders at time $t = 0.18s$. Theoretically, the two cylinders should have already collided and the velocity between them should be zero. However, since the IBS method cannot handle two intersections in one cell at the same time, the cells between the cylinders are not completely dry. In order for two cylinders to touch, the STLs must overlap by one cell so that the cells are for at least one patch within an STL, as can be seen in figure 4.4b. This in turn results in the requirement for a sufficient fine mesh so that neither the overlap nor the distance between the two IB patches is too large.



(a) Time = 0.18s

(b) Time = 0.20s

Figure 4.4: Velocity field together with STL file of the two cylinders in the tutorial case `twoIbPatches` at two time steps

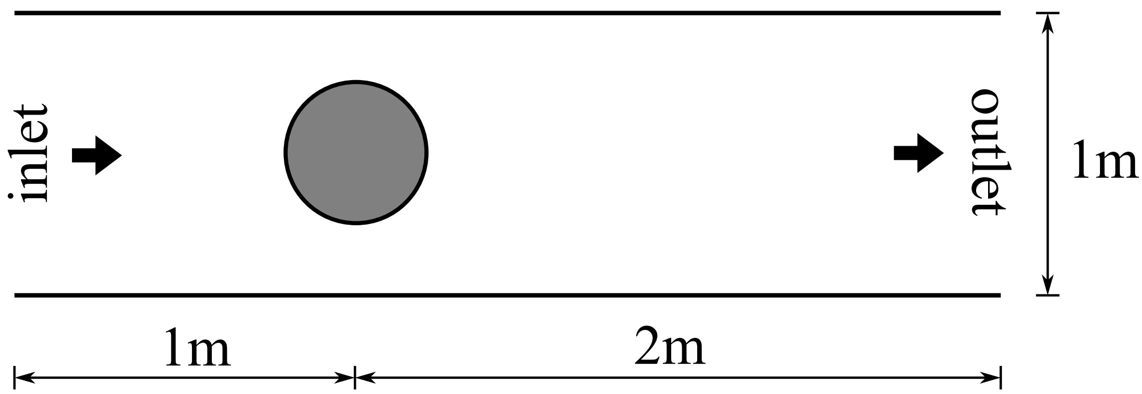


Figure 4.5: Test case stationary IB: domain with stationary IB cylinder. — : wall

4.2 Motion and Mass Fluxes

In this section, the IBS method is checked for mass conservation. To find out whether the IBS method is mass conserving, three different cases were used to analyze fluxes on stationary, moving and volume-changing IBs. To minimize the preparation time and simplify the cases, the IB tutorial case `movingCylinderInChannelTurbulent` was used for all three situations with only minor modifications. The fluid, used for these test cases, is water with a density of $\rho = 1kg/m^3$.

In this section, a second version of `foam-extend 4.1` is introduced for the analysis of the motion and mass fluxes. The second version is the master branch and latest updated on 5th July 2021 15:48
(git commit: 70b064d0f32604f4ce76c9c72cbdf643015a3250). This version will be named fe41.

4.2.1 Stationary IB

In the first mass conservation test case, the IB tutorial case
`movingCylinderInChannelTurbulent` is taken without changes, except that the cylinder does not move. At the inlet, water enters the channel with an inlet velocity of $1m/s$. The IB cylinder has a solid wall and the water leaves the domain on the right side at the outlet, see figure 4.5. To account for a stationary IB, the oscillatory motion of the cylinder inside the dynamicMeshDict was removed and the velocity boundary condition changed from movingImmersedBoundaryVelocity to mixedIb with the Dirichlet condition of zero velocity at the boundary. The domain is discretized with 75x25 cells and has a width of $0.08m$.

Due to incompressible flow and the stationary IB with constant volume, the absolute value of the inflow and outflow of the domain should be exactly the same over time. As figure 4.6 shows, the mass flow at the inlet is constant and $0.08m^3/s$, as it should be, but the mass flow at the outlet does not match the inflow. Since the mass flux through the IB is zero, the mass is not conserved using fe41NR.

The same test case was calculated using fe41 and the results are shown in figure 4.7. Here the expected results of constant and equal inflow and outflow are achieved.

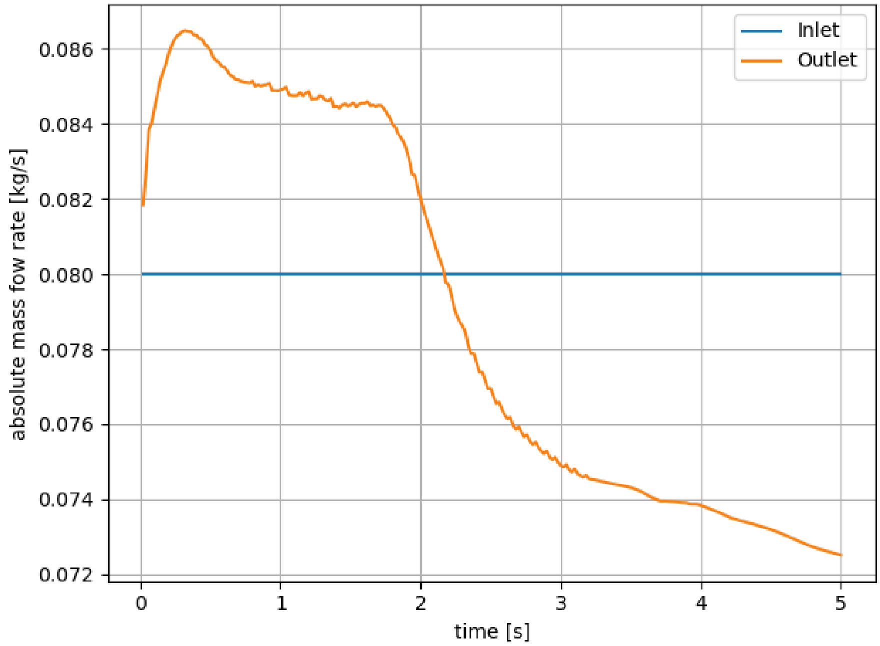


Figure 4.6: Mass flow at inlet and outlet for the stationary IB test case using fe41NR

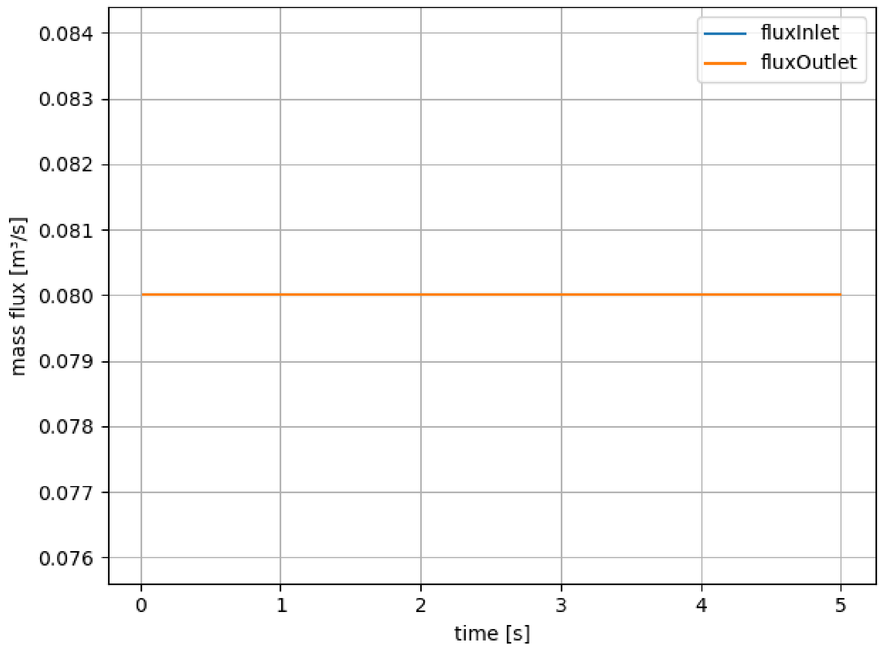


Figure 4.7: Mass flow at inlet and outlet for the stationary IB test case using fe41

Differences between the two `foam-extend` versions can also be seen in figure 4.8 and 4.9. While the vector field for fe41 looks realistic in figure 4.9, the flow seems to go into the cylinder when calculated with fe41NR, which would explain the lower outflow after 5 seconds in figure 4.6. However, this is contradicted by the fact that the summarized mass flow rate `phi` around the cylinder is zero.

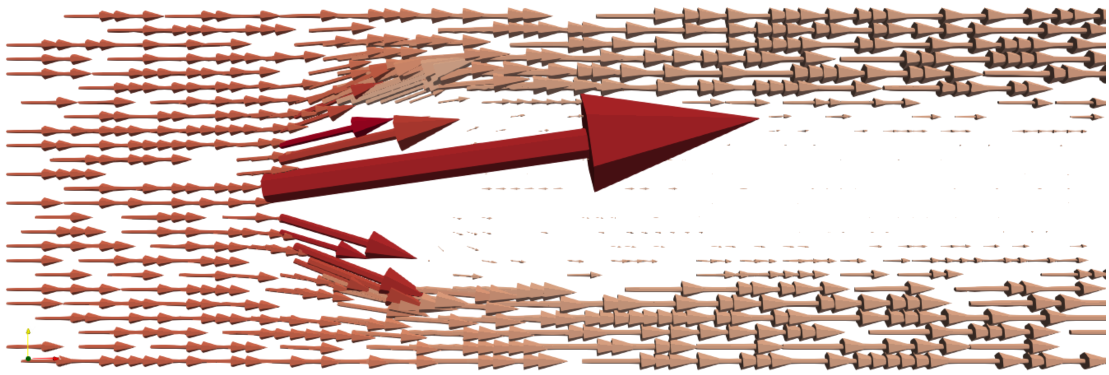


Figure 4.8: Velocity vector field for the stationary IB test case using fe41NR at $t = 5s$

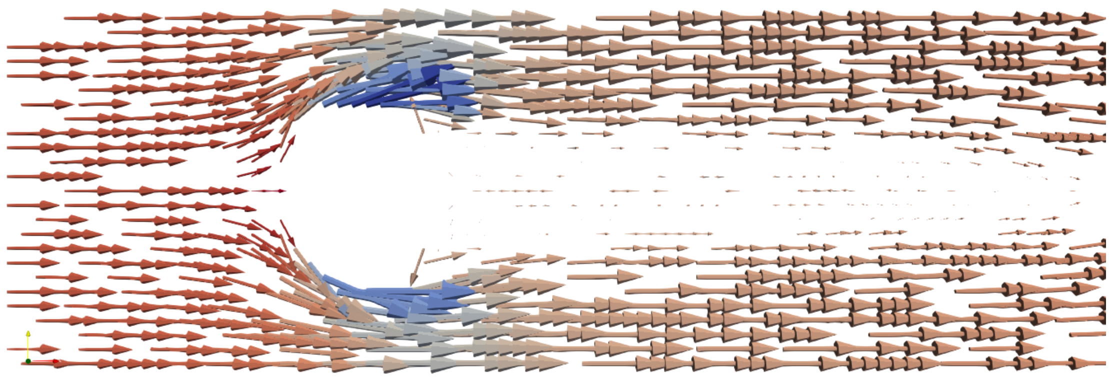


Figure 4.9: Velocity vector field for the stationary IB test case using fe41 at $t = 5s$

4.2.2 Moving IB

In the second mass conservation test case, the IB cylinder oscillates horizontally inside the channel with an amplitude of $0.5m$. The boundary conditions for the inlet and outlet are changed so that the velocity inside the channel is almost zero. The in- and outlet are defined with the pressureInletOutletVelocity condition so that the fluid can flow into and out of the domain respectively, see table 4.1. This means that the flow in and out of the domain is only due to the movements of the IB. Therefore, the domain was slightly reduced to increase the impact of the IB on the fluid inside the channel. Again, the test case was calculated using fe41NR and fe41.

Despite the differences between the two `foam-extend` version, there are a number of similarities. First, both versions are not mass conserving. Since positive mass

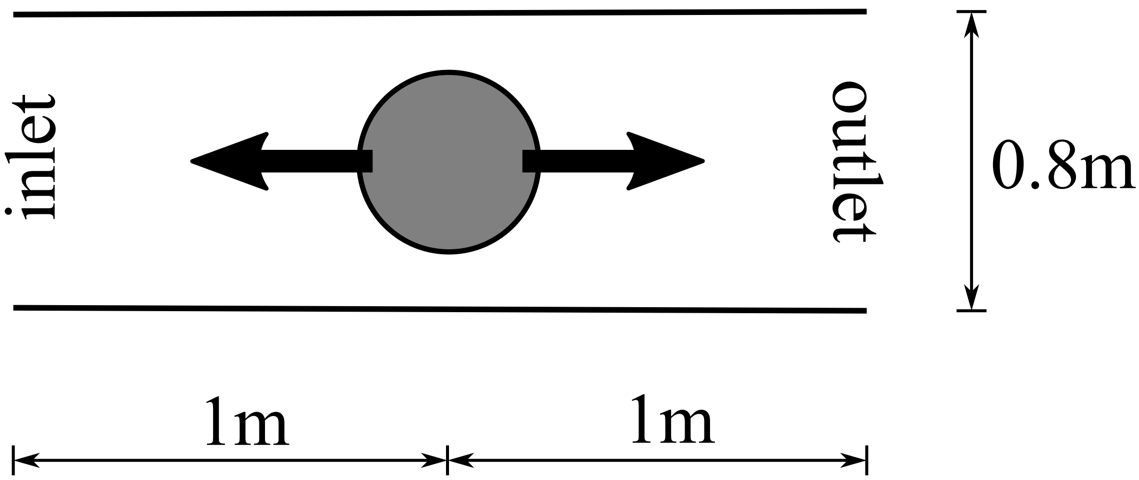


Figure 4.10: Test case moving IB: domain with horizontal oscillating IB cylinder, amplitude = 0.5m. — : wall

Table 4.1: Boundary conditions for the second and third mass conservation test case

patch	U	p
Inlet	type pressureInletOutletVelocity value uniform (0 0 0)	type totalPressure p0 uniform 0 gamma 0 value uniform 0
Outlet	type pressureInletOutletVelocity value uniform (0 0 0)	type fixedValue value uniform 1e-5

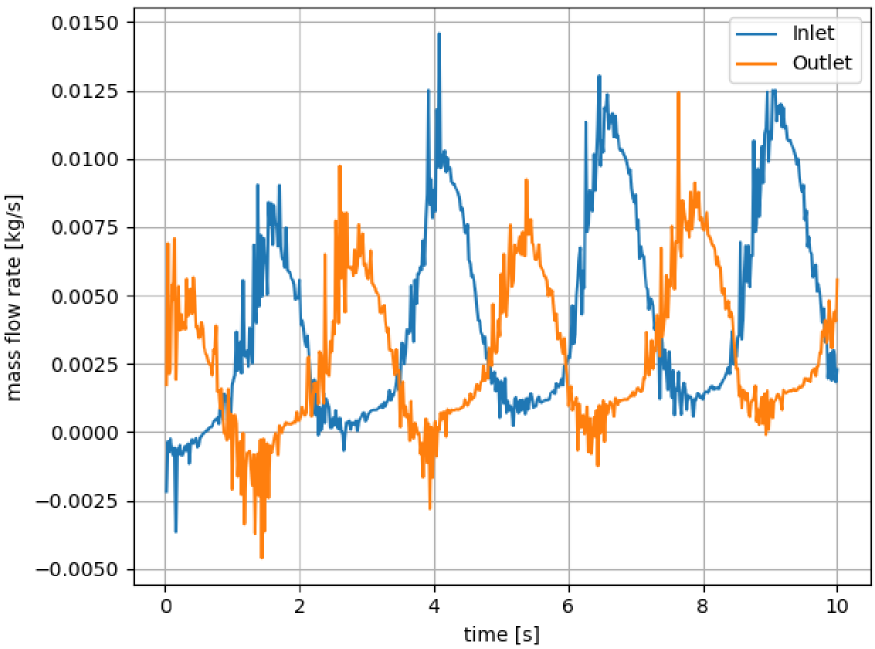


Figure 4.11: Mass flow at inlet and outlet for the moving IB test case using fe41NR

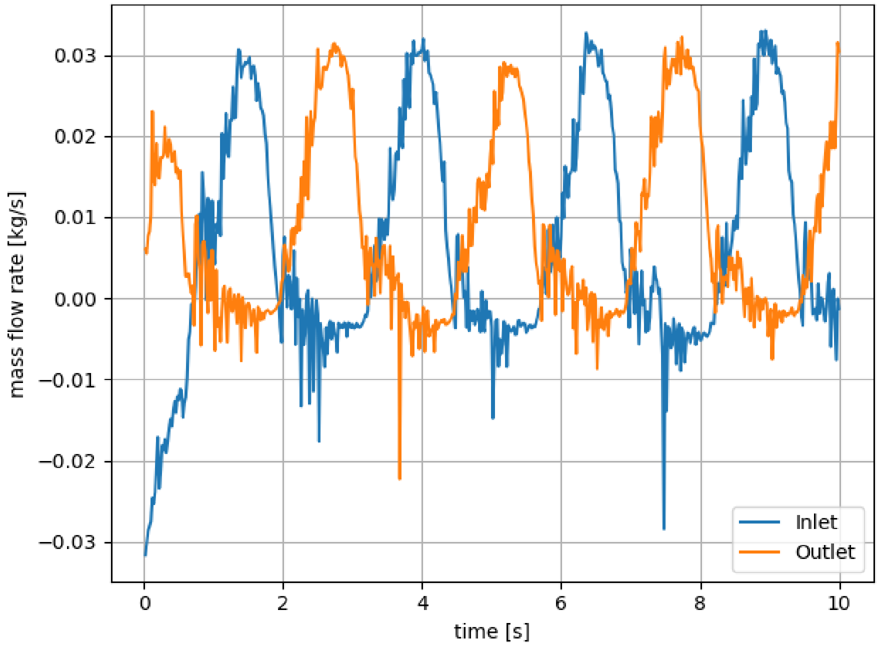


Figure 4.12: Mass flow at inlet and outlet for the moving IB test case using fe41

flow rate means that fluid is going out of the domain and both inflow and outflow are positive most of the time, the total volume inside the domain should decrease, as the volume of the IB does not change noticeably. The IB obviously pushes the fluid out of the domain but behind the IB the fluid is not sucked back in. This can also be seen in figure 4.13, where the fluid velocity behind the IB is almost zero. In figure 4.14, the fluid behind the IB shows a little more of the expected behaviour, but even in this case there is almost no inflow into the domain, see figure 4.12.

In addition to the similarities mentioned, the results of the two different versions differ greatly in the magnitude of the fluid velocity and mass flow rate, which can be seen in the magnitude of the mass flow rate in the figures 4.11 and 4.12, and in the different colours in the figures 4.13 and 4.14. Nevertheless, both versions are not mass conserving.

4.2.3 Volume-Changing IB

Although it is to be expected that the mass is not conserved even with volume-changing IBs, this case is presented to highlight the previous findings. The results are shown using only the `foam-extend` version fe41NR.

This thirds test case is similar to the second test case with the difference that the IB cylinder oscillates vertically with an amplitude of $1m$. As the channel is $0.8m$ high, the cylinder will exit and enter the domain during the movement. This changes the total volume of the IB, simulating a volume-changing IB.

In the figure 4.16, the mass flow rate at the inlet and outlet is presented together with the mass change per time of the entire domain. The mass change per time is

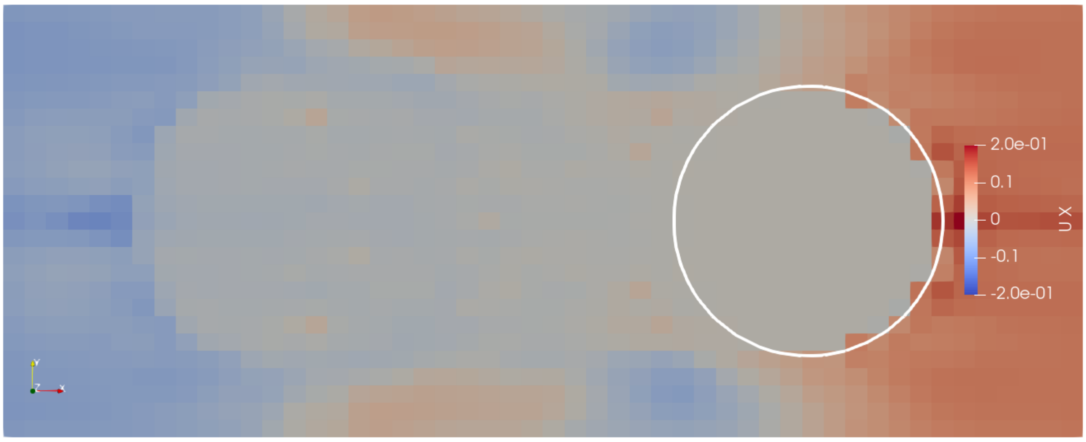


Figure 4.13: U_x velocity field for the moving IB test case using fe41NR at $t = 8.2s$

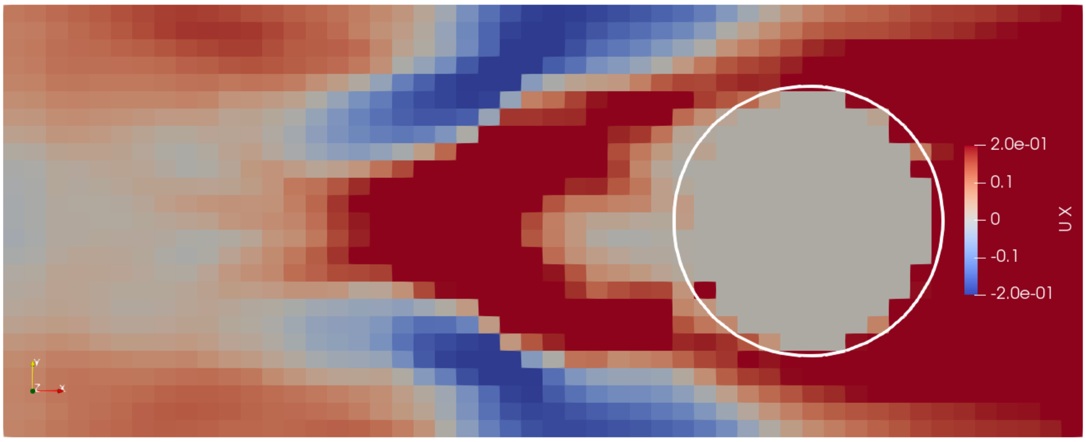


Figure 4.14: U_x velocity field for the moving IB test case using fe41 at $t = 8.2s$

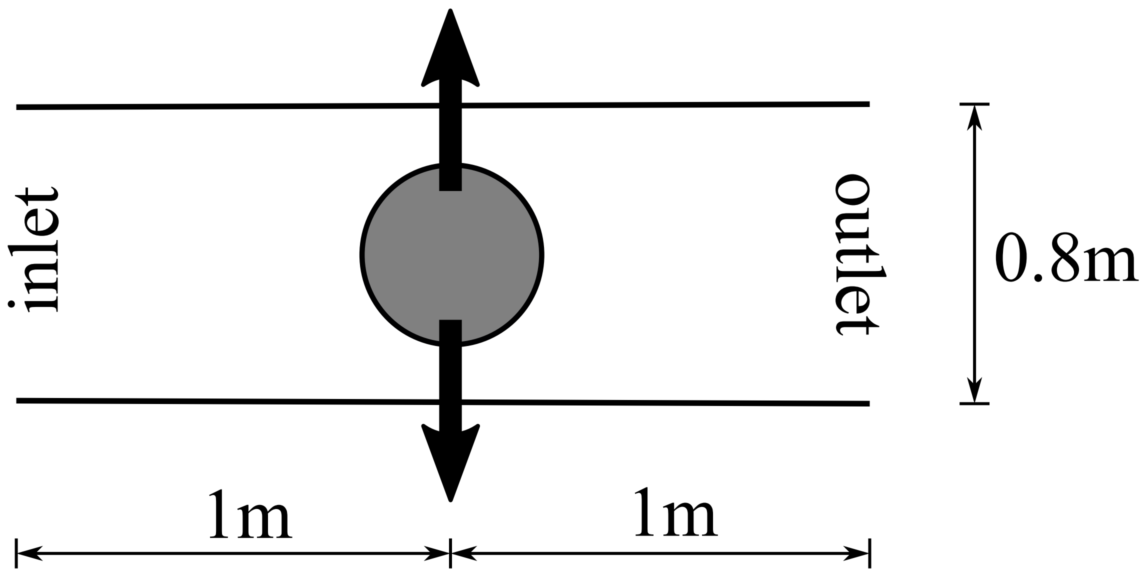


Figure 4.15: Test case volume-changing IB: domain with vertical oscillating IB cylinder, amplitude = 1.0m. — : wall

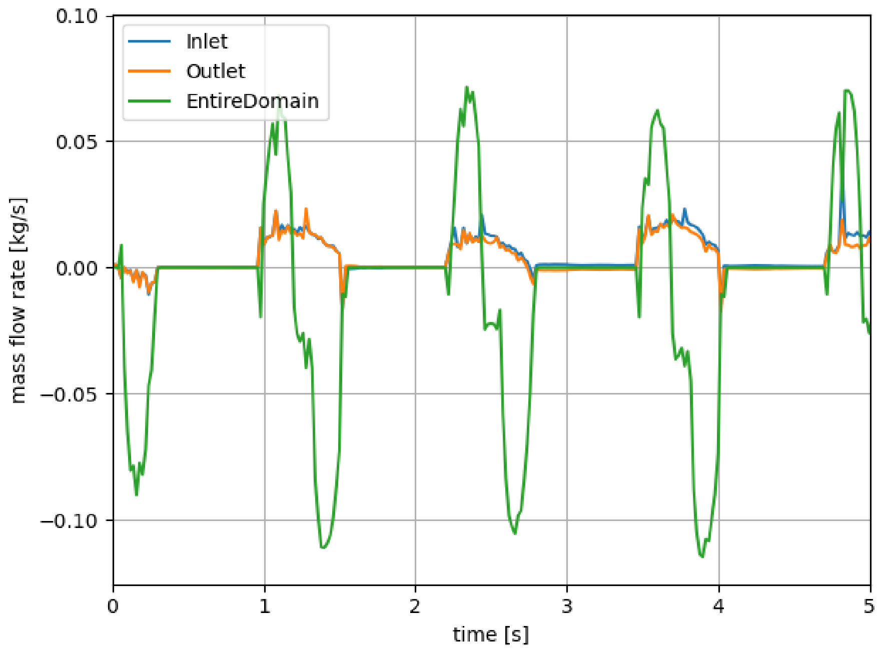


Figure 4.16: Mass flow at inlet and outlet together with the total mass change per time inside the domain for the volume-changing IB test case using fe41NR

calculated as follows:

$$\dot{m} = \frac{\rho[V_{ib}(t_n) - V_{ib}(t_{n-1})]}{\Delta t} - (\dot{m}_{inlet} + \dot{m}_{outlet}). \tag{4.1}$$

As in the second test case, the mass flow rate through the inlet and outlet is predominantly positive. Again, the fluid is pushed out of the domain but not back in. When the mass change inside the entire domain is positive, the cylinder enters the domain and the fluid is not pushed out of the domain to the same extent as the IB volume increases. During the short period when the cylinder is completely inside the domain and the IB volume does not change, the green line in figure 4.16 is almost horizontal but negative because the mass flow rate at the inlet and outlet is positive. The peaks in the negative values appear when the cylinder leaves the domain and the IB volume becomes smaller. Since the mass change of the domain, green curve, is not entirely zero, the total mass inside the volume is not conserved, which underlines the findings from the previous two test cases. It should be mentioned again that the flux through the IB is zero.

4.3 Wall Functions

For laminar cases, the boundary conditions for the velocity and pressure field described in section 3.2.4 are sufficient. For turbulent cases, on the other hand, additional boundary conditions with wall functions are required. Since IB methods have the advantage of being able to use uniform background meshes, refinements at the IB to solve the near-wall region and having the first cells inside the viscous sublayer would negate this advantage. Therefore, instead of mesh refinement, wall functions can be used to model the near-wall region at immersed boundaries.

So far, one immersed boundary wall function class has been implemented for each turbulence parameter.

- immersedBoundaryEpsilonWallFunction
- immersedBoundaryKqRWallFunction
- immersedBoundaryNutWallFunction
- immersedBoundaryOmegaWallFunctions

These wall functions are based on the corresponding BF wall functions and are compared with the BF boundary conditions in test cases below. Due to time constraints, a detailed description of the IB wall functions is not given here and the interested reader is referred to the source code of fe41NR.

In the following, the IB wall functions are compared with the BF wall functions for three different two-dimensional test cases. The first two test cases, a backward facing step and a forward facing step in a channel, are simple and well known cases that make it easy to use the same grid and geometry for IB and BF methods. A final test case examines the flow around a cylinder in a channel, which is a much more complex case for the implemented wall functions. In all three test cases, no experimental or validation data is used. Furthermore, for all cases the steady-state solver `simpleFoam` was used, which uses the SIMPLE algorithm and is suitable for incompressible turbulent flows. The numerical schemes ware taken from the IB tutorial case `pitzDailyTurbulent` and are listed in the table 4.2.

4.3.1 Test Case 1: Backward Facing Step

The first test case is a slightly modified backward facing step case. The flow entering through the velocity-driven inlet follows a channel for 30 meters before reaching the backward step. After 30 meters, corresponding to $40D = 40 \cdot 0.75m = 30m$, the flow should be fully developed [15]. Behind the backward facing step, the channel has a symmetry plane at the bottom instead of another wall, see figure 4.17. The flow inside the channel has a constant mass inflow and the kinematic viscosity $\nu = 10^{-6}$.

The boundary conditions of the test cases are given in table 4.3. For k, nut, epsilon and omega the previously mentioned special IB wall functions are used, while for the BF cases the corresponding wall functions `epsilonWallFunction`, `kqRWallFunction`, `nutWallFunction` and `omegaWallFunction` are used.

Ideally, the STL would lie directly on the background mesh and no cutting would be required. This would result in exactly the same geometry as in the BF backward facing step case. Since the aim of this analysis is to compare the wall functions between IB and BF as good as possible, this ideal case of the exact same geometry is used in the following test cases. However, it is usually very unlikely that the IBS method can model the exact geometry, and to account for these cutting inaccuracies,

Table 4.2: Numerical schemes used for all wall function test cases

`ddtSchemes`	
default	steadyState
`gradSchemes`	
default	cellLimited leastSquares 1
`divSchemes`	
default	none
div(phi, U)	Gauss vanLeerDC
div(phi, k)	Gauss upwind
div(phi, Epsilon)	Gauss upwind
div(phi, Omega)	Gauss upwind
div(nuEff*dev(T(grad(U))))	Gauss linear
`laplacianSchemes`	
default	Gauss linear limited 0.5
`interpolationSchemes`	
default	linear
`snGradSchemes`	
default	limited 0.5

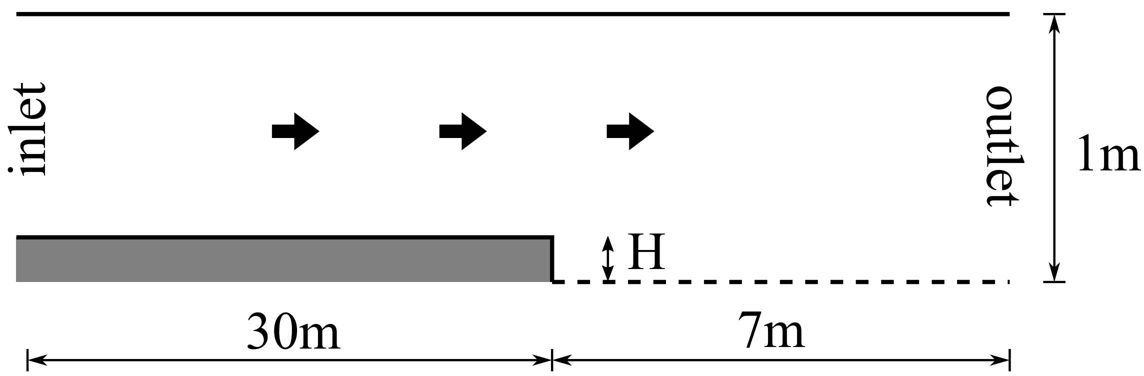
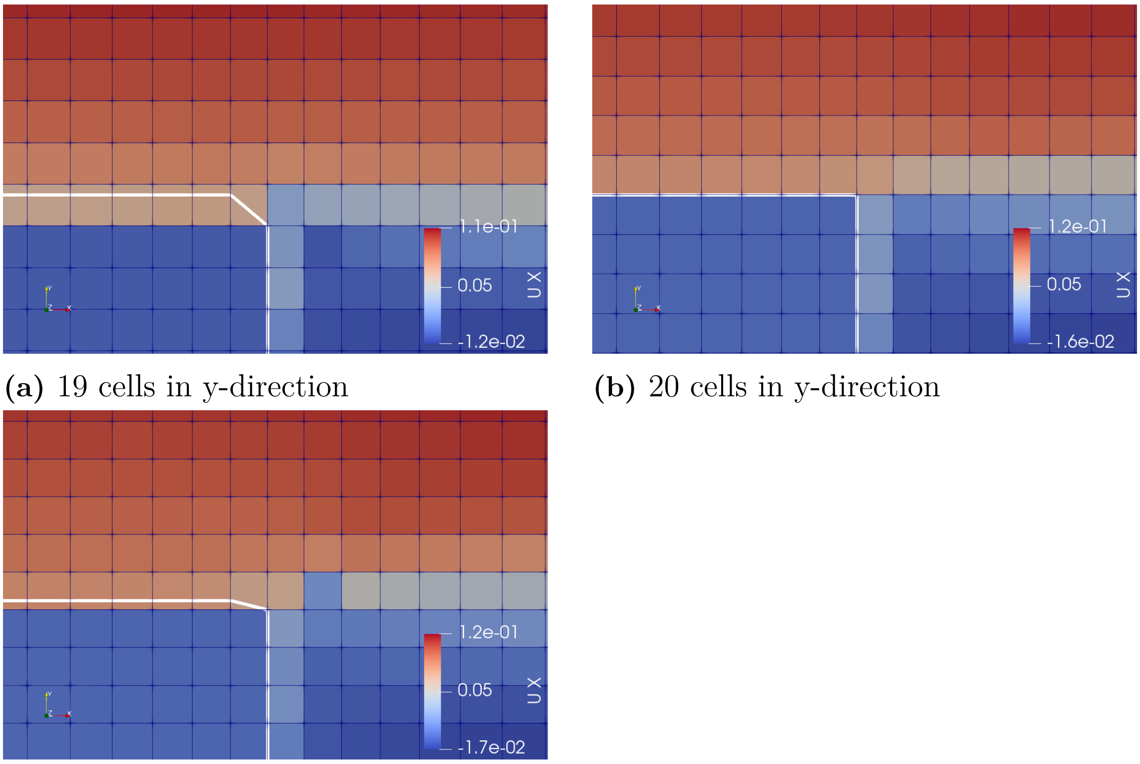


Figure 4.17: Test case 1: backward facing step with $H = 0.25m$. —— : wall; - - - : symmetry plane

Table 4.3: Velocity and pressure boundary conditions for BF and IB case

patch	U	p
inlet	fixedValue, uniform 0.1	zeroGradient
outlet	inletOutlet	fixedValue, uniform 0
top	fixedValue, uniform 0	zeroGradient
bottom	symmetryPlane	symmetryPlane
front and back	empty	empty
obstacle	IB: mixedIb (Dirichlet u=0) BF: wall (no-slip)	IB: mixedIb (zeroGradient) BF: zeroGradient



(a) 19 cells in y-direction

(b) 20 cells in y-direction

(c) 21 cells in y-direction

Figure 4.18: IB cutting for three different meshes in the backward facing step case

additional cases with a slightly changed mesh are computed as well. Three different IB cases are shown in figure 4.18. In all three cases the STL has a height of 0.25 meters, but the number of cells in y-direction is different. In figure 4.18b the STL lies on the background grid and therefore results in an exact replication of the backward facing step. In figure 4.18a and 4.18c, on the other hand, the STL lies inside a row of cells, which is why the cells have to be cut and the corner cell is not rectangular. A rectangular STL object will always have small cutting inaccuracies at the corners as long as the STL is not exactly on the background grid. The reason for this is that a face is only cut when the STL intersects the face. This also prevents cases where a face could get two neighbours, if the corner of the STL object lies on a face of the background mesh.

The difference between the cases in figure 4.18 is not only that the geometry is different, but also that the cells above the IB have different heights. While in figure 4.18b and 4.18c the cells above the IB have the same or almost the same heights as all the others cells, the $y+$ values in the case with 19 cells in y-direction in figure 4.18a are much smaller.

Figure 4.19 shows the residual plot for the IB case with the k-Omega-SST model as an exemplary for all cases. A steady state solution with small fluctuations is reached after about 300 steps. The same behaviour can also be seen for the BF mesh cases, even though the fluctuations are significantly smaller. For this reason, the following results are taken after 600 steps.

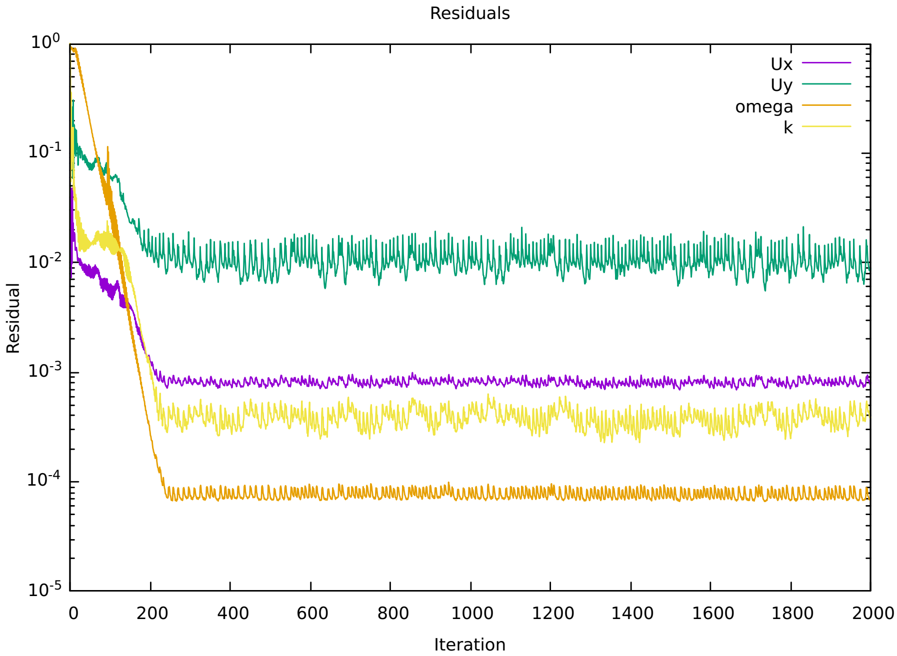


Figure 4.19: Residual plot for IB case with k-Omega-SST model in backward facing step case

The velocity profiles, normalized with a reference velocity at the centre of the fully developed channel flow and multiplied by a factor 1.25, are shown for six different cases at five different locations in figure 4.20. The first position is located on the block while the other four are behind the backward facing step. The six different cases are two BF cases using the k-Epsilon and k-Omega-SST models (BF_kEpsilon and BF_kOmega) and four IB cases, also using the k-Epsilon and k-Omega-SST models. For the three IB cases with the k-Omega-SST model, three different meshes were used as previously described, see figure 4.18.

In general, it can be said that the newly implemented IB wall functions are quite similar to the BF wall function in the backward facing step case. In front of the step, the velocities in the channel are almost exactly the same and also behind the backward facing step only small differences in the range 0.8H - 1.5H can be seen. For the IB_kOmega_19 case the differences almost disappear, which can be explained by the cutting inaccuracies and the different corner. It is probably a coincidence that the cutting inaccuracies correct the velocity profile in the right direction.

Looking closely at the first location (5H) in figure 4.20, it appears that the velocity in the x-direction at the IB is not zero, as it should be. This, however, is not an error in the IBS method in fe41NR, but an incorrect interpolation of the velocity field. Inside the domain, the velocity field is written out and stored in the cell centres and manipulated at the boundaries. Since the results do not store the inserted IB faces and the values set at the IB, the results do not show that the velocity at the IB is zero. Instead, the velocity at the IB is an interpolated value between the first dead cell inside the IB and the first cell inside the fluid domain. The velocity at the IB is stored in an additional VTK file at each output time step, which in

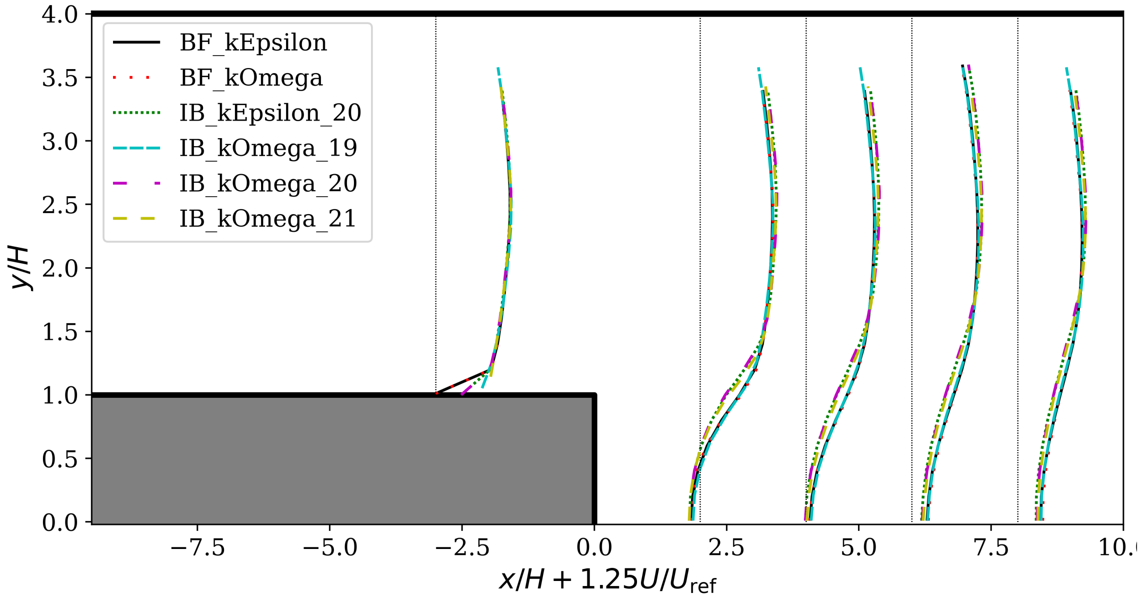


Figure 4.20: Relative velocity profiles at five different locations for 6 different cases

this case only provides zero values. The velocity profiles in figure 4.20 were created with the post-processing utility `sample`, which interpolates the velocity to the faces. For the cases IB_kOmega_19 and IB_kOmega_21 this means that lines of the velocity profile have different lengths because the faces of the background mesh lie at different heights.

For the mesh with 20 cells in y-direction, where the IB lies on the background grid, the velocity values at the IB could be set to zero to correct the incorrect values. With the other two meshes, however, it is more difficult. For the cells that are cut by the IB, the centre point is moved during the cutting and the centre point with the velocity vector lies closer to the neighbouring face, see figure 3.3 in section 3.1.2. During post-processing, the cell-centres are not moved, resulting in a different location of the velocity vector of the cut-cells. Hence, post-processing also results in incorrectly interpolated velocity values at the first face inside the stream.

For the contour plots of the velocity field, as figure 4.18a, a similar behaviour is observed. For the cut-cells with their manipulated cell centre, the velocity value is applied to the entire background cell, giving the impression that the velocity inside the IB is not zero. In summary, post-processing utilities and Paraview cannot visualize cut-cells and the fact that the velocity is interpolated incorrectly at the IB should be taken into account when analyzing the velocity.

For the analysis of the wall function, wall shear stress and wall friction are important quantities. Therefore, a comparison of the friction coefficient between the the BF and IB methods is shown in figure 4.21. Again, the black solid and the red dotted line are the BF grids, while the other four lines are IB cases. Despite the fact that all IB cases are quite close to the BF methods, the IB case with 21 cells in the y-direction shows a bigger difference than the other IB cases. Since the IB case with 19 cells in the y-direction is again closer to the BF cases, there seems to be a correlation between the y+ values and the quality of the wall functions. The y+ values for the

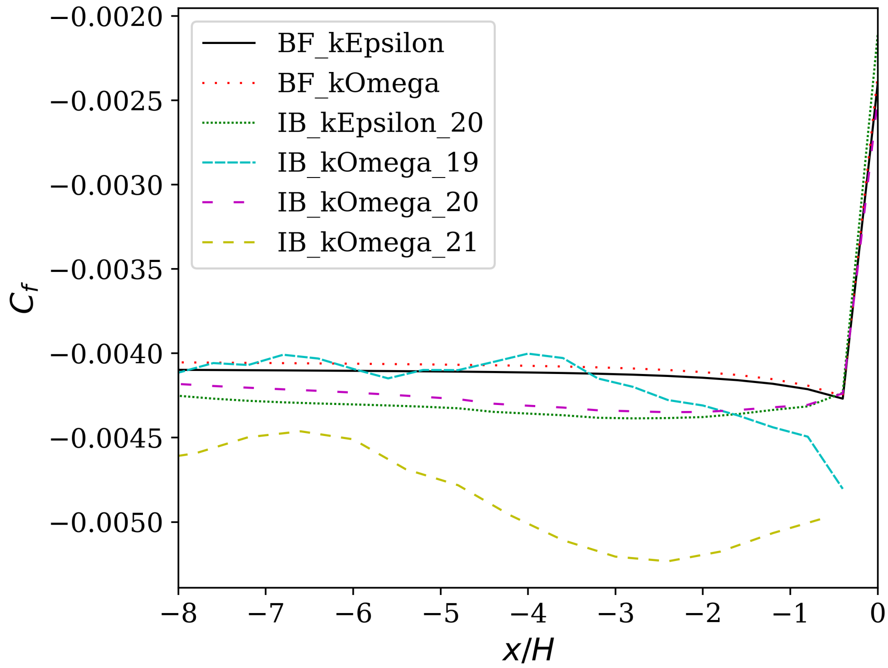


Figure 4.21: Friction coefficient on the last meter before the backward facing step

Table 4.4: y+ values for k-Omega-SST cases in backward facing step case at $x/H = -2$

case	y+ value at $x/H = -2$
BF_kOmega	112.8
IB_kOmega_19	31.4
IB_kOmega_20	120.6
IB_kOmega_21	94.5

different cases are shown in the table 4.4. The largest differences occur at the corner of the rectangle, where the cases with a sharp corner have higher values than in the channel. Since the two IB cases with cutting inaccuracies don't have such a sharp corner, the increase in c_f is not visible. For the backward facing step case, it can be said that the IB wall functions work as they should. The differences between the BF wall functions and the IB wall functions are relatively small and do not depend much on the choice of the background mesh and type of cutting. Despite the differences in the friction coefficient due to cutting inaccuracies, the velocity profiles are very similar to the BF mesh cases.

4.3.2 Test Case 2: Forward Facing Step

The second test case is the forward facing step case. Again, flow separation occurs behind a corner, but this time the flow is disturbed by a narrowing of the channel instead of a widening. As a result, the recirculation region is above the IB and challenges the wall functions in a different way than in the backward facing step. The solver settings and boundary conditions are no different from the backward

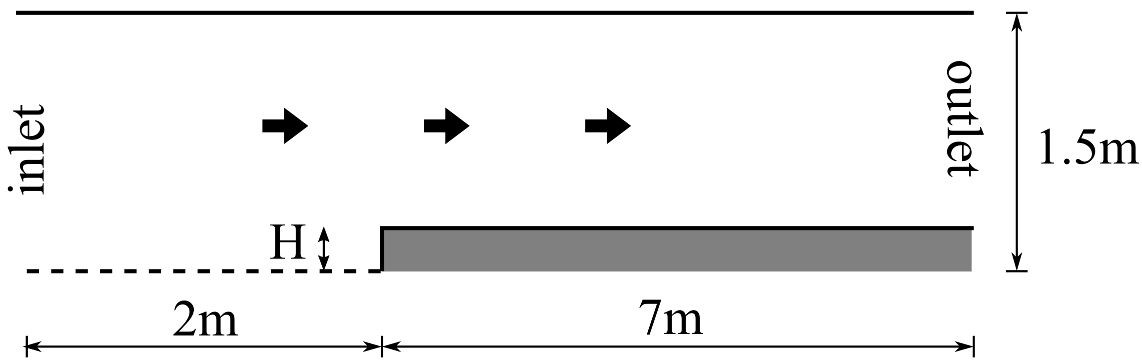


Figure 4.22: Test case 2: forward facing step with $H = 0.25m$. — : wall; - - - : symmetry plane

Table 4.5: Highest y+ values for k-Epsilon cases in forward facing step case

case	max. y+ value
BF_coarse_kEpsilon	86.4
BF_fine_kEpsilon	26.8
IB_coarse_kEpsilon	194.9
IB_fine_kEpsilon	76.3

facing step, only the domain is changed to a forward facing step, as can be seen in figure 4.22. Since a `symmetryPlane` boundary condition is used at the bottom, a long inlet is not required for a fully developed channel flow. The flow inside the channel has a constant mass inflow and a kinematic viscosity $\nu = 10^{-6}$. The residual plot in figure 4.23 shows again that a running time of 600 steps is sufficient to obtain a converged solution. Small fluctuations as in the backward facing step case can be seen as well. After testing different IB cuttings due to different number of cells in the y-direction in the backward facing step, the IB surface of the STL always lies on the background grid in this test case, which results in exactly the same geometry for the IB cases as for the BF cases. The shape of the corner of the forward facing step has too much influence on the analysis to test different IB cuttings. Instead, a coarse and a fine background mesh are analyzed. Table 4.5 shows the highest y+ values for the coarse and fine mesh. The large differences between the BF cases and the IB cases can be explained by the differences in the velocity field, which are discussed later. In the coarse mesh cases, the same cell height was used as in the backward facing step case. In the fine mesh, on the other hand, the number of cells in the y-direction is three times as high, and much smaller y+ values are achieved.

The figure 4.24 shows four different velocity profiles at six different x-positions. The velocities are normalized and multiplied by a factor as in the backward facing step case.

In the BF mesh cases, a clear recirculation zone with negative velocities can be seen immediately after the forward facing step. After about 1.25 meters ($\equiv 5H$), all x-velocities become greater than zero for the k-Epsilon model and after around 1.75m ($\equiv 7H$) for the k-Omega-SST model. This behaviour cannot be observed for the IB cases.

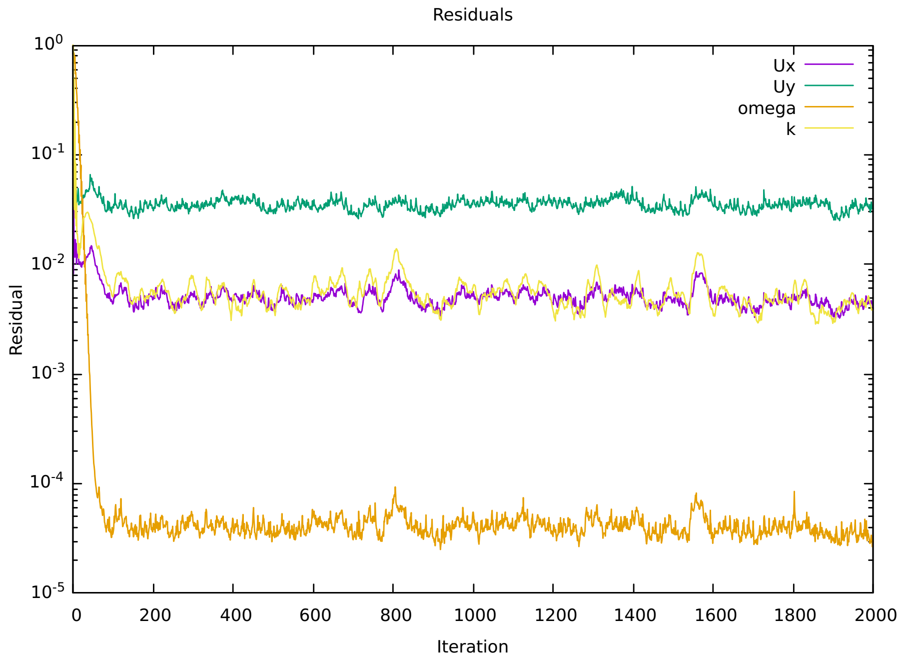


Figure 4.23: Residual plot for IB case with k-Omega-SST model in forward facing step case

For both the k-Epsilon model and the k-Omega-SST model, the results of the IB cases show reduced velocities in the separation region and a transition to the channel flow with a boundary layer, but no recirculation. Therefore, the differences between the BF and the IB wall functions are high in the recirculation region, but become smaller once the flow transitions back to channel flow. Furthermore, for the IB cases, the differences between the k-Omega-SST and the k-Epsilon model are negligible.

While the recirculation in the coarse mesh could not be modeled with IB wall functions, negative x-velocities are clearly seen for the fine mesh cases in figure 4.25. In contrast to the coarse mesh, the k-Epsilon IB wall functions show similar results to the BF cases up to $5H$ after the forward facing step when using a fine mesh. Slightly higher differences are observed for the k-Omega wall functions. However, after the recirculation region, the cases with a fine mesh show an excessive increase in velocity in the near-wall area. This suggests that the implemented wall functions better model recirculation behind the forward facing step in fine meshes, but become worse in normal channel flow with boundary layers. It should be noted that the y+ values for the BF case with a fine mesh even reach values below 30, where good results with near-wall modelling cannot be assumed.

As with the velocity profiles, large differences can be observed in the friction coefficient for coarse meshes. While in the BF methods with coarse meshes a large part of the first 2 meters consists of positive friction coefficient values, which means negative x-velocities, the friction coefficient in the IB cases is predominantly negative. This confirms the results obtained from the velocity profiles. The further away from the forward facing step, the more the IB wall functions approach the BF wall functions.

For the fine mesh IB cases, the friction coefficient also confirms the results from the

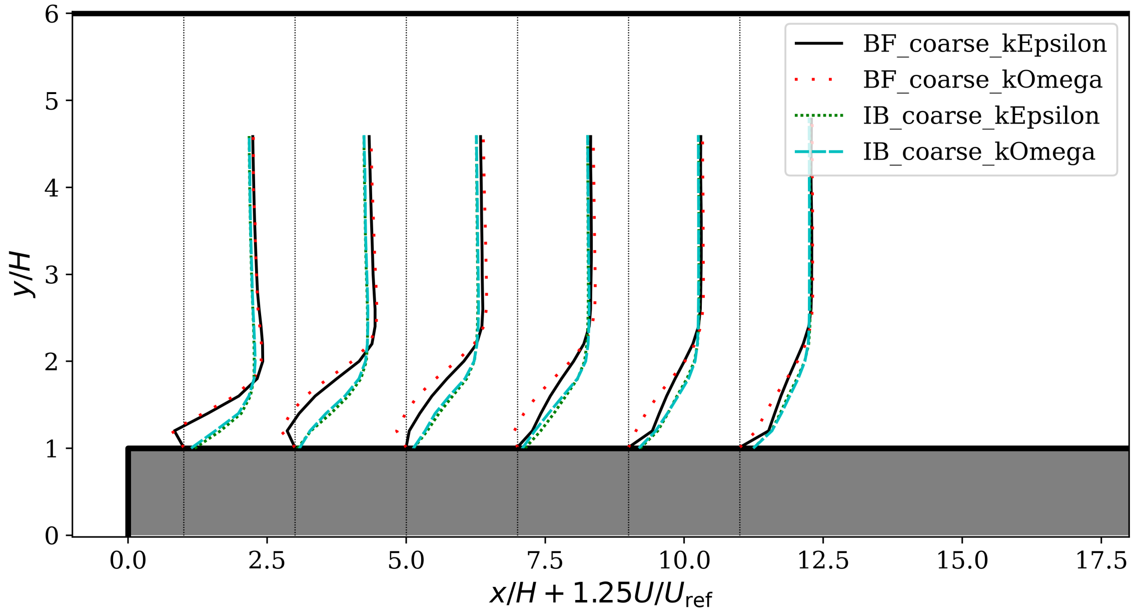


Figure 4.24: Relative velocity profiles at five different locations for 4 different cases with a coarse mesh

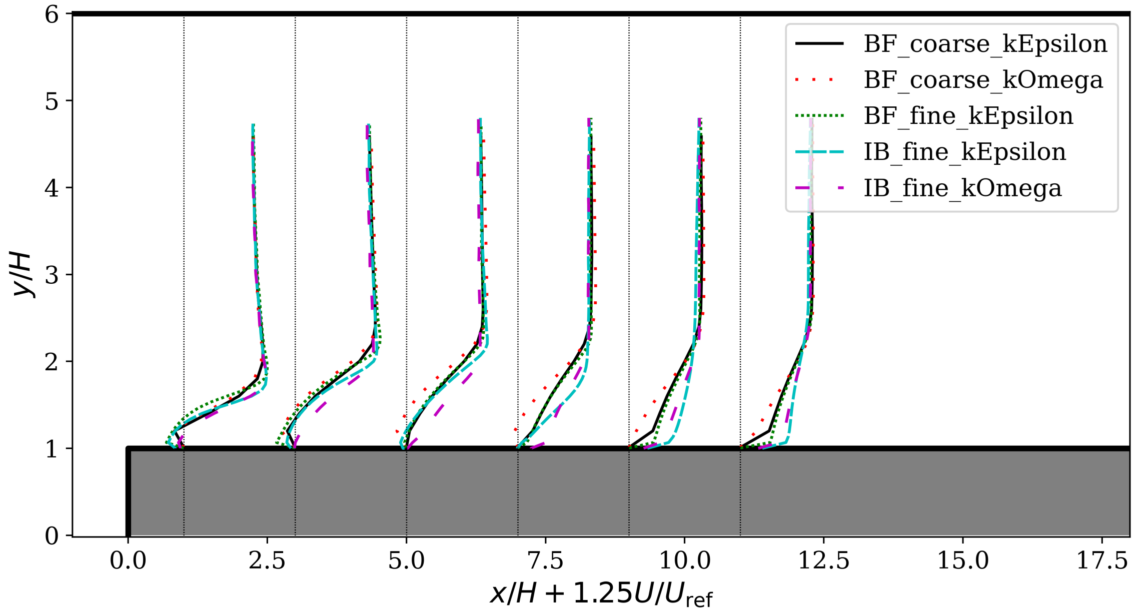


Figure 4.25: Relative velocity profiles at five different locations for 2 IB cases with a fine mesh and 3 BF cases

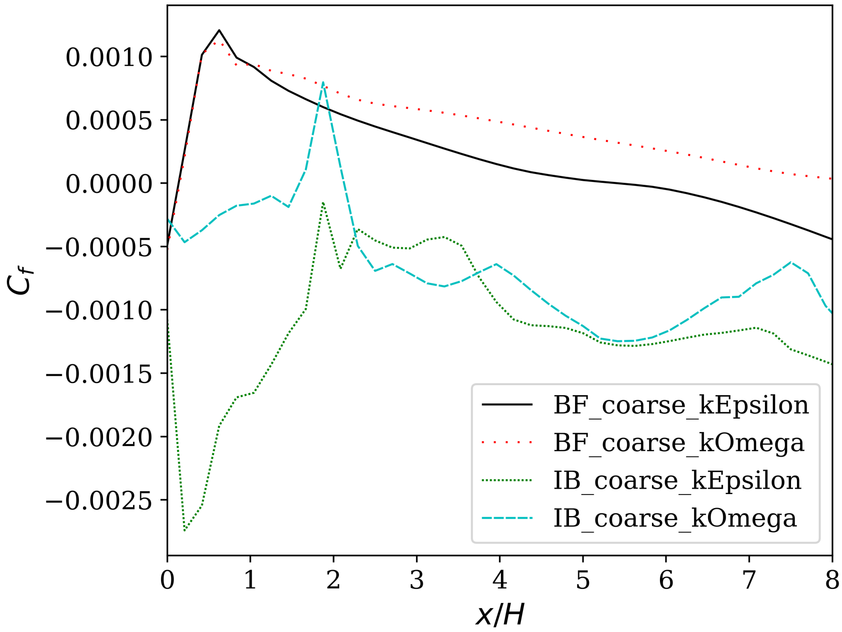


Figure 4.26: Friction coefficient on the obstacle behind the forward facing step for the coarse mesh cases

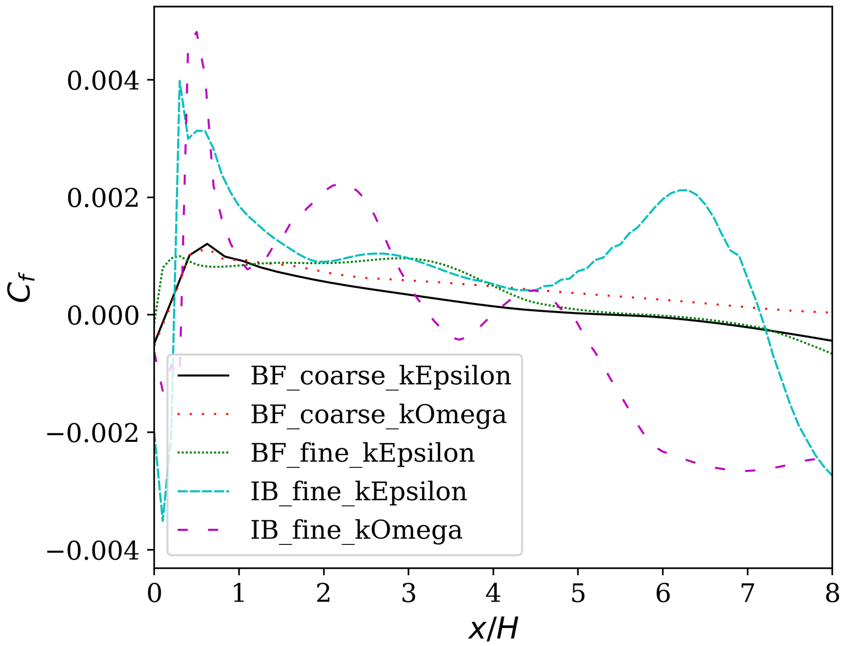


Figure 4.27: Friction coefficient on the obstacle behind the forward facing step for fine and coarse mesh cases

velocity profiles. The recirculation is modelled by the IB wall function, albeit somewhat exaggerated. Besides the fact that the friction coefficients between $x = 2H$ and $x = 4H$ are all quite close, it can be seen that the cases with a fine background mesh do not converge after 600 steps. The friction coefficient and the velocity field

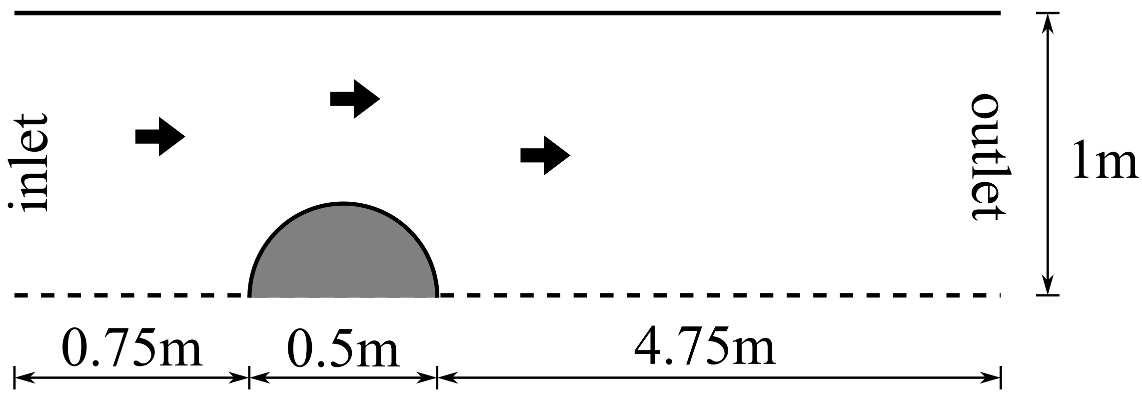


Figure 4.28: Test case 3: cylinder in channel. —— : wall; - - - : symmetry plane

vary behind the forward facing step, especially when using the k-Omega-SST model.

Nevertheless, it can be summarized that the IB wall functions adequately model near-wall regions in undisturbed channel flow and in the backward facing step. Behind a forward facing step, on the other hand, a fine mesh is required to achieve results similar to the BF wall functions. However, this also leads to poor results inside the channel, further away from the forward facing step. As this analysis was performed on a perfect replica of the IB object with rectangular corners, cutting inaccuracies were not taken into account. The use of fine meshes also has the advantage of reducing the inaccuracies which would greatly affect the results in cases such as the forward facing step.

4.3.3 Test Case 3: Cylinder in Channel Flow

In the third test case, a cylinder is placed inside a channel. This not only changes the IB cutting completely, but also shifts the separation point from a sharp corner to a smooth, curved face. The main solver settings are kept as in the previous two cases, only the domain and boundary conditions are slightly changed.

As in the second test case, the upper boundary is a wall, while the lower boundary is a symmetry plane. For the separation analysis of the flow around the cylinder, a fully developed channel flow is not required and the channel can be comparatively short. The cylinder has a diameter of 0.5 meters and is firmly anchored in the domain. The inlet velocity varies between some of the cases to manipulate the Reynolds number and hence the y+ values for different meshes. The kinematic viscosity is $\nu = 10^{-6}$ for the turbulent cases and $\nu = 10^{-1}$ for the laminar cases, respectively.

The test case was calculated a total of six times with the SST $k-\omega$ model and two times with laminar flow. For the turbulent cases, different mesh sizes were used to analyze the effects on the velocity field.

Figure 4.29 shows the velocity contour plots of 5 turbulent cases. The first two, 4.29a and 4.29b, have the inlet velocity $U_{inlet} = 0.1m/s$, while all others have $U_{inlet} = 1.0m/s$. The y+ values of each case can be seen in table 4.6 with an additional case, IB fine ($U_{inlet} = 0.1m/s$), presented later.

In figure 4.29 it is clear that the IB wall functions obviously do not work as well as for

Table 4.6: y+ values for cylinder in channel test cases

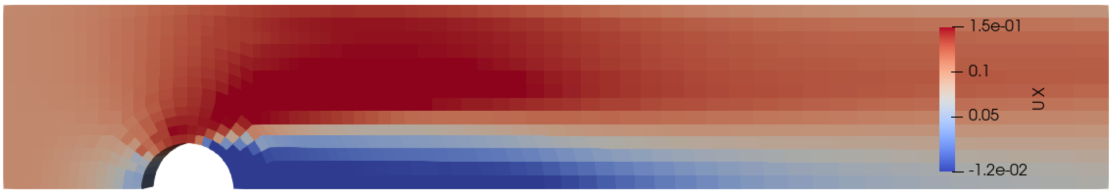
cases	min.	max.	average
IB coarse ($U_{inlet} = 0.1$)	1.3	188.9	41.6
IB medium ($U_{inlet} = 1.0$)	1.0	646.8	126.0
IB fine ($U_{inlet} = 1.0$)	0.5	416.1	72.5
IB fine ($U_{inlet} = 3.0$)	1.2	1059.4	218.1
BF coarse ($U_{inlet} = 0.1$)	31.4	222.5	128.6
BF fine ($U_{inlet} = 1.0$)	24.4	486.9	219.1

the backward or even the forward facing step case. The separation and recirculation region is much smaller for the IB cases than for the cases with body-fitted meshes. When using a coarse mesh, the difference is not tremendous, but when the cell heights are reduced, the differences become much larger. While the recirculation region in the BF cases becomes larger when the number of cells is increased, the recirculation region in the IB cases becomes smaller. The flow around the cylinder is attached longer and the influence of the cylinder on the stream is smaller. This behaviour does not correlate with the average y+ value as can be seen in table 4.6. Due to the higher inlet velocity in the IB medium and IB fine case, the y+ values do not deviate as much from the IB coarse case. However, it can be seen that the variance between the highest and lowest y+ values is much larger for the IB cases than for the BF cases. This is due to the IB cutting, which is different for each cell when using a curved STL surface. Unlike the first two test cases, which used a rectangle, the cell height varies greatly when cutting a cylinder into a uniform Cartesian background mesh. This can be seen in figure 4.30. The fact that the y+ values can vary greatly on the same boundary for the IBS method should be taken into account when using IB wall functions. It also makes it more difficult to implement resilient and stable IB wall functions.

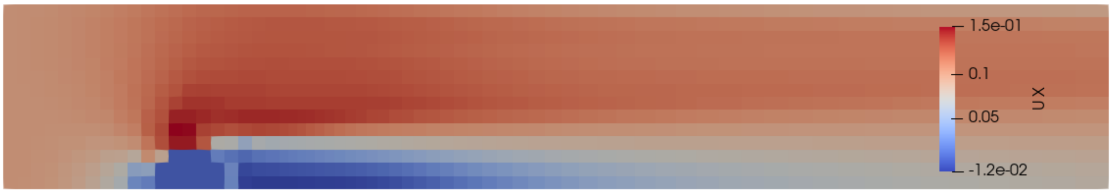
The real reason why the results get worse with finer meshes is probably the shape of the cylinder. Since the cells are cut linearly, the cylinder surface in IBS is not as smooth as when using body-fitted meshes. In figure 4.30a, it can be seen that the surface is much more angular with results in a stronger separation. For a nearly smooth circular surface, as in figure 4.30c, the flow attaches much longer than in the other cases. The implemented IB wall functions have problems modelling detached flow due to strongly curved surfaces.

That the size of the recirculation area does not depend on the y+ values can be seen again in figure 4.31. In both cases the same mesh was used, but the inlet velocity changed to obtain higher y+ values for the case in figure 4.31b. The y+ values for this case can also be found in the table 4.6 above.

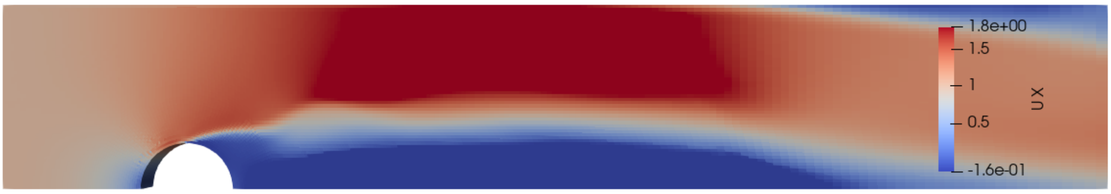
As a final comparison, test case 3 was calculated in laminar flow, as the kinematic viscosity was changed from $\nu = 10^{-6}$ to $\nu = 10^{-1}$. A fine mesh with 56 cells in y-direction was used in both the BF and IB case. This led to almost identical results for the BF and IB cases. Visually, no differences between figure 4.32a and figure 4.32b can be seen.



(a) BF with 14 cells in y-direction and $U_{inlet} = 0.1$



(b) IB with 14 cells in y-direction and $U_{inlet} = 0.1$

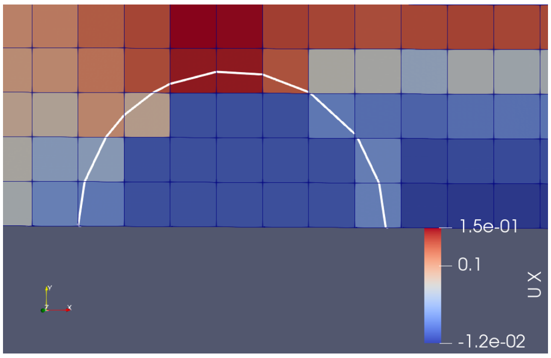


(c) BF with 56 cells in y-direction and $U_{inlet} = 1.0$

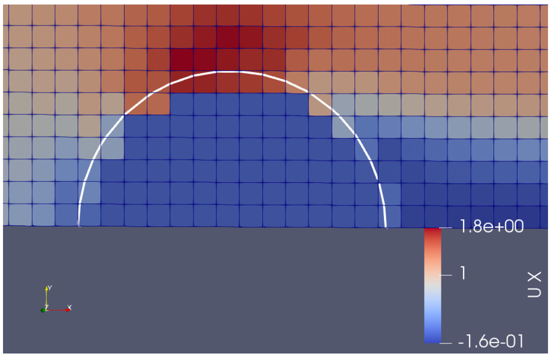
(d) IB with 28 cells in y-direction and $U_{inlet} = 1.0$

(e) IB with 56 cells in y-direction and $U_{inlet} = 1.0$

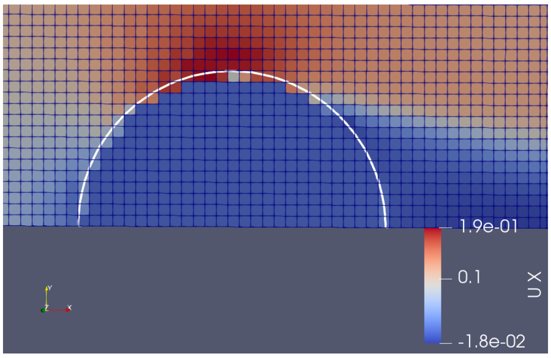
Figure 4.29: Test case 3: x-velocity contour plots after 3000 steps for 3 different IB cases and 2 BF cases



(a) Coarse mesh with 14 cells in y-direction



(b) Medium mesh with 28 cells in y-direction



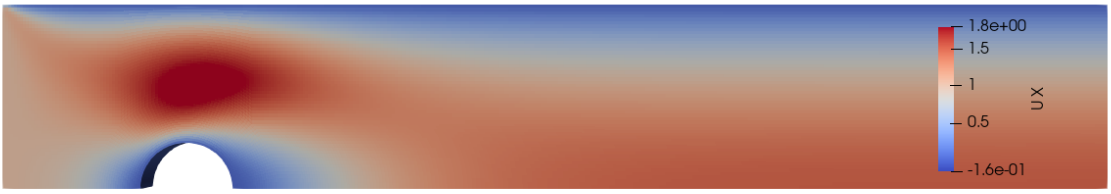
(c) Fine mesh with 56 cells in y-direction

Figure 4.30: Test case 3: IB cutting for the three different meshes

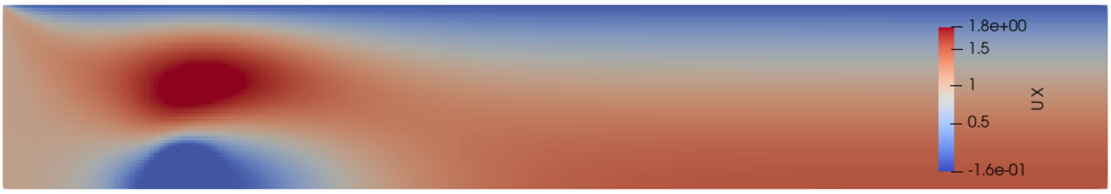
(a) IB with 56 cells in y-direction and $U_{inlet} = 1.0$

(b) IB with 56 cells in y-direction and $U_{inlet} = 3.0$

Figure 4.31: Test case 3: x-velocity contour plots after 3000 steps for two IB cases with fine mesh but different inlet velocities



(a) Laminar BF with 56 cells in y-direction and $U_{inlet} = 1.0$



(b) Laminar IB with 56 cells in y-direction and $U_{inlet} = 1.0$

Figure 4.32: Test case 3: x-velocity contour plots after 3000 steps for a laminar IB case and a laminar BF case

5 Conclusion

This master thesis gives an overview of the Immersed Boundary Surface (IBS) method in `foam-extend`. In this overview, the implementation of the cutting process as well as the cutting limitations and boundary conditions are addressed. Furthermore, an analysis on the functionality and reliability of the IBS method is given with the topics mesh refinement, mass fluxes and wall functions. For the analysis, simple test cases have been created to highlight individual components.

Between the `foam-extend` version 4.0 and 4.1, a lot has changed in the implementation of the IB method. The newly implemented IB method in `foam-extend` version 4.1, called IBS method, no longer uses polynomials but cuts the background mesh and manipulates the cell data. For all intersected cells, the volume and face data are recalculated only for the part inside the fluid, which further changes the discretization matrix. Although the change from a ghost-cell approach to a cut-cell approach has enabled sharp interfaces and improved several aspects, the IBS is still a work in progress with many of improvements to come over time.

When merging complex objects or multiple IBs into the background mesh, or even when choosing the coarseness of the background mesh, the IBS method is limited by the fact that each cell can only be cut once. Therefore, the cells of the background mesh should not be thicker than the immersed object, and also the correct mapping of sharp corners cannot be guaranteed. In the case of multiple IBs, an overlap of at least one cell is required for a contact between two IB patches, as two IB boundaries per cell are not possible.

When it comes to fluid mechanics, the conservation of mass is probably one of the most important principles. But when using the IBS method with moving IBs, this simple conservation law is not so easy to fulfil. As described in section 3.1.2, the IBS method uses the idea of manipulating the old cell volume to account for mesh fluxes. In most cases, the movement of the IB changes not only the volume of the newly intersected cells but also the volume of old intersected cells and other cells, resulting in volume changes which would be neglected. However, this method, which is consistent in theory, does not yet lead to mass conservation. As shown by several test cases in section 4.2, the IBS method is not mass conserving for moving IBs. Although the mass fluxes on the IB surface are zero, the fluid volume appears to vanish and emerge. The translation of mesh fluxes to mass fluxes seems to work better in high-pressure regions than in low-pressure regions, but still both are wrong. Furthermore, the implemented version in the `foam-extend 4.1 nextRelease` branch is also not mass conserving for stationary IBs. However, this seems to be a result of earlier bug fixes, as it already worked in previous versions as in the master branch.

To validate and analyze the implemented IB wall functions, three different test cases are used to compare the results with established body-fitted wall functions. While the IB wall functions show very good results for normal channel flow and in the backward facing step case, differences can be found in the forward facing step case with strong separation by a sharp corner. For strongly curved surfaces such as a cylinder in a channel, the IB wall function do no provide reliable results. In addition to the fact that the implemented IB wall functions are not yet as evolved as BF wall function, cutting limitations and a large range of y+ values complicate the modelling of turbulence in near wall regions. Due to cell cutting, the height of the IB cells varies greatly and thus also the y+ values.

All in all, it can be said that the IBS method in foam-extend can be a good alternative to BF meshing methods. Due to major problems with mass conservation, it is not yet possible in the `nextRelease` branch to produce reliable results for either stationary IBs or moving IBs. However, the IBS method is a work in progress that will become more applicable with further updates and has the potential to be a good alternative to BF meshing methods. At this stage, it is very important to understand the limitations of the IBS method, to know its purpose and that the method is constantly evolving.

Bibliography

[1] D. K. Clarke, M. D. Salas, and H. A. Hassan. "Euler calculations for multielement airfoils using Cartesian grids". In: *AIAA Journal* 24.3 (1986), pp. 353–358. DOI: 10.2514/3.9273. eprint: https://doi.org/10.2514/3.9273. URL: https://doi.org/10.2514/3.9273.

[2] J. H. Ferziger, M. Perić, and R. L. Street. *Computational Methods for Fluid Dynamics*. Switzerland: Springer, Cham, 2020. DOI: https://doi-org.proxy.lib.chalmers.se/10.1007/978-3-319-99693-6.

[3] D. Goldstein, R. Handler, and L. Sirovich. "Modeling a No-Slip Flow Boundary with an External Force Field". In: *Journal of Computational Physics* 105.2 (1993), pp. 354–366. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1993.1081. URL: https://www.sciencedirect.com/science/article/pii/S0021999183710818.

[4] R. Issa. "Solution of the implicitly discretised fluid flow equations by operator-splitting". In: *Journal of Computational Physics* 62.1 (1986), pp. 40–65. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(86)90099-9. URL: https://www.sciencedirect.com/science/article/pii/0021999186900999.

[5] H. Jasak et al. *foam-extend vol. 4.1*. Available at https://sourceforge.net/projects/foam-extend/files/foam-extend-4.1/. 2019.

[6] H. Jasak. "Error analysis and estimation for the finite volume method with applications to fluid flows". PhD thesis. Imperial College of Science, Technology and Medicine, Department of Mechanical Engineering, 1996.

[7] H. Jasak. "Immersed boundary surface method in foam-extend". In: *The 13th OpenFOAM Workshop (OFW13)* (June 2018), pp. 55–59.

[8] H. Jasak, D. Rigler, and Ž. Tuković. "Design and Implementation of Immersed Boundary Method with Discrete Forcing Approach for Boundary Conditions". In: *Proc. of the 6th European Conference on Computational Fluid Dynamics*. ECFD VI. 2014, pp. 5319–5332.

[9] J. Kettemann and C. Bonten. "Application of the immersed boundary surface method in OpenFOAM". In: *AIP Conference Proceedings* 2289.1 (2020), p. 020032. DOI: 10.1063/5.0028625. URL: https://aip.scitation.org/doi/abs/10.1063/5.0028625.

[10] J. Kettemann, I. Gatin, and C. Bonten. "Verification and validation of a finite volume immersed boundary method for the simulation of static and moving geometries". In: *Journal of Non-Newtonian Fluid Mechanics* 290 (2021). Available at https://doi.org/10.1016/j.jnnfm.2021.104510, p. 104510. ISSN:

0377-0257. URL: https://www.sciencedirect.com/science/article/pii/S0377025721000343.

[11] B. Launder and D. Spalding. "The numerical computation of turbulent flows". In: *Computer Methods in Applied Mechanics and Engineering* 3.2 (1974), pp. 269–289. ISSN: 0045-7825. DOI: https://doi.org/10.1016/0045-7825(74)90029-2. URL: https://www.sciencedirect.com/science/article/pii/0045782574900292.

[12] W. Malalasekera and H. Versteeg. *An introduction to computational fluid dynamics: the finite volume method*. Pearson Prentice Hall, 2007.

[13] F. Menter, M. Kuntz, and R. Langtry. "Ten years of industrial experience with the SST turbulence model". In: *Heat and Mass Transfer* 4 (Jan. 2003).

[14] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics*. Switzerland: Springer, Cham, 2016. DOI: https://doi.org/10.1007/978-3-319-16874-6.

[15] J. Nikuradse. "Gesetzmassigkeiten der Turbulenten Stromung in Glatten Rohren". In: *VDI-Forschungsheft* 356 (1932), pp. 1–36.

[16] S. Patankar and D. Spalding. "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows". In: *International Journal of Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806. ISSN: 0017-9310. DOI: https://doi.org/10.1016/0017-9310(72)90054-3. URL: https://www.sciencedirect.com/science/article/pii/0017931072900543.

[17] C. S. Peskin. "Flow patterns around heart valves: A numerical method". In: *Journal of Computational Physics* 10.2 (1972), pp. 252–271. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(72)90065-4. URL: https://www.sciencedirect.com/science/article/pii/0021999172900654.

[18] E. Saiki and S. Biringen. "Numerical Simulation of a Cylinder in Uniform Flow: Application of a Virtual Boundary Method". In: *Journal of Computational Physics* 123.2 (1996), pp. 450–465. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1996.0036. URL: https://www.sciencedirect.com/science/article/pii/S0021999196900364.

[19] U. Şentürk et al. "Benchmark simulations of flow past rigid bodies using an open-source, sharp interface immersed boundary method". In: *Progress in Computational Fluid Dynamics An International Journal* 1 (Mar. 2018). DOI: 10.1504/PCFD.2017.10009753.

[20] Y.-H. Tseng and J. H. Ferziger. "A ghost-cell immersed boundary method for flow in complex geometry". In: *Journal of Computational Physics* 192.2 (2003), pp. 593–623. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2003.07.024. URL: https://www.sciencedirect.com/science/article/pii/S0021999103004108.

[21] T. Ye et al. "An Accurate Cartesian Grid Method for Viscous Incompressible Flows with Complex Immersed Boundaries". In: *Journal of Computational Physics* 156.2 (1999), pp. 209–240. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1999.6356. URL: https://www.sciencedirect.com/science/article/pii/S0021999199963568.

A

immersedBoundaryPolyPatch.C

Listing A.1: immersedBoundaryPolyPatch.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

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

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

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

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

#include "immersedBoundaryPolyPatch.H"
#include "foamTime.H"
#include "polyBoundaryMesh.H"
#include "polyMesh.H"
#include "emptyPolyPatch.H"
#include "ImmersedFace.H"
#include "ImmersedCell.H"
#include "triSurfaceDistance.H"
#include "mergePoints.H"
#include "processorPolyPatch.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(immersedBoundaryPolyPatch, 0);

    addToRunTimeSelectionTable(polyPatch, immersedBoundaryPolyPatch, word);
    addToRunTimeSelectionTable
    (
        polyPatch,
        immersedBoundaryPolyPatch,
        dictionary
    );
}


```

```
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

const Foam::debug::tolerancesSwitch
Foam::immersedBoundaryPolyPatch::spanFactor_
(
    "immersedBoundarySpanFactor",
    20
);


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

Foam::vector Foam::immersedBoundaryPolyPatch::cellSpan
(
    const label cellID
) const
{
    const polyMesh& mesh = boundaryMesh().mesh();

    // Calculate span from the bounding box size (prefactor is arbitrary, IG
    // 10/Nov/2018)
    const scalar delta = spanFactor_()*cmptMax
    (
        boundBox
        (
            mesh.cells()[cellID].points
            (
                mesh.faces(),
                mesh.points()
            ),
            false // Do not reduce
        ).span()
    );

    return vector(delta, delta, delta);
}


void Foam::immersedBoundaryPolyPatch::calcTriSurfSearch() const
{
    if (debug)
    {
        InfoInFunction
            << "creating triSurface search algorithm"
            << endl;
    }

    // It is an error to attempt to recalculate
    // if the pointer is already
    if (triSurfSearchPtr_)
    {
        FatalErrorInFunction
            << "triSurface search algorithm already exist"
            << abort(FatalError);
    }

    triSurfSearchPtr_ = new triSurfaceSearch(ibMesh_);
}


void Foam::immersedBoundaryPolyPatch::calcImmersedBoundary() const
{
    if (debug)
    {
        InfoInFunction
            << "Calling calcImmersedBoundary for patch "
            << name() << " for mesh "
            << boundaryMesh().mesh().time().path()
            << endl;
    }
```

```
    // It is an error to attempt to recalculate
    // if the pointer is already
    if
    (
        ibPatchPtr_
     || ibCellsPtr_
     || ibCellCentresPtr_
     || ibCellVolumesPtr_
     || ibFacesPtr_
     || ibFaceCentresPtr_
     || ibFaceAreasPtr_
     || nearestTriPtr_
     || deadCellsPtr_
     || deadFacesPtr_
    )
    {
        FatalErrorInFunction
            << "Geometry already calculated"
            << abort(FatalError);
    }

    // Get reference to the mesh
    const polyBoundaryMesh& bMesh = boundaryMesh();
    const polyMesh& mesh = bMesh.mesh();

    // Get triSurface search
    const triSurfaceSearch& tss = triSurfSearch();

    // Get mesh points
    const pointField& p = mesh.points();

    // Get mesh faces
    const faceList& f = mesh.faces();

    // Get mesh face centres
    const vectorField& Cf = mesh.faceCentres();

    // Get mesh face areas
    const vectorField& S = mesh.faceAreas();

    // Get mesh cell centres
    const vectorField& C = mesh.cellCentres();

    // Get mesh cell volumes
    const scalarField& V = mesh.cellVolumes();

    // Get face addressing
    const labelList& owner = mesh.faceOwner();
    const labelList& neighbour = mesh.faceNeighbour();

    // Get cell-point addressing
    const labelListList& cellPoints = mesh.cellPoints();

    // Algorithm
    // Initialise the search by marking the inside points using calcInside
    // Based on inside points addressing, check intersected faces and cells
    // For all intersected cells, calculate the actual intersection and
    // - calculate the (cell) intersection face, its centre, and area vector
    // - adjust the cell volume and centre
    // - adjust the face area and face centre

    // Mark points that are inside or outside of the triangular surface
    boolList pointsInside = tss.calcInside(p);

    // Adjust selection of points: inside or outside of immersed boundary
    if (internalFlow())
    {
        Info<< "Internal flow  for patch "
            << name() << " for mesh "
```

```
            << boundaryMesh().mesh().time().path() << endl;
    }
    else
    {
        Info<< "External flow for patch "
            << name() << " for mesh "
            << boundaryMesh().mesh().time().path() << endl;

        // Flip all points inside identifier
        forAll (pointsInside, i)
        {
            pointsInside[i] = !pointsInside[i];
        }
    }

    // Check cell intersections
    labelList intersectedCell(mesh.nCells(), immersedPoly::UNKNOWN);

    // Estimate the number of intersected cells.
    // Used for sizing of dynamic list only
    // HJ, 11/Dec/2017
    label nIntersectedCells = 0;

    // Go through the faces at the interface between a live and dead cell
    // and mark the band of possible intersections
    forAll (intersectedCell, cellI)
    {
        // Get current cell points
        const labelList& curCp = cellPoints[cellI];

        bool foundInside = false;
        bool foundOutside = false;

        forAll (curCp, cpI)
        {
            if (pointsInside[curCp[cpI]])
            {
                // Found a point inside
                foundInside = true;
            }
            else
            {
                // Found a points outside
                foundOutside = true;
            }
        }

        // Check cell classification
        if (foundInside && !foundOutside)
        {
            // All points inside: cell is wet
            intersectedCell[cellI] = immersedPoly::WET;
        }
        else if (!foundInside && foundOutside)
        {
            // All points outside: cell is dry
            intersectedCell[cellI] = immersedPoly::DRY;
        }
        else if (foundInside && foundOutside)
        {
            // Get span
            const vector span = cellSpan(cellI);

            // If the nearest triangle cannot be found within span than this is
            // most probably a tri surface search error. Mark unknown and check
            // later. (IG 22/Nov/2018)
            if (tss.nearest(C[cellI], span/spanFactor_()).index() == -1)
            {
                intersectedCell[cellI] = immersedPoly::UNKNOWN;
            }
```

```
            else
            {
                // Intersected cell
                intersectedCell[cellI] = immersedPoly::CUT;
                nIntersectedCells++;
            }
        }
    }

    // Do a check of the cells selected for cutting but not within the span of
    // the tri surface. The cause of this can either be a stl that is not
    // perfect or ther was an error in the inside/outside tri-search for other
    // reasons. Look at the neigbours that are not CUT and assign their status.
    const cellList& cells = mesh.cells();

    forAll (intersectedCell, cellI)
    {
        if (intersectedCell[cellI] == immersedPoly::UNKNOWN)
        {
            // Check the neigbours
            const cell& curCell = cells[cellI];
            Switch foundWetNei = false;
            Switch foundDryNei = false;

            forAll (curCell, faceI)
            {
                // Only do the check for internal faces. If the face is boundary
                // face then there is nothing to do.
                // NOTE: parallelisation needed?
                if (mesh.isInternalFace(curCell[faceI]))
                {
                    label own = intersectedCell[owner[curCell[faceI]]];
                    label nei = intersectedCell[neighbour[curCell[faceI]]];

                    if
                    (
                        (nei == immersedPoly::DRY)
                     || (own == immersedPoly::DRY)
                    )
                    {
                        foundDryNei = true;
                    }
                    if
                    (
                        (nei == immersedPoly::WET)
                     || (own == immersedPoly::WET)
                    )
                    {
                        foundWetNei = true;
                    }
                }
            }

            if (foundWetNei && !foundDryNei)
            {
                intersectedCell[cellI] = immersedPoly::WET;
            }
            else if (!foundWetNei && foundDryNei)
            {
                intersectedCell[cellI] = immersedPoly::DRY;
            }
            else
            {
                // There are either no wet or dry negbours or there are both.
                // This should not be possible. NOTE: the check is not
                // parallelised and this can theoretically lead to failures in
                // strange arrangements.
                // Issue a warning, mark CUT and hope for the best.
                // (IG 22/Nov/2018)
                if (debug)
```

```
                {
                    WarningInFunction
                        << "Cannot find wet or dry neigbours! Cell C:"
                        << C[cellI]
                        << " Neighbours: WET:" << foundWetNei
                        << ", DRY:" << foundDryNei
                        << endl;
                }

                intersectedCell[cellI] = immersedPoly::CUT;
                nIntersectedCells++;
            }
        }
    }

    // Count all IB cells and faces for debug
    labelList totalIbCount(4);

    // Collect intersection points and faces.  Primitive patch will be created
    // after renumbering

    // IB points
    // Note: it is difficult to estimate the correct size, so use a guessed
    // number of intersected cells and a dynamic list for automatic resizing
    // HJ, 11/Dec/2017
    DynamicList<point> unmergedPoints
    (
        nIntersectedCells*primitiveMesh::pointsPerFace_
    );
    label nIbPoints = 0;

    // IB patch faces: Cell intersections with the IB patch
    faceList unmergedFaces(mesh.nCells());

    // IB cells: cells intersected by the IB patch
    // This also corresponds to faceCells next to the IB patch
    ibCellsPtr_ = new labelList(mesh.nCells());
    labelList& ibCells = *ibCellsPtr_;

    // IB cellCentres: centre of live part of the intersected cell
    // next to the IB patch
    ibCellCentresPtr_ = new vectorField(mesh.nCells());
    vectorField& ibCellCentres = *ibCellCentresPtr_;

    // IB cellCentres: centre of live part of the intersected cell
    // next to the IB patch
    ibCellVolumesPtr_ = new scalarField(mesh.nCells());
    scalarField& ibCellVolumes = *ibCellVolumesPtr_;

    // Nearest triangle
    nearestTriPtr_ = new labelList(mesh.nCells());
    labelList& nearestTri = *nearestTriPtr_;

    // Count interected cells
    label nIbCells = 0;

    // At this point, all live cells are marked with 1
    // Intesect all cells that are marked for intersection

    forAll (intersectedCell, cellI)
    {
        if (intersectedCell[cellI] == immersedPoly::CUT)
        {
            // Found intersected cell

            // Get span
            const vector span = cellSpan(cellI);

            // Create a cutting object with a local tolerance
            triSurfaceDistance dist
```

```
            (
                tss,
                2*span,
                internalFlow(),
                true                // iterate intersection
            );

            // Calculate the intersection
            ImmersedCell<triSurfaceDistance> cutCell
            (
                cellI,
                mesh,
                dist
            );

            // Check for irregular intersections
            if (cutCell.isAllWet())
            {
                intersectedCell[cellI] = immersedPoly::WET;
            }
            else if (cutCell.isAllDry())
            {
                intersectedCell[cellI] = immersedPoly::DRY;
            }
            else
            {
                // True intersection.  Cut the cell and store all
                // derived data

                // Note: volumetric check is not allowed because true
                // intersection guarantees that the faces of the cell
                // have been cut.  Therefore, the cell MUST be an IB cell.
                // If the cut is invalid, Marooney Maneouvre shall correct
                // the error in sum(Sf).  HJ, 12/Mar/2019

                // Store ibFace with local points. Points merge will
                // take place later
                const face& cutFace = cutCell.faces()[0];

                const pointField& cutPoints = cutCell.points();

                // Collect the renumbered face, using the point labels
                // from the unmergedPoints list
                face renumberedFace(cutFace.size());

                // Insert points and renumber the face
                forAll (cutFace, cpI)
                {
                    unmergedPoints.append(cutPoints[cutFace[cpI]]);
                    renumberedFace[cpI] = nIbPoints;
                    nIbPoints++;
                }

                // Record the face
                unmergedFaces[nIbCells] = renumberedFace;

                // Collect cut cell index
                ibCells[nIbCells] = cellI;

                // Record the live centre
                ibCellCentres[nIbCells] = cutCell.wetVolumeCentre();

                // Record the live volume
                ibCellVolumes[nIbCells] = cutCell.wetVolume();

                // Record the nearest triangle to the face centre
                nearestTri[nIbCells] =
                    tss.nearest(cutFace.centre(cutPoints), span).index();

                nIbCells++;
```

```
                }
            }
        }

        // Pick up direct face cuts after regular cell cuts are collected
        forAll (neighbour, faceI)
        {
            if
            (
                intersectedCell[owner[faceI]] == immersedPoly::WET
             && intersectedCell[neighbour[faceI]] == immersedPoly::DRY
            )
            {
                // Direct face cut, owner

                // Grab a point and wet cell and make an IB face
                pointField facePoints = f[faceI].points(p);
                face renumberedFace(facePoints.size());

                // Insert points
                forAll (facePoints, fpI)
                {
                    unmergedPoints.append(facePoints[fpI]);
                    renumberedFace[fpI] = nIbPoints;
                    nIbPoints++;
                }

                // Record the face
                unmergedFaces[nIbCells] = renumberedFace;

                // Collect cut cell index
                ibCells[nIbCells] = owner[faceI];

                // Record the live centre
                ibCellCentres[nIbCells] = C[owner[faceI]];

                // Record the live volume: equal to owner volume
                ibCellVolumes[nIbCells] = V[owner[faceI]];

                // Get span of owner and neighbour
                vector span = cellSpan(owner[faceI]);

                span = Foam::max
                (
                    span,
                    cellSpan(neighbour[faceI])
                );

                // Record the nearest triangle to the face centre
                nearestTri[nIbCells] = tss.nearest(Cf[faceI], span).index();

                nIbCells++;
            }
            else if
            (
                intersectedCell[owner[faceI]] == immersedPoly::DRY
             && intersectedCell[neighbour[faceI]] == immersedPoly::WET
            )
            {
                // Direct face cut, neighbour

                // Grab a point and wet cell and make an IB face
                // Note: reverse face in cut
                pointField facePoints = f[faceI].reverseFace().points(p);

                face renumberedFace(facePoints.size());

                // Insert points
                forAll (facePoints, fpI)
                {
```

```
                unmergedPoints.append(facePoints[fpI]);
                renumberedFace[fpI] = nIbPoints;
                nIbPoints++;
            }

            // Record the face
            unmergedFaces[nIbCells] = renumberedFace;

            // Collect cut cell index
            ibCells[nIbCells] = neighbour[faceI];

            // Record the live centre
            ibCellCentres[nIbCells] = C[neighbour[faceI]];

            // Record the live volume: equal to neighbour volume
            ibCellVolumes[nIbCells] = V[neighbour[faceI]];

            // Get span of neighbour and neighbour
            vector span = cellSpan(neighbour[faceI]);

            span = Foam::max
            (
                span,
                cellSpan(owner[faceI])
            );

            // Record the nearest triangle to the face centre
            nearestTri[nIbCells] = tss.nearest(Cf[faceI], span).index();

            nIbCells++;
        }
    }

    // Check coupled boundaries for direct face cuts

    // Assemble local and neighbour cuts for coupled patches only
    labelListList coupledPatchOwnCut(bMesh.size());
    labelListList coupledPatchNbrCut(bMesh.size());

    // Note: this part requires a rewrite using virtual functions
    // to communicate the cut data from the shadow cell
    // (across the coupled interface) in order to determine
    // the coupled face status.
    // Currently, this is enabled only for processor boundaries.
    // HJ, 28/Dec/2017

    // Send loop
    forAll (bMesh, patchI)
    {
        if (bMesh[patchI].coupled())
        {
            if (isA<processorPolyPatch>(bMesh[patchI]))
            {
                if (Pstream::parRun())
                {
                    const processorPolyPatch& curProcPatch =
                        refCast<const processorPolyPatch>(bMesh[patchI]);

                    // Send internal cut
                    coupledPatchOwnCut[patchI] = labelList
                    (
                        intersectedCell,
                        bMesh[patchI].faceCells()
                    );

                    OPstream toNeighbProc
                    (
                        Pstream::blocking,
                        curProcPatch.neighbProcNo(),
                        sizeof(label)*curProcPatch.size()
```

```
                    );

                    toNeighbProc << coupledPatchOwnCut[patchI];
                }
            }
            else
            {
                // Possible code missing: reconsider Immersed boundary
                // cutting non-matching coupled patches.
                // HJ and HN, 20/Mar/2020
                // WarningInFunction
                //     << "Non-processor coupled patch detected for "
                //     << "immersed boundary.  "
                //     << "Direct face cut may not be detected"
                //     << endl;
            }
        }
    }

    // Receive loop
    forAll (bMesh, patchI)
    {
        if (bMesh[patchI].coupled())
        {
            if (isA<processorPolyPatch>(bMesh[patchI]))
            {
                if (Pstream::parRun())
                {
                    const processorPolyPatch& curProcPatch =
                        refCast<const processorPolyPatch>(bMesh[patchI]);

                    IPstream fromNeighbProc
                    (
                        Pstream::blocking,
                        curProcPatch.neighbProcNo(),
                        sizeof(label)*curProcPatch.size()
                    );

                    coupledPatchNbrCut[patchI] = labelList(fromNeighbProc);
                }
            }
        }
    }

    // Analyse the cut
    forAll (bMesh, patchI)
    {
        if (!coupledPatchOwnCut[patchI].empty())
        {
            const labelList& curOwnCut = coupledPatchOwnCut[patchI];
            const labelList& curNbrCut = coupledPatchNbrCut[patchI];

            const labelList& fc = bMesh[patchI].faceCells();

            forAll (curOwnCut, patchFaceI)
            {
                if
                (
                    curOwnCut[patchFaceI] == immersedPoly::WET
                 && curNbrCut[patchFaceI] == immersedPoly::DRY
                )
                {
                    // Direct face cut, coupled on live side

                    // Get face index.  Note the difference between faceI
                    // and patchFaceI
                    const label faceI = bMesh[patchI].start() + patchFaceI;

                    // Grab a point and wet cell and make an IB face
                    pointField facePoints = f[faceI].points(p);
```

```
                    face renumberedFace(facePoints.size());

                    // Insert points
                    forAll (facePoints, fpI)
                    {
                        unmergedPoints.append(facePoints[fpI]);
                        renumberedFace[fpI] = nIbPoints;
                        nIbPoints++;
                    }

                    // Record the face
                    unmergedFaces[nIbCells] = renumberedFace;

                    // Collect cut cell index
                    ibCells[nIbCells] = fc[patchFaceI];

                    // Record the live centre
                    ibCellCentres[nIbCells] = C[fc[patchFaceI]];

                    // Record the live volume: equal to owner volume
                    ibCellVolumes[nIbCells] = V[fc[patchFaceI]];

                    // Get span of owner.  Cannot reach neighbour
                    vector span = cellSpan(fc[patchFaceI]);

                    // Record the nearest triangle to the face centre
                    nearestTri[nIbCells] =
                        tss.nearest(Cf[faceI], span).index();

                    nIbCells++;
                }
            }
        }
    }

    // Record the number of IB cells for debug
    totalIbCount[0] = nIbCells;

    // Reset the cell lists
    unmergedFaces.setSize(nIbCells);
    ibCells.setSize(nIbCells);
    ibCellCentres.setSize(nIbCells);
    ibCellVolumes.setSize(nIbCells);
    nearestTri.setSize(nIbCells);

    // Check tri addressing
    if (min(nearestTri) == -1)
    {
        FatalErrorInFunction
            << "Cannot find nearestTri for all points"
            << abort(FatalError);
    }

    // Build stand-alone patch
    // Memory management
    {
        unmergedPoints.shrink();

        pointField ibPatchPoints;
        labelList pointMap;

        mergePoints
        (
            unmergedPoints,
            1e-6,               // mergeTol.  Review.  Do not like the algorithm
            false,              // verbose
            pointMap,
            ibPatchPoints
        );

```

```
        // Renumber faces after point merge
        faceList ibPatchFaces(unmergedFaces.size());

        forAll (unmergedFaces, faceI)
        {
            // Get old and new face
            const face& uFace = unmergedFaces[faceI];
            face& rFace = ibPatchFaces[faceI];
            rFace.setSize(uFace.size());
            forAll (uFace, pointI)
            {
                rFace[pointI] = pointMap[uFace[pointI]];
            }
        }

        // Create IB patch from renumbered points and faces
        ibPatchPtr_ = new standAlonePatch(ibPatchFaces, ibPatchPoints);

        if (mesh.time().outputTime())
        {
            Info << "Writing immersed patch as VTK" << endl;

            fileName fvPath(mesh.time().path()/"VTK");
            mkDir(fvPath);

            fileName surfaceFileName
            (
                "immersed" + name() + "_live_"
              + Foam::name(boundaryMesh().mesh().time().timeIndex())
            );

            ibPatchPtr_->writeVTK(fvPath/surfaceFileName);

            fileName normalsFileName
            (
                "normals" + name() + "_live_"
              + Foam::name(boundaryMesh().mesh().time().timeIndex())
            );

            ibPatchPtr_->writeVTKNormals(fvPath/normalsFileName);
        }
    }

    // Count and collect dead cells

    // Memory management
    {
        label nDeadCells = 0;

        forAll (intersectedCell, cellI)
        {
            if (intersectedCell[cellI] == immersedPoly::DRY)
            {
                nDeadCells++;
            }
        }

        // Allocate storage and collect dead cells
        deadCellsPtr_ = new labelList(nDeadCells);
        labelList& dc = *deadCellsPtr_;

        // Reset the counter
        nDeadCells = 0;

        forAll (intersectedCell, cellI)
        {
            if (intersectedCell[cellI] == immersedPoly::DRY)
            {
                dc[nDeadCells] = cellI;
                nDeadCells++;
```

```
            }
        }

        // Record the number of dead cells for debug
        totalIbCount[1] = nDeadCells;
    }

    // IB faces: faces intersected by the IB patch
    // This also corresponds to faceCells next to the IB patch
    ibFacesPtr_ = new labelList(mesh.nFaces());
    labelList& ibFaces = *ibFacesPtr_;

    // IB face centres: centre of live part of the intersected face
    // next to the IB patch
    ibFaceCentresPtr_ = new vectorField(mesh.nFaces());
    vectorField& ibFaceCentres = *ibFaceCentresPtr_;

    // IB face areas: surface-normal area of live part of the intersected face
    // next to the IB patch
    ibFaceAreasPtr_ = new vectorField(mesh.nFaces());
    vectorField& ibFaceAreas = *ibFaceAreasPtr_;
    label nIbFaces = 0;

    // Classify faces
    labelList intersectedFace(mesh.nFaces(), immersedPoly::UNKNOWN);

    // Resolve simple face intersections based on the cell intersection data
    // First, kill all faces touching dead cells, including internal
    // and boundary faces.
    // If a face touches a live cell, it is live
    // The intersection belt will be handled separately by detailed intersection

    // Quick intersection scan: if owner and neighbour are in the same state
    // the face is in the same state

    // Internal faces
    forAll (neighbour, faceI)
    {
        // Wet on wet
        if
        (
            intersectedCell[owner[faceI]] == immersedPoly::WET
         && intersectedCell[neighbour[faceI]] == immersedPoly::WET
        )
        {
            intersectedFace[faceI] = immersedPoly::WET;
        }

        // Dry on dry
        if
        (
            intersectedCell[owner[faceI]] == immersedPoly::DRY
         && intersectedCell[neighbour[faceI]] == immersedPoly::DRY
        )
        {
            intersectedFace[faceI] = immersedPoly::DRY;
        }

        // Wet on cut face must remain wet.  Error in cut cell is fixed
        // by the Marooney Maneouvre.  HJ, 5/Apr/2019
        if
        (
            (
                intersectedCell[owner[faceI]] == immersedPoly::WET
             && intersectedCell[neighbour[faceI]] == immersedPoly::CUT
            )
         || (
                intersectedCell[owner[faceI]] == immersedPoly::CUT
             && intersectedCell[neighbour[faceI]] == immersedPoly::WET
            )
```

```
        )
        {
            intersectedFace[faceI] = immersedPoly::WET;
        }

        // Special check for directly cut faces
        // Wet-to-dry and dry-to-wet is a direct face cut
        // Dry-to-cut or cut-to-dry are cutting errors.  They will be
        // corrected later in corrected face areas, based on closed cell
        // tolerance.  HJ, 11/Dec/2017
        if
        (
            (
                intersectedCell[owner[faceI]] == immersedPoly::WET
             && intersectedCell[neighbour[faceI]] == immersedPoly::DRY
            )
         || (
                intersectedCell[owner[faceI]] == immersedPoly::DRY
             && intersectedCell[neighbour[faceI]] == immersedPoly::WET
             )
         || (
                intersectedCell[owner[faceI]] == immersedPoly::DRY
             && intersectedCell[neighbour[faceI]] == immersedPoly::CUT
             )
         || (
                intersectedCell[owner[faceI]] == immersedPoly::CUT
             && intersectedCell[neighbour[faceI]] == immersedPoly::DRY
             )
        )
        {
            // Note:
            // Wet-to-dry: this face has been declared to be a
            //             cut face and needs to be taken out as live face
            // Cut-to-dry: this is either an outside edge of cut faces or
            //             a cutting error
            intersectedFace[faceI] = immersedPoly::DRY;
        }
    }

    // Boundary faces
    forAll (bMesh, patchI)
    {
        const label patchStart = bMesh[patchI].start();

        if (bMesh[patchI].coupled())
        {
            // Coupled patch: two-sided check
            const labelList& curOwnCut = coupledPatchOwnCut[patchI];
            const labelList& curNbrCut = coupledPatchNbrCut[patchI];

            forAll (curOwnCut, patchFaceI)
            {
                // Wet on wet
                if
                (
                    curOwnCut[patchFaceI] == immersedPoly::WET
                 && curNbrCut[patchFaceI] == immersedPoly::WET
                )
                {
                    intersectedFace[patchStart + patchFaceI] =
                        immersedPoly::WET;
                }

                // Dry on dry
                if
                (
                    curOwnCut[patchFaceI] == immersedPoly::DRY
                 && curNbrCut[patchFaceI] == immersedPoly::DRY
                )
                {
```

```
                    intersectedFace[patchStart + patchFaceI] =
                        immersedPoly::DRY;
                }

                // Wet on cut face must remain wet.  Error in cut cell is fixed
                // by the Marooney Maneouvre.  HJ, 5/Apr/2019
                if
                (
                    (
                        curOwnCut[patchFaceI] == immersedPoly::WET
                     && curNbrCut[patchFaceI] == immersedPoly::CUT
                    )
                 || (
                        curOwnCut[patchFaceI] == immersedPoly::CUT
                     && curNbrCut[patchFaceI] == immersedPoly::WET
                    )
                )
                {
                    intersectedFace[patchStart + patchFaceI] =
                        immersedPoly::WET;
                }

                // Special check for directly cut faces
                // Wet-to-dry and dry-to-wet is a direct face cut
                // Dry-to-cut or cut-to-dry are cutting errors.  They will be
                // corrected later in corrected face areas, based on closed cell
                // tolerance.  HJ, 11/Dec/2017
                if
                (
                    (
                        curOwnCut[patchFaceI] == immersedPoly::WET
                      && curNbrCut[patchFaceI] == immersedPoly::DRY
                    )
                 || (
                        curOwnCut[patchFaceI] == immersedPoly::DRY
                     && curNbrCut[patchFaceI] == immersedPoly::WET
                    )
                 || (
                        curOwnCut[patchFaceI] == immersedPoly::DRY
                     && curNbrCut[patchFaceI] == immersedPoly::CUT
                    )
                 || (
                        curOwnCut[patchFaceI] == immersedPoly::CUT
                     && curNbrCut[patchFaceI] == immersedPoly::DRY
                     )
                )
                {
                    // Note:
                    // Wet-to-dry: this face has been declared to be a
                    // cut face and needs to be taken out as live face
                    // Cut-to-dry: this is either an outside edge of cut faces
                    // or a cutting error
                    intersectedFace[patchStart + patchFaceI] =
                        immersedPoly::DRY;
                }
            }
        }
        else
        {
            // Regular patch: one-sided check
            const labelList& fc = bMesh[patchI].faceCells();

            forAll (fc, patchFaceI)
            {
                if
                (
                    intersectedCell[fc[patchFaceI]] == immersedPoly::WET
                )
                {
                    intersectedFace[patchStart + patchFaceI] =
```

```
                        immersedPoly::WET;
                }

                if
                (
                    intersectedCell[fc[patchFaceI]] == immersedPoly::DRY
                )
                {
                    intersectedFace[patchStart + patchFaceI] =
                        immersedPoly::DRY;
                }
            }
        }
    }

    // Detailed face check after initial rejection scan
    forAll (intersectedFace, faceI)
    {
        if (intersectedFace[faceI] == immersedPoly::UNKNOWN)
        {
            // Possibly intersected face.  Check existance of intersection
            // via points
            const labelList& curF = f[faceI];

            bool foundInside = false;
            bool foundOutside = false;

            forAll (curF, fI)
            {
                if (pointsInside[curF[fI]])
                {
                    // Found a point inside
                    foundInside = true;
                }
                else
                {
                    // Found a points outside
                    foundOutside = true;
                }
            }

            // Check face classification
            if (foundInside && !foundOutside)
            {
                // All points inside: cell is wet
                intersectedFace[faceI] = immersedPoly::WET;
            }
            else if (!foundInside && foundOutside)
            {
                // All points outside: cell is dry
                intersectedFace[faceI] = immersedPoly::DRY;
            }
            else if (foundInside && foundOutside)
            {
                // Real intersection.  Try to cut the face

                // Get search span
                vector span = cellSpan(owner[faceI]);

                // For internal face, check the neighbour span as well
                if (mesh.isInternalFace(faceI))
                {
                    span = Foam::max
                    (
                        span,
                        cellSpan(neighbour[faceI])
                    );
                }

                // Create a cutting object with a local tolerance
```

```
                triSurfaceDistance dist
                (
                    tss,
                    span,
                    internalFlow(),
                    true                // iterate intersection
                );

                // Calculate the intersection
                ImmersedFace<triSurfaceDistance> cutFace
                (
                    faceI,
                    mesh,
                    dist
                );

                if (cutFace.isAllWet())
                {
                    intersectedFace[faceI] = immersedPoly::WET;
                }
                else if (cutFace.isAllDry())
                {
                    intersectedFace[faceI] = immersedPoly::DRY;
                }
                else
                {
                    // Real intesection.  Check cut. Rejection on thin cut is
                    // performed by ImmersedFace.  HJ, 13/Mar/2019
                    const scalar faceFactor =
                        cutFace.wetAreaMag()/mag(S[faceI]);

                    // True intersection.  Collect data
                    intersectedFace[faceI] = immersedPoly::CUT;

                    // Get intersected face index
                    ibFaces[nIbFaces] = faceI;

                    // Get wet centre
                    ibFaceCentres[nIbFaces] = cutFace.wetAreaCentre();

                    // Get wet area, preserving original normal direction
                    ibFaceAreas[nIbFaces] = faceFactor*S[faceI];

                    nIbFaces++;
                }
            }
        }
    }

    // Record the number of IB faces for debug
    totalIbCount[2] = nIbFaces;

    // Reset the sizes of the list
    ibFaces.setSize(nIbFaces);
    ibFaceCentres.setSize(nIbFaces);

    // Count and collect dead faces
    // Memory management
    {
        label nDeadFaces = 0;

        forAll (intersectedFace, faceI)
        {
            if (intersectedFace[faceI] == immersedPoly::DRY)
            {
                nDeadFaces++;
            }
        }

        // Allocate storage and collect dead faces
```

```
        deadFacesPtr_ = new labelList(nDeadFaces);
        labelList& df = *deadFacesPtr_;

        // Reset the counter
        nDeadFaces = 0;

        forAll (intersectedFace, faceI)
        {
            if (intersectedFace[faceI] == immersedPoly::DRY)
            {
                df[nDeadFaces] = faceI;
                nDeadFaces++;
            }
        }

        // Record the number of dead faces for debug
        totalIbCount[3] = nDeadFaces;
    }

    // Reduce is not allowed in parallel load balancing
    // HJ, 24/Oct/2018
    if (debug)
    {
        // reduce(totalIbCount, sumOp<List<label> >());

        InfoInFunction
            << "Finished calcImmersedBoundary"
            << endl;

        Pout<< "Immersed boundary " << name() << " info: "
            << "nIbCells: " << totalIbCount[0]
            << " nDeadCells: " << totalIbCount[1]
            << " nIbFaces: " << totalIbCount[2]
            << " nDeadFaces: " << totalIbCount[3]
            << endl;
    }
}


void Foam::immersedBoundaryPolyPatch::calcCorrectedGeometry() const
{
    if (debug)
    {
        InfoInFunction
            << "Calculating corrected geometry"
            << endl;
    }

    // Corrected patch face areas are in a separate storage per patch
    // Use it to signal if the function has been called
    if (correctedIbPatchFaceAreasPtr_)
    {
        FatalErrorInFunction
            << "Corrected geometry already calculated"
            << abort(FatalError);
    }

    // Get mesh reference
    const polyMesh& mesh = boundaryMesh().mesh();

    // Get mesh geometry from polyMesh.  It will be modified
    vectorField& C =
        const_cast<vectorField&>(boundaryMesh().mesh().cellCentres());

    vectorField& Cf =
        const_cast<vectorField&>(boundaryMesh().mesh().faceCentres());

    scalarField& V =
        const_cast<scalarField&>(boundaryMesh().mesh().cellVolumes());

```

```
    vectorField& Sf =
        const_cast<vectorField&>(boundaryMesh().mesh().faceAreas());


    // Initialise IB patch face areas with the areas of the stand-alone patch
    // They will be corrected using the Marooney Maneouvre
    correctedIbPatchFaceAreasPtr_ = new vectorField(ibPatch().areas());
    vectorField& ibSf = *correctedIbPatchFaceAreasPtr_;

    // Correct for all cut cells

    // Get cut cells
    const labelList& cutCells = ibCells();
    const vectorField& cutCellCentres = ibCellCentres();
    const scalarField& cutCellVolumes = ibCellVolumes();

    forAll (cutCells, ccI)
    {
        // Correct the volume and area
        C[cutCells[ccI]] = cutCellCentres[ccI];

        V[cutCells[ccI]] = cutCellVolumes[ccI];
    }

    // Deactivate dead cells
    const labelList& dc = deadCells();

    forAll (dc, dcI)
    {
        // Scale dead volume to small
        V[dc[dcI]] *= SMALL;
    }

    // Correct for all cut faces

    // Get cut faces
    const labelList& cutFaces = ibFaces();
    const vectorField& cutFaceCentres = ibFaceCentres();
    const vectorField& cutFaceAreas = ibFaceAreas();

    forAll (cutFaces, cfI)
    {
        Cf[cutFaces[cfI]] = cutFaceCentres[cfI];

        // Preserve the original face normal
        Sf[cutFaces[cfI]] = cutFaceAreas[cfI];
    }

    // Deactivate dead faces
    const labelList& df = deadFaces();

    forAll (df, dfI)
    {
        // Scale dead area to small
        Sf[df[dfI]] *= SMALL;
    }

    // In case of cutting errors due to finite tolerance, some cut cells may
    // remain opened and have to be closed by force.  This will be achieved
    // by the Marooney Maneouvre, where the face sum imbalance is compensated
    // in the cut face.  HJ, 11/Dec/2017

    const labelList& owner = mesh.faceOwner();

    label nMarooneyCells = 0;

    // Get valid directions to avoid round-off errors in 2-D cases
    const Vector<label> dirs = mesh.geometricD();
    vector validDirs = vector::zero;

```

```
    for (direction cmpt = 0; cmpt < Vector<label>::nComponents; cmpt++)
    {
        if (dirs[cmpt] > 0)
        {
            validDirs[cmpt] = 1;
        }
    }

    forAll (cutCells, cutCellI)
    {
        const label ccc = cutCells[cutCellI];

        // Calculate sum Sf and sumMagSf for the cell
        const cell& curCell = mesh.cells()[ccc];

        vector curSumSf = vector::zero;
        scalar curSumMagSf = 0;

        // Collect from regular faces
        forAll (curCell, cfI)
        {
            const vector& curSf = Sf[curCell[cfI]];

            // Check owner/neighbour
            if (owner[curCell[cfI]] == ccc)
            {
                curSumSf += curSf;
            }
            else
            {
                curSumSf -= curSf;
            }

            curSumMagSf += mag(curSf);
        }

        // Add cut face only into mag.  The second part is handled in the
        // if-statement
        curSumMagSf += mag(ibSf[cutCellI]);

        // Adjustment is peformed when the openness is greater than a certain
        // fraction of surface area.  Criterion by IG, 13/Mar/2019
        // Switched to using absolute check from primitiveMeshCheck.
        // HJ, 13/Mar/2019
        // if (mag(curSumSf + ibSf[cutCellI]) > 1e-6*curSumMagSf)
        if (mag(curSumSf + ibSf[cutCellI]) > primitiveMesh::closedThreshold_)
        {
            if (debug)
            {
                Pout<< "Marooney Maneouvre for cell " << ccc
                    << " error: " << curSumSf + ibSf[cutCellI] << " "
                    << " V: " << cutCellVolumes[cutCellI]
                    << " Sf: " << ibSf[cutCellI]
                    << " corr S: " << curSumSf << endl;
            }

            nMarooneyCells++;

            // Create IB face to ideally close the cell
            ibSf[cutCellI] = cmptMultiply(validDirs, -curSumSf);
        }
    }

    if (debug)
    {
        if (nMarooneyCells > 0)
        {
            InfoInFunction
                << "Marooney Maneouvre used for " << nMarooneyCells
                << " out of " << cutCells.size()
```

```
                    << endl;
            }
        }

        if (min(mag(ibSf)) < SMALL)
        {
            WarningInFunction
                << "Minimum IB face area for patch " << name()
                << ": " << min(mag(ibSf)) << ".  Possible cutting error.  "
                << "Review immersed boundary tolerances."
                << endl;
        }

        if (debug)
        {
            InfoInFunction
                << "Finished calculating corrected geometry"
                << endl;
        }
}


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

void Foam::immersedBoundaryPolyPatch::initAddressing()
{
    // Force calculation of mesh directions before comms
    // This is needed in immersed boundary calculation and should not
    // interfere with other comms
    // HJ, 17/Sep/2021
    boundaryMesh().mesh().geometricD();

    calcImmersedBoundary();
}


void Foam::immersedBoundaryPolyPatch::initGeometry()
{
    calcCorrectedGeometry();
}


void Foam::immersedBoundaryPolyPatch::movePoints(const pointField& p)
{
    if (debug)
    {
        InfoInFunction
            << "Moving mesh: immersedBoundary update"
            << endl;
    }

    // Handle motion of the mesh for new immersed boundary position
    if (ibUpdateTimeIndex_ < boundaryMesh().mesh().time().timeIndex())
    {
        // New motion in the current time step.  Clear
        ibUpdateTimeIndex_ = boundaryMesh().mesh().time().timeIndex();

        clearOut();
    }

    polyPatch::movePoints(p);
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::immersedBoundaryPolyPatch::immersedBoundaryPolyPatch
(
    const word& name,
    const label size,
```

```
    const label start,
    const label index,
    const polyBoundaryMesh& bm
)
:
    polyPatch(name, size, start, index, bm),
    ibMesh_
    (
        IOobject
        (
            name + ".ftr",
            bm.mesh().time().constant(), // instance
            "triSurface",                // local
            bm.mesh().parent(),          // registry
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        )
    ),
    internalFlow_(false),
    isWall_(true),
    movingIb_(false),
    ibUpdateTimeIndex_(-1),
    triSurfSearchPtr_(nullptr),
    ibPatchPtr_(nullptr),
    ibCellsPtr_(nullptr),
    ibCellCentresPtr_(nullptr),
    ibCellVolumesPtr_(nullptr),
    ibFacesPtr_(nullptr),
    ibFaceCentresPtr_(nullptr),
    ibFaceAreasPtr_(nullptr),
    nearestTriPtr_(nullptr),
    deadCellsPtr_(nullptr),
    deadFacesPtr_(nullptr),
    correctedIbPatchFaceAreasPtr_(nullptr),
    oldIbPointsPtr_(nullptr)
{}


Foam::immersedBoundaryPolyPatch::immersedBoundaryPolyPatch
(
    const word& name,
    const dictionary& dict,
    const label index,
    const polyBoundaryMesh& bm
)
:
    polyPatch(name, dict, index, bm),
    ibMesh_
    (
        IOobject
        (
            name + ".ftr",
            bm.mesh().time().constant(), // instance
            "triSurface",                // local
            bm.mesh().parent(),          // read from parent registry
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    ),
    internalFlow_(dict.lookup("internalFlow")),
    isWall_(dict.lookup("isWall")),
    movingIb_(false),
    ibUpdateTimeIndex_(-1),
    triSurfSearchPtr_(nullptr),
    ibPatchPtr_(nullptr),
    ibCellsPtr_(nullptr),
    ibCellCentresPtr_(nullptr),
    ibCellVolumesPtr_(nullptr),
    ibFacesPtr_(nullptr),
    ibFaceCentresPtr_(nullptr),
```

```
    ibFaceAreasPtr_(nullptr),
    nearestTriPtr_(nullptr),
    deadCellsPtr_(nullptr),
    deadFacesPtr_(nullptr),
    correctedIbPatchFaceAreasPtr_(nullptr),
    oldIbPointsPtr_(nullptr)
{
    if (size() > 0)
    {
        FatalIOErrorInFunction(dict)
            << "Faces detected in the immersedBoundaryPolyPatch.  "
            << "This is not allowed: please make sure that the patch size "
            << "equals zero."
            << abort(FatalIOError);
    }
}


Foam::immersedBoundaryPolyPatch::immersedBoundaryPolyPatch
(
    const immersedBoundaryPolyPatch& pp,
    const polyBoundaryMesh& bm,
    const label index,
    const label newSize,
    const label newStart
)
:
    polyPatch(pp, bm, index, newSize, newStart),
    ibMesh_
    (
        IOobject
        (
            pp.name() + ".ftr",
            bm.mesh().time().constant(), // instance
            "triSurface",                // local
            bm.mesh().parent(),          // parent registry
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        pp.ibMesh()   // Take ibMesh from pp
    ),
    internalFlow_(pp.internalFlow_),
    isWall_(pp.isWall_),
    movingIb_(false),
    ibUpdateTimeIndex_(-1),
    triSurfSearchPtr_(nullptr),
    ibPatchPtr_(nullptr),
    ibCellsPtr_(nullptr),
    ibCellCentresPtr_(nullptr),
    ibCellVolumesPtr_(nullptr),
    ibFacesPtr_(nullptr),
    ibFaceCentresPtr_(nullptr),
    ibFaceAreasPtr_(nullptr),
    nearestTriPtr_(nullptr),
    deadCellsPtr_(nullptr),
    deadFacesPtr_(nullptr),
    correctedIbPatchFaceAreasPtr_(nullptr),
    oldIbPointsPtr_(nullptr)
{}


Foam::immersedBoundaryPolyPatch::immersedBoundaryPolyPatch
(
    const immersedBoundaryPolyPatch& pp
)
:
    polyPatch(pp),
    ibMesh_
    (
        IOobject
```

```
        (
            pp.name() + ".ftr",
            pp.boundaryMesh().mesh().time().constant(), // instance
            "triSurface",                               // local
            pp.boundaryMesh().mesh().parent(),          // parent registry
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        pp.ibMesh()   // Take ibMesh from pp
    ),
    internalFlow_(pp.internalFlow_),
    isWall_(pp.isWall_),
    movingIb_(false),
    ibUpdateTimeIndex_(-1),
    triSurfSearchPtr_(nullptr),
    ibPatchPtr_(nullptr),
    ibCellsPtr_(nullptr),
    ibCellCentresPtr_(nullptr),
    ibCellVolumesPtr_(nullptr),
    ibFacesPtr_(nullptr),
    ibFaceCentresPtr_(nullptr),
    ibFaceAreasPtr_(nullptr),
    nearestTriPtr_(nullptr),
    deadCellsPtr_(nullptr),
    deadFacesPtr_(nullptr),
    correctedIbPatchFaceAreasPtr_(nullptr),
    oldIbPointsPtr_(nullptr)
{}


Foam::immersedBoundaryPolyPatch::immersedBoundaryPolyPatch
(
    const immersedBoundaryPolyPatch& pp,
    const polyBoundaryMesh& bm
)
:
    polyPatch(pp, bm),
    ibMesh_
    (
        IOobject
        (
            pp.name() + ".ftr",
            bm.mesh().time().constant(), // instance
            "triSurface",                // local
            bm.mesh().parent(),          // parent registry
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        pp.ibMesh()   // Take ibMesh from pp
    ),
    internalFlow_(pp.internalFlow_),
    isWall_(pp.isWall_),
    movingIb_(false),
    ibUpdateTimeIndex_(-1),
    triSurfSearchPtr_(nullptr),
    ibPatchPtr_(nullptr),
    ibCellsPtr_(nullptr),
    ibCellCentresPtr_(nullptr),
    ibCellVolumesPtr_(nullptr),
    ibFacesPtr_(nullptr),
    ibFaceCentresPtr_(nullptr),
    ibFaceAreasPtr_(nullptr),
    nearestTriPtr_(nullptr),
    deadCellsPtr_(nullptr),
    deadFacesPtr_(nullptr),
    correctedIbPatchFaceAreasPtr_(nullptr),
    oldIbPointsPtr_(nullptr)
{}
```

```
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

Foam::immersedBoundaryPolyPatch::~immersedBoundaryPolyPatch()
{
    clearOut();

    deleteDemandDrivenData(oldIbPointsPtr_);
}


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

const Foam::triSurfaceSearch&
Foam::immersedBoundaryPolyPatch::triSurfSearch() const
{
    if (!triSurfSearchPtr_)
    {
        calcTriSurfSearch();
    }

    return *triSurfSearchPtr_;
}

const Foam::standAlonePatch&
Foam::immersedBoundaryPolyPatch::ibPatch() const
{
    if (!ibPatchPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibPatchPtr_;
}


const Foam::labelList&
Foam::immersedBoundaryPolyPatch::ibCells() const
{
    if (!ibCellsPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibCellsPtr_;
}


const Foam::vectorField&
Foam::immersedBoundaryPolyPatch::ibCellCentres() const
{
    if (!ibCellCentresPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibCellCentresPtr_;
}


const Foam::scalarField&
Foam::immersedBoundaryPolyPatch::ibCellVolumes() const
{
    if (!ibCellVolumesPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibCellVolumesPtr_;
}

```

```
const Foam::labelList&
Foam::immersedBoundaryPolyPatch::ibFaces() const
{
    if (!ibFacesPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibFacesPtr_;
}


const Foam::vectorField&
Foam::immersedBoundaryPolyPatch::ibFaceCentres() const
{
    if (!ibFaceCentresPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibFaceCentresPtr_;
}


const Foam::vectorField&
Foam::immersedBoundaryPolyPatch::ibFaceAreas() const
{
    if (!ibFaceAreasPtr_)
    {
        calcImmersedBoundary();
    }

    return *ibFaceAreasPtr_;
}


const Foam::labelList&
Foam::immersedBoundaryPolyPatch::nearestTri() const
{
    if (!nearestTriPtr_)
    {
        calcImmersedBoundary();
    }

    return *nearestTriPtr_;
}


const Foam::labelList&
Foam::immersedBoundaryPolyPatch::deadCells() const
{
    if (!deadCellsPtr_)
    {
        calcImmersedBoundary();
    }

    return *deadCellsPtr_;
}


const Foam::labelList&
Foam::immersedBoundaryPolyPatch::deadFaces() const
{
    if (!deadFacesPtr_)
    {
        calcImmersedBoundary();
    }

    return *deadFacesPtr_;
```

```
}


const Foam::vectorField&
Foam::immersedBoundaryPolyPatch::correctedIbPatchFaceAreas() const
{
    if (!correctedIbPatchFaceAreasPtr_)
    {
        calcCorrectedGeometry();
    }

    return *correctedIbPatchFaceAreasPtr_;
}


const Foam::pointField&
Foam::immersedBoundaryPolyPatch::oldIbPoints() const
{
    if (!oldIbPointsPtr_)
    {
        // The mesh has never moved: old points are equal to current points
        ibUpdateTimeIndex_ = boundaryMesh().mesh().time().timeIndex();

        oldIbPointsPtr_ = new pointField(ibMesh_.points());
    }

    return *oldIbPointsPtr_;
}

Foam::tmp<Foam::vectorField>
Foam::immersedBoundaryPolyPatch::triMotionDistance() const
{
    // Calculate the distance between new and old coordinates on
    // the ibPatch face centres

    // Calculate the motion on the triangular mesh face centres
    return ibMesh_.coordinates()
      - PrimitivePatch<labelledTri, List, const pointField&>
        (
            ibMesh_,
            oldIbPoints()
        ).faceCentres();
}


Foam::tmp<Foam::vectorField>
Foam::immersedBoundaryPolyPatch::motionDistance() const
{
    // Interpolate the values from tri surface using nearest triangle
    return tmp<vectorField>
    (
        new vectorField(triMotionDistance(), nearestTri())
    );
}


void Foam::immersedBoundaryPolyPatch::moveTriSurfacePoints
(
    const pointField& p
)
{
    // Record the motion of the patch
    movingIb_ = true;

    // Move points of the triSurface
    const pointField& oldPoints = ibMesh_.points();

    if (oldPoints.size() != p.size())
    {
        FatalErrorInFunction
```

```
            << "Incorrect size of motion points for patch " << name()
            << ".  oldPoints = "
            << oldPoints.size() << " p = " << p.size()
            << abort(FatalError);
    }

    if (ibUpdateTimeIndex_ < boundaryMesh().mesh().time().timeIndex())
    {
        // New motion in the current time step.  Store old points
        ibUpdateTimeIndex_ = boundaryMesh().mesh().time().timeIndex();

        deleteDemandDrivenData(oldIbPointsPtr_);

        Info<< "Storing old points for time index " << ibUpdateTimeIndex_
            << endl;
        oldIbPointsPtr_ = new pointField(oldPoints);
    }

    Info<< "Moving immersed boundary points for patch " << name()
        << endl;

    ibMesh_.movePoints(p);

    if (boundaryMesh().mesh().time().outputTime())
    {
        fileName path(boundaryMesh().mesh().time().path()/"VTK");

        mkDir(path);
        ibMesh_.triSurface::write
        (
            path/
            word
            (
                name() + "_tri_"
              + Foam::name(boundaryMesh().mesh().time().timeIndex())
              + ".stl"
            )
        );
    }

    // Note: the IB patch is now in the new position, but the mesh has not
    // been updated yet.  movePoints() needs to be executed to update the
    // fv mesh data
}


void Foam::immersedBoundaryPolyPatch::clearGeom()
{
    clearOut();
}


void Foam::immersedBoundaryPolyPatch::clearAddressing()
{
    clearOut();
}


void Foam::immersedBoundaryPolyPatch::clearOut() const
{
    if (debug)
    {
        InfoInFunction
            << "Clear immersed boundary for patch "
            << name() << " for mesh "
            << boundaryMesh().mesh().time().path()
            << endl;
    }

    deleteDemandDrivenData(triSurfSearchPtr_);
```

```

    deleteDemandDrivenData(ibPatchPtr_);
    deleteDemandDrivenData(ibCellsPtr_);
    deleteDemandDrivenData(ibCellCentresPtr_);
    deleteDemandDrivenData(ibCellVolumesPtr_);
    deleteDemandDrivenData(ibFacesPtr_);
    deleteDemandDrivenData(ibFaceCentresPtr_);
    deleteDemandDrivenData(ibFaceAreasPtr_);
    deleteDemandDrivenData(nearestTriPtr_);
    deleteDemandDrivenData(deadCellsPtr_);
    deleteDemandDrivenData(deadFacesPtr_);

    deleteDemandDrivenData(correctedIbPatchFaceAreasPtr_);

    // Warning.  This function should also clear the geometry in polyMesh
    // to avoid double cutting of polyMesh geometry data.
    // This is protected by the presence of correctedIbPatchFaceAreasPtr_
    // pointer, but may possibly go wrong.
    // HJ, 11/May/2022
    // boundaryMesh().mesh().clearOut();

    // Cannot delete old motion points.  HJ, 10/Dec/2017
}


void Foam::immersedBoundaryPolyPatch::write(Ostream& os) const
{
    polyPatch::write(os);
    os.writeKeyword("internalFlow") << internalFlow_
        << token::END_STATEMENT << nl;
    os.writeKeyword("isWall") << isWall_
        << token::END_STATEMENT << nl;
}


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

B

immersedBoundaryFvPatch.C

Listing B.1: immersedBoundaryFvPatch.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

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

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

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

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

#include "fvMesh.H"
#include "volFields.H"
#include "surfaceFields.H"
#include "slicedVolFields.H"
#include "slicedSurfaceFields.H"
#include "immersedBoundaryFvPatch.H"
#include "emptyFvPatch.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(immersedBoundaryFvPatch, 1);

    addToRunTimeSelectionTable(fvPatch, immersedBoundaryFvPatch, polyPatch);
}

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

const Foam::debug::tolerancesSwitch
Foam::immersedBoundaryFvPatch::nonOrthogonalFactor_
(
    "immersedBoundaryNonOrthogonalFactor",
    0.1
);


```

```
// * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * * //

void Foam::immersedBoundaryFvPatch::makeCf(slicedSurfaceVectorField& Cf) const
{
    // Insert the patch data for the immersed boundary
    // Note: use the face centres from the stand-alone patch within the IB
    // HJ, 30/Nov/2017
    // Inserting only local data
    Cf.boundaryField()[index()].UList::operator=
    (
        ibPolyPatch().ibPatch().faceCentres()
    );
}


void Foam::immersedBoundaryFvPatch::makeSf(slicedSurfaceVectorField& Sf) const
{
    // Insert the patch data for the immersed boundary
    // Note: use the corrected face areas from immersed boundary instead of
    // the stand-alone patch areas within the IB
    // HJ, 30/Nov/2017
    // Inserting only local data
    Sf.boundaryField()[index()].UList::operator=
    (
        ibPolyPatch().correctedIbPatchFaceAreas()
    );
}


void Foam::immersedBoundaryFvPatch::makeC(slicedVolVectorField& C) const
{
    // Insert the patch data for the immersed boundary
    // Note: use the face centres from the stand-alone patch within the IB
    // HJ, 30/Nov/2017
    // Inserting only local data
    C.boundaryField()[index()].UList::operator=
    (
        ibPolyPatch().ibPatch().faceCentres()
    );
}


void Foam::immersedBoundaryFvPatch::makeV(scalarField& V) const
{}


void Foam::immersedBoundaryFvPatch::updatePhi
(
    DimensionedField<scalar, volMesh>& V,
    DimensionedField<scalar, volMesh>& V0,
    surfaceScalarField& phi
) const
{
    // Correct face fluxes for cut area and insert the immersed patch fluxes

    const fvMesh& mesh = boundaryMesh().mesh();

    const polyBoundaryMesh& bm = boundaryMesh().mesh().boundaryMesh();

    scalar deltaT = mesh.time().deltaT().value();
    scalar rDeltaT = 1.0/deltaT;


    // Scaling of internal mesh flux field should be done only for the current
    // ib patch to avoid scaling multiple times in case of multiple Ib patches
    // present. (IG 3/Dec/2018)

    // Scale internalField
    scalarField& phiIn = phi.internalField();

```

```
    const labelList& deadFaces = ibPolyPatch_.deadFaces();
    forAll (deadFaces, dfI)
    {
        const label faceI = deadFaces[dfI];
        if (mesh.isInternalFace(faceI))
        {
            phiIn[faceI] = scalar(0);
        }
        else
        {
            // Boundary face
            const label patchID = bm.whichPatch(faceI);

            if (!isA<emptyFvPatch>(boundaryMesh()[patchID]))
            {
                const label faceID = bm[patchID].whichFace(faceI);

                phi.boundaryField()[patchID][faceID] = scalar(0);
            }
        }
    }

    // Multiply the raw mesh motion flux with the masking function

    const pointField& points = mesh.points();
    const faceList& faces = mesh.faces();

    const vectorField& faceAreas = mesh.faceAreas();

    const labelList& cutFaces = ibPolyPatch_.ibFaces();
    forAll (cutFaces, cfI)
    {
        const label faceI = cutFaces[cfI];

        const scalar ibAreaRatio =
            mag(faceAreas[faceI])/faces[faceI].mag(points);

        if (mesh.isInternalFace(faceI))
        {
            // Multiply by masking function
            phiIn[faceI] *= ibAreaRatio;
        }
        else
        {
            // Boundary face
            const label patchID = bm.whichPatch(faceI);

            if (!isA<emptyFvPatch>(boundaryMesh()[patchID]))
            {
                const label faceID = bm[patchID].whichFace(faceI);

                phi.boundaryField()[patchID][faceID] *= ibAreaRatio;
            }
        }
    }

    // Immersed boundary patch
    // Calculate the mesh motion flux from the old and new coordinate of
    // triangular face centres and the time step dotted with the new face area
    phi.boundaryField()[index()] =
    (
        ibPolyPatch_.motionDistance()
      & ibPolyPatch_.correctedIbPatchFaceAreas()
    )*rDeltaT;

    // Check and adjust the immersed boundary space conservation law
    // The mesh motion fluxes come from the actual mesh motion or the motion
    // of the immersed boundary
    // The new cell volumes come from the current mesh configuration
    // The space conservation law will be satisfied by adjusting either
```

```
    // the old or the new cell volume.  HJ, 15/Dec/2017

    // First sum up all the fluxes
    scalarField divPhi(mesh.nCells(), 0);

    const unallocLabelList& owner = mesh.owner();
    const unallocLabelList& neighbour = mesh.neighbour();

    forAll (owner, faceI)
    {
        divPhi[owner[faceI]] += phiIn[faceI];
        divPhi[neighbour[faceI]] -= phiIn[faceI];
    }

    // Add the mesh motion fluxes from all patches including immersed boundary
    forAll (mesh.boundary(), patchI)
    {
        const unallocLabelList& pFaceCells =
            mesh.boundary()[patchI].faceCells();

        const scalarField& pssf = phi.boundaryField()[patchI];

        // Check for size since uninitialised ib patches can have zero size at
        // this point (IG 7/Nov/2018)
        if (pssf.size() > 0)
        {
            forAll (pFaceCells, faceI)
            {
                divPhi[pFaceCells[faceI]] += pssf[faceI];
            }
        }
    }

    // Use corrected cell volume
    scalarField& newVols = V.field();
    scalarField& oldVols = V0.field();

    // Multiply by the time-step size and add new volume
    scalarField magDivPhi = mag((newVols - oldVols)*rDeltaT - divPhi);

    // Note:
    // The immersed boundary is now in the new position.  Therefore, some
    // cells that were cut are no longer in the contact with the IB, meaning
    // that ALL cells need to be checked and corrected
    // HJ, 22/Dec/2017
    forAll (magDivPhi, cellI)
    {
        // if (magDivPhi[cellI] > SMALL)
        if (magDivPhi[cellI] > 1e-40)
        {
            // Attempt to correct via old volume
            scalar corrOldVol = newVols[cellI] - divPhi[cellI]*deltaT;

            // Pout<< "Flux maneouvre for cell " << cellI << ": "
            //     << " error: " << magDivPhi[cellI]
            //     << " V: " << newVols[cellI]
            //     << " V0: " << oldVols[cellI]
            //     << " divPhi: " << divPhi[cellI];

            if (corrOldVol < SMALL)
            {
                // Update new volume because old volume cannot carry
                // the correction
                newVols[cellI] = oldVols[cellI] + divPhi[cellI]*deltaT;
            }
            else
            {
                oldVols[cellI] = corrOldVol;
            }

```

```
            // scalar corrDivMeshPhi =
            //     mag((newVols[cellI] - oldVols[cellI]) - divPhi[cellI]*deltaT);
            // Pout<< " Corrected: " << corrDivMeshPhi << endl;
        }
    }
}


void Foam::immersedBoundaryFvPatch::makeDeltaCoeffs
(
    fvsPatchScalarField& dc
) const
{
    const vectorField d = delta();

    dc = 1.0/max((nf() & d), 0.05*mag(d));
}


void Foam::immersedBoundaryFvPatch::makeCorrVecs(fvsPatchVectorField& cv) const
{
    // Set patch non-orthogonality correction to zero on the patch
    cv = vector::zero;

    // Kill correction vectors in dead cells
    // Potential problem: cannot kill correction vectors on coupled boundaries
    // because the are set later.  For the moment, only the internal
    // correction vectors are killed.
    // HJ, 3/May/2022

    vectorField& cvIn = const_cast<vectorField&>(cv.internalField());

    // Get dead faces
    const labelList& deadFaces = ibPolyPatch_.deadFaces();

    const fvMesh& mesh = boundaryMesh().mesh();

    forAll (deadFaces, dfI)
    {
        if (mesh.isInternalFace(deadFaces[dfI]))
        {
            cvIn[deadFaces[dfI]] = vector::zero;
        }
    }
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::immersedBoundaryFvPatch::immersedBoundaryFvPatch
(
    const polyPatch& patch,
    const fvBoundaryMesh& bm
)
:
    fvPatch(patch, bm),
    ibPolyPatch_(refCast<const immersedBoundaryPolyPatch>(patch)),
    mesh_(bm.mesh())
{}


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

Foam::label Foam::immersedBoundaryFvPatch::size() const
{
    // Immersed boundary patch size equals to the number of intersected cells
    // HJ, 28/Nov/2017

    // Note: asking for patch size triggers the cutting which involves
    // parallel communication.  This should be avoided under read/write, ie
```

```
    // when the ibPolyPatch_ is not initialised.
    // Initialisation happens when the fvMesh is initialised, which should be
    // sufficient
    //  HJ, 12/Dec/2018
    // if (!ibPolyPatch_.active())
    // {
    //     return 0;
    // }

    return ibPolyPatch_.ibCells().size();
}


const Foam::unallocLabelList&
Foam::immersedBoundaryFvPatch::faceCells() const
{
    return ibPolyPatch_.ibCells();
}


Foam::tmp<Foam::vectorField> Foam::immersedBoundaryFvPatch::nf() const
{
    // The algorithm has been changed because basic IB patch information
    // (nf and delta) is used in assembly of derived information
    // (eg. deltaCoeffs) and circular dependency needs to be avoided.
    // nf and delta vectors shall be calculated directly from the intersected
    // patch.  HJ, 21/Mar/2019

    return ibPolyPatch_.ibPatch().faceNormals();
}


Foam::tmp<Foam::vectorField> Foam::immersedBoundaryFvPatch::delta() const
{
    // Not strictly needed: this is for debug only.  HJ, 5/Apr/2019
    return ibPolyPatch_.ibPatch().faceCentres() - Cn();
}


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

C

ImmersedCell.C

Listing C.1: ImmersedCell.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

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

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

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

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

#include "ImmersedCell.H"
#include "plane.H"
#include "transform.H"
#include "SortableList.H"
#include "tetPointRef.H"

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

template<class Distance>
void Foam::ImmersedCell<Distance>::getBase
(
    const vector& n,
    vector& e0,
    vector& e1
) const
{
    // Copy from class: geomCellLooper

    // Guess for vector normal to n.
    vector base(1, 0, 0);

    scalar nComp = n & base;

    if (mag(nComp) > 0.8)
    {
        // Was bad guess. Try with different vector.

        base.x() = 0;
```

```
        base.y() = 1;

        nComp = n & base;

        if (mag(nComp) > 0.8)
        {
            base.y() = 0;
            base.z() = 1;

            nComp = n & base;
        }
    }

    // Use component normal to n as base vector.
    e0 = base - nComp*n;

    e0 /= mag(e0) + VSMALL;

    e1 = n ^ e0;
}


template<class Distance>
void Foam::ImmersedCell<Distance>::insertIntersectionPoints()
{
    // Get list of edges
    const edgeList& edges = this->edges();

    // Get edge-face addressing
    const labelListList& edgeFaces = this->edgeFaces();

    // There may be an extra point on every edge.  Resize the list of points
    const label oldSize = points_.size();
    points_.setSize(oldSize + edges.size());
    label nPoints = oldSize;

    // Loop through all edges
    forAll (edges, edgeI)
    {
        // Get reference to currentEdge
        const edge& curEdge = edges[edgeI];

        const label start = curEdge.start();
        const label end = curEdge.end();

        const scalar edgeLength = mag(points_[end] - points_[start]);

        // Check if there is a legitimate cut to be found
        // Note: synced tolerances in ImmersedCell and ImmersedFace
        // HJ, 13/Mar/2019
        if
        (
            depth_[start]*depth_[end] < 0
         && edgeLength > SMALL
         && mag(depth_[start]) > edgeLength*immersedPoly::tolerance_()
         && mag(depth_[end]) > edgeLength*immersedPoly::tolerance_()
        )
        {
            // Prepare a new point to insert
            point cutPoint;
            scalar depthAtCut = 0;

            // Intersection is along the edge length (pf[end] - pf[start])
            // times the ratio of the depth at start and the difference
            // between depth at start and end; add to this the start point
            // and you have the location
            cutPoint =
                points_[start]
              + depth_[start]/(depth_[start] - depth_[end])*
                (points_[end] - points_[start]);
```

```

            // Execute iterative cut if necessary
            if (dist_.iterateDistance())
            {
                // Initialize bisection starting points
                point p0 = points_[start];
                point p1 = points_[end];

                // Depth at starting points
                scalar d0 = depth_[start];
                scalar d1 = depth_[end];

                // Convergence criterion is the depth at newP
                depthAtCut = dist_.distance(cutPoint);

                // initialize loop counter
                label iters = 0;

                while
                (
                    (mag(depthAtCut) > immersedPoly::tolerance_())
                 && (iters < immersedPoly::nIter_())
                )
                {
                    // is the guessed point on the same side of the surface
                    // as p0? If yes, move p0 to the guessed point and thus
                    // shorten the interval
                    if (sign(depthAtCut) == sign(d0))
                    {
                        d0 = depthAtCut;
                        p0 = cutPoint;
                    }
                    // Otherwise, shorten the other side
                    else
                    {
                        d1 = depthAtCut;
                        p1 = cutPoint;
                    }

                    // Determine new intersection point and its depth
                    cutPoint = p0 + mag(d0)/(mag(d0) + mag(d1))*(p1 - p0);

                    depthAtCut = dist_.distance(cutPoint);

                    iters++;
                }
            }

            // Store the newly found cut point
            points_[nPoints] = cutPoint;

            // Find faces connected to edge
            const labelList& edgeFaceIDs = edgeFaces[edgeI];

            // Add the new point to each connected face at the right position!
            forAll (edgeFaceIDs, edgeFaceI)
            {
                // Get old face
                const face& oldFace = faces_[edgeFaceIDs[edgeFaceI]];

                // Make new face with one extra label
                face newFace(oldFace.size() + 1);

                // Count points added to new face
                label nfp = 0;

                // Loop through old face.  If this edge is found, add the
                // cut point label into the edge
                forAll (oldFace, fpI)
                {
```

```
                        // Add the point
                        newFace[nfp] = oldFace[fpI];
                        nfp++;

                        const label curPoint = oldFace[fpI];
                        const label nextPoint = oldFace.nextLabel(fpI);

                        if
                        (
                            (curPoint == start && nextPoint == end)
                         || (curPoint == end && nextPoint == start)
                        )
                        {
                            // Found the edge.  Inser the point
                            newFace[nfp] = nPoints;
                            nfp++;
                        }
                    }

                    // Debug: check if point insertion was successful
                    if (nfp < newFace.size())
                    {
                        FatalErrorInFunction
                            << "badInsertion"
                            << abort(FatalError);
                    }

                    faces_[edgeFaceIDs[edgeFaceI]] = newFace;
                }

                // Finished point insertion
                nPoints++;
            }
        }

    // Resize the points list
    points_.setSize(nPoints);

    // Extra depths are all zero
    depth_.setSize(nPoints);

    // For all cut points set depth to exactly zero
    for (label i = oldSize; i < depth_.size(); i++)
    {
        depth_[i] = 0;
    }
}


template<class Distance>
Foam::face Foam::ImmersedCell<Distance>::createInternalFace() const
{
    // Declare internal face with mixed-up point ordering
    face unorderedInternalFace(points_.size());

    // Collect all points with zero distance to surface
    label nPif = 0;

    forAll (depth_, pointI)
    {
        if (mag(depth_[pointI]) < absTol_)
        {
            // Found point on zero plane
            unorderedInternalFace[nPif] = pointI;
            nPif++;
        }
    }

    unorderedInternalFace.setSize(nPif);

```

```
    // Sanity check: Do we have at least 3 points at zero distance?
    if (nPif < 3)
    {
         FatalErrorInFunction
             << "Less than 3 intersection points in cell on free surface." << nl
             << "depth: " << depth_
             << abort(FatalError);
    }

    // Order points, so that they form a polygon
    // Algorithm in analogy to geomCellLooper.C

    // Calculate centre
    point centre = average(unorderedInternalFace.points(points_));

    // Get base vectors of coordinate system normal
    // define plane that approximates the surface from 3 points

    // Line segment between points 0 and 1
    // Note: face orientation is unknown and needs to be adjusted
    // after the face has been created
    // HJ, 28/Nov/2017
    vector S0 =
        points_[unorderedInternalFace[1]]
      - points_[unorderedInternalFace[0]];

    S0 /= mag(S0) + SMALL;

    label pointID = -1;
    scalar minDotProd = 1 - SMALL;

    // Take best non-colinear value
    for (label pI = 2; pI < unorderedInternalFace.size(); pI++)
    {
        // Create second line segment
        vector S1 =
            points_[unorderedInternalFace[pI]]
          - points_[unorderedInternalFace[0]];

        S1 /= mag(S1) + SMALL;
        scalar curDotProd = mag(S0 & S1);

        if (curDotProd < minDotProd)
        {
            pointID = pI;
            minDotProd = curDotProd;
        }
    }

    if (pointID == -1)
    {
        // All intersection points are colinear
        FatalErrorInFunction
            << "Colinear points in cut"
            << abort(FatalError);
    }

    // Now create surface
    plane surface
    (
        points_[unorderedInternalFace[0]],
        points_[unorderedInternalFace[1]],
        points_[unorderedInternalFace[pointID]]
    );

    vector e0, e1;
    getBase(surface.normal(), e0, e1);

    // Get sorted angles from point on loop to centre of loop.
    SortableList<scalar> sortedAngles(unorderedInternalFace.size());
```

```
    forAll (sortedAngles, angleI)
    {
        vector toCentre(points_[unorderedInternalFace[angleI]] - centre);
        toCentre /= mag(toCentre);

        sortedAngles[angleI] = pseudoAngle(e0, e1, toCentre);
    }
    sortedAngles.sort();

    // Re-order points
    const labelList& indices = sortedAngles.indices();

    face orderedInternalFace(unorderedInternalFace.size());

    forAll (indices, i)
    {
        orderedInternalFace[i] = unorderedInternalFace[indices[i]];
    }

    // Check direction of the new face using average wet and dry point
    // HJ, 5/Dec/2017
    point wetPoint = vector::zero;
    label nWet = 0;

    point dryPoint = vector::zero;
    label nDry = 0;

    label nUndecided = 0;

    forAll (depth_, i)
    {
        if (depth_[i] > absTol_)
        {
            dryPoint += points_[i];
            nDry++;
        }
        else if  (depth_[i] < -absTol_)
        {
            wetPoint += points_[i];
            nWet++;
        }
        else
        {
            nUndecided++;
        }
    }

    if (nUndecided == depth_.size())
    {
         FatalErrorInFunction
             << "All points lay on the tri surface, zero volume cell?"
             << nl << "Points: " << points_
             << abort(FatalError);
    }

    wetPoint /= nWet;
    dryPoint /= nDry;

    // Good direction points out of the wet cell
    vector dir = dryPoint - wetPoint;
    dir /= mag(dir) + SMALL;

    vector n = orderedInternalFace.normal(points_);
    n /= mag(n);

    if ((dir & n) < 0)
    {
        orderedInternalFace = orderedInternalFace.reverseFace();
    }
```

```

    // Note: the face may have wrong orientation here.  It is corrected later
    // HJ, 5/Dec/2017
    return orderedInternalFace;
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

template<class Distance>
Foam::ImmersedCell<Distance>::ImmersedCell
(
    const label cellID,
    const polyMesh& mesh,
    const Distance& dist
)
:
    primitiveMesh
    (
        mesh.cells()[cellID].labels(mesh.faces()).size(), // nPoints
        0,                                // nInternalFaces (init to zero)
        mesh.cells()[cellID].size(),      // nFaces
        1                                 // nCells
    ),
    cellID_(cellID),
    mesh_(mesh),
    dist_(dist),
    absTol_(0),
    isAllWet_(false),
    isAllDry_(false),
    isBadCut_(false),
    // Initialize points_ with  points from cell
    points_(mesh_.cells()[cellID_].points(mesh_.faces(), mesh_.points())),
    faces_(),
    // We start with single cell with ID = 0, so it owns all faces
    faceOwner_(faces_.size(), 0),
    faceNeighbour_(),
    depth_(dist.distance(points_))
{
    const cell& origCell = mesh_.cells()[cellID];

    // Build a valid 1-cell mesh in local addressing

    // Create hash table that maps points on global mesh to local point list
    HashTable<label, label, Hash<label> > pointMapTable(points_.size());

    labelList origCellPointLabels = origCell.labels(mesh_.faces());

    forAll (points_, pointI)
    {
        // Insert globalID and localID
        pointMapTable.insert(origCellPointLabels[pointI], pointI);
    }

    // Make local face list by remapping the faces of the cell
    // Maximum number of new faces is twice the number of original faces
    // plus one internal face
    faces_ = faceList(origCell.size());

    forAll (origCell, faceI)
    {
        // Get old point list of faceI
        face origFace(mesh_.faces()[origCell[faceI]]);

        // Make sure that all faces point outward,
        // since they are going to be outside cells
        if (!(mesh_.faceOwner()[origCell[faceI]] == cellID))
        {
            // Cell is not owner of face, revert face orientation
            // for the use in a 1-cell mesh
```

```
            origFace = origFace.reverseFace();
        }

        // Make list to store new points
        labelList newLabels(origFace.size());

        // Map labels
        forAll (origFace, facePointI)
        {
            newLabels[facePointI] =
                pointMapTable.find(origFace[facePointI])();
        }

        // Create face from new point labels
        faces_[faceI] = face(newLabels);
    }

    // At this point, a 1-cell mesh is valid

    // Calculating absolute tolerances based on minimum edge length
    {
        // Use local edges
        const edgeList& cellEdges = edges();

        // Calculate min edge length for a quick check
        scalar minEdgeLength = GREAT;

        // Note: expensive calculation of min length.  HJ, 28/May/2015
        forAll (cellEdges, edgeI)
        {
            minEdgeLength =
                Foam::min(minEdgeLength, cellEdges[edgeI].mag(points_));
        }

        absTol_ = minEdgeLength*immersedPoly::tolerance_();
    }

    // Check if we have to perform cut at all
    if (max(depth_) < absTol_)
    {
        // All points of cell are below water surface
        isAllWet_ = true;

        return;
    }
    else if (min(depth_) > -absTol_)
    {
        // All points are above water surface
        isAllDry_ = true;

        return;
    }

#   ifdef WET_DEBUG
    Info << "Cell ID: " << cellID << "  BEFORE" << nl
        << "points: " << points_ << nl
        << "faces: " << faces_ << nl
        << "depth: " << depth_ << endl;
#   endif

    /*********************************************************************/
    // Starting to modify the 1-cell primitiveMesh.
    // Beyond this point be sure to know what points_, faces_, etc. contain,
    // before calling inherited primitiveMesh functions of this class.
    // Here be dragons!
    /*********************************************************************/

    // Created expanded point and face lists

    // Insert intersection points and adjust depth for intersections
```

```
    // This will add further points into the intersected face if needed
    // Depth at intersection will be zero.  HJ, 5/Dec/2017
    // Note that it is possible to have the cut face even if no new points
    // have been introduced.  HJ, 13/Mar/2019
    insertIntersectionPoints();

    // Update primitiveMesh parameters
    this->reset
    (
        points_.size(),         // nPoints
        0,                      // nInternalFaces
        faces_.size(),          // nFaces
        1                       // nCells
    );

#   ifdef WET_DEBUG
    Info << "Cell ID: " << cellID << "  ENRICHED" << nl
        << "points: " << points_ << nl
        << "faces: " << faces_ << nl
        << "depth: " << depth_ << endl;
#   endif
    // At this point, a 1-cell mesh with faces enriched for intersections
    // is valid.  HJ, 5/Dec/2017

    // Check if there has been a successful cut at all
    // For a good cut there should be at least 3 points at zero level
    label nIntersections = 0;

    // Added collinearity check.  HJ, 8/Apr/2022

    // Collect first intersection point as reference for colinearity check
    point refPoint;
    vector refVec;
    scalar minDot = GREAT;

    forAll (depth_, pointI)
    {
        if (mag(depth_[pointI]) < absTol_)
        {
            if (nIntersections == 0)
            {
                // First intersection: collect reference point
                refPoint = points_[pointI];
            }
            else if (nIntersections == 1)
            {
                // Second intersection: collect reference vector
                refVec = points_[pointI] - refPoint;

                // Normalise
                refVec /= mag(refVec) + SMALL;
            }
            else
            {
                // Third and further intersection: collinearity check
                vector otherVec = points_[pointI] - refPoint;

                // Normalise
                otherVec /= mag(otherVec) + SMALL;

                // Collect minimum dot-product
                minDot = Foam::min(minDot, (refVec & otherVec));
            }

            nIntersections++;
        }

        // Can the check be terminated early?
        if (nIntersections >= 3 && minDot < immersedPoly::collinearity_())
        {
```

```
                // Condition satisfied.  No need to keep checking
                break;
            }
        }

        // Check if the intersection is sufficient to make a proper face
        if
        (
            // Insufficient number of intersections
            nIntersections < 3
            // More than 3 intersections, but collinear
         || (nIntersections >= 3 && minDot > immersedPoly::collinearity_())
        )
        {
            // Check if cell centre is wet or dry, depending on greatest distance
            // away from the cutting surface
            // Note: cannot measure  distance geometrically because of
            // the unknown resolution of the immersed surface
            // HJ, 5/Dec/2017
            if (mag(min(depth_)) > mag(max(depth_)))
            {
                // All points of cell are below water surface
                isAllWet_ = true;

                return;
            }
            else
            {
                // All points are above water surface
                isAllDry_ = true;

                return;
            }
        }
        // From here on, there exists a valid intersection

        // Resize the face list.  Each face can be split into two, with one
        // extra internal face.  HJ, 5/Dec/2017

        // Make a copy of enriched faces, on which the cutting is performed
        faceList enrichedFaces = faces_;

        // Reset face lists, preserving existing faces
        faces_.setSize(2*faces_.size() + 1);
        faceOwner_.setSize(2*faces_.size() + 1);
        faceNeighbour_.setSize(1);

        // If we are not merely touching the water surface
        // with one point or edge, insert internal face that
        // connects all intersection points
        // create internal face, which gets inserted at front of faces_ list
        faces_[0] = createInternalFace();

        // Internal face points out of the wet cell.  Make the wet cell its owner
        faceOwner_[0] = WET;
        faceNeighbour_[0] = DRY;

        // Count new faces
        label nFaces = 1;

        // For all faces with inserted points, do face splitting
        forAll (enrichedFaces, oldFaceI)
        {
            const face& oldFace = mesh_.faces()[origCell[oldFaceI]];
            const face& newFace = enrichedFaces[oldFaceI];

            // Calculate old face area locally to avoid triggering polyMesh
            const scalar oldFaceArea =
                mesh_.faces()[origCell[oldFaceI]].mag(mesh_.points());

```

```
        // If a face has been modified, it will have extra points
        if (newFace.size() != oldFace.size())
        {
            // Make two faces: wet and dry
            // Wet face: wet points and intersection points
            face wetFace(newFace.size());
            label nWet = 0;

            // Dry face: dry points and intersection points
            face dryFace(newFace.size());
            label nDry = 0;

            forAll (newFace, pointI)
            {
                if (mag(depth_[newFace[pointI]]) < absTol_)
                {
                    // Intersection point. Add to both faces
                    wetFace[nWet] = newFace[pointI];
                    nWet++;

                    dryFace[nDry] = newFace[pointI];
                    nDry++;
                }
                else if (depth_[newFace[pointI]] < -absTol_)
                {
                    // Point is submerged, add to wetFace
                    wetFace[nWet] = newFace[pointI];
                    nWet++;
                }
                else // depth_[newFace[pointI]] > absTol_
                {
                    // Otherwise point must be dry, add to dryFace
                    dryFace[nDry] = newFace[pointI];
                    nDry++;
                }
            }

            // Check for a successful cut
            if (nWet >= 3)
            {
                // Insert wet face
                wetFace.setSize(nWet);
                faces_[nFaces] = wetFace;
                faceOwner_[nFaces] = WET;

                nFaces++;

                // Check for bad wet face cut
                if
                (
                    wetFace.mag(points_)
                  > (1 + immersedPoly::badCutFactor_())*oldFaceArea
                )
                {
                    // Wet face area is greater than original face area
                    // This is a bad cut
#                   ifdef WET_DEBUG
                    Pout<< "Bad cell face cut: wet = ("
                        << wetFace.mag(points_) << " "
                        << oldFaceArea
                        << ")" << endl;
#                   endif

                    isBadCut_ = true;
                }
            }

            if (nDry >= 3)
            {
                // Insert dry face
```

```
                dryFace.setSize(nDry);
                faces_[nFaces] = dryFace;
                faceOwner_[nFaces] = DRY;

                nFaces++;

                // Check for bad dry face cut
                if
                (
                    dryFace.mag(points_)
                  > (1 + immersedPoly::badCutFactor_())*oldFaceArea
                )
                {
                    // Dry face area is greater than original face area
                    // This is a bad cut
#                   ifdef WET_DEBUG
                    Pout<< "Bad cell face cut: dry = ("
                        << dryFace.mag(points_) << " "
                        << oldFaceArea
                        << ")" << endl;
#                   endif

                    isBadCut_ = true;
                }
            }
        }
        else
        {
            // Face cut has failed.  Insert original face and owner
            faces_[nFaces] = newFace;

            // Determine wet/dry based on distance to face centre
            // Note: cannot measure  distance geometrically because of
            // the unknown resolution of the immersed surface
            // HJ, 5/Dec/2017

            // Create face depth distance as a subset
            scalarField faceDepth(depth_, newFace);

            // Since the face has not been cut, all faceDepth should have the
            // same sign.  Otherwise, the face should straddle the immersed
            // surface. Check on minimum.
            // Note: this is a very precise check on purpose: there is no cut
            // and the face belongs either to a wet cell or a dry cell
            // HJ, 12/Mar/2019
            if (min(faceDepth) < scalar(0))
            {
                // Negative distance: wet face
                faceOwner_[nFaces] = WET;
            }
            else
            {
                // Positive distance: dry face
                faceOwner_[nFaces] = DRY;
            }

            nFaces++;
        }
    }

    faces_.setSize(nFaces);
    faceOwner_.setSize(nFaces);

    // Update primitiveMesh parameters
    this->reset
    (
        points_.size(),                   // nPoints
        faceNeighbour_.size(),            // nInternalFaces
        faces_.size(),                    // nFaces
        faceNeighbour_.size() + 1         // nCells
```

```
    );


#   ifdef WET_DEBUG
    this->checkMesh();
    Info << "Cell ID: " << cellID << "  AFTER" << nl
        << "points: " << points_ << nl
        << "faces: " << faces_ << nl
        << "depth: " << depth_ << endl;
#   endif

    const scalar oldCellVolume =
        mesh_.cells()[cellID_].mag(mesh_.points(), mesh_.faces());

    // Note: is it legal to cut a zero volume cell?  HJ, 11/Mar/2019

    scalar wetCut = cellVolumes()[WET]/oldCellVolume;

    scalar dryCut = cellVolumes()[DRY]/oldCellVolume;

    // Check for bad cell cut based on volume
    if
    (
        wetCut < -immersedPoly::badCutFactor_()
     || wetCut > (1 + immersedPoly::badCutFactor_())
     || dryCut < -immersedPoly::badCutFactor_()
     || dryCut > (1 + immersedPoly::badCutFactor_())
    )
    {
        isBadCut_ = true;
    }

    //  If the cut is not bad, adjust the cell for thin cell cut
    if (!isBadCut_)
    {
        if (mag(wetCut) < immersedPoly::liveFactor_())
        {
            // Cell is dry; reset
            isAllDry_ = true;
        }

        if (mag(dryCut) < immersedPoly::liveFactor_())
        {
            // Cell is wet; reset
            isAllWet_ = true;
        }
    }
    else
    {
#       ifdef WET_DEBUG
        Pout<< "Bad cell cut: volume = (" << wetCut << " " << dryCut
            << ") = " << wetCut + dryCut << nl
            // << "Points: " << nl << this->points() << nl
            // << "Faces: " << nl << this->faces() << nl
            // << "Owner: " << nl << this->faceOwner() << nl
            // << "Neighbour: " << nl << this->faceNeighbour() << nl
            // << "Cut (wet dry) = (" << isAllWet_ << " " << isAllDry_ << ")"
            << endl;
#       endif
    }

    // Correction on cutting is not allowed, as it results in an open cell
    // if faces are cut and the cell is not.
    // Previous check confirmed more than 3 valid cut points in the cell,
    // which means that some of the faces were cut.
    // Cutting tolerances for the cell and face have been adjusted to make sure
    // identical cut has been produced.
    // HJ, 11/Mar/2019
}
```

```

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

D

ImmersedFace.C

Listing D.1: ImmersedFace.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

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

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

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

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

#include "ImmersedFace.H"

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

template<class Distance>
void Foam::ImmersedFace<Distance>::createSubfaces
(
    const face& localFace,
    const scalarField& depth
)
{
    // Cut edges that cross the surface at the surface and add to
    // points and intersections

    // Make a copy of starting face points
    pointField localPoints(facePointsAndIntersections_);

    // Note: depth corresponds to local points

    // Expand the list for additional points. This leaves sufficient
    // space for intersection at every edge
    facePointsAndIntersections_.setSize(2*localPoints.size());
    scalarField newDepth(2*localPoints.size());

    // Get list of edges
    const edgeList edges = localFace.edges();

    // Count the number of newly created points, including original points
```

```
    label nNewPoints = 0;

    // For each point, determine if it is submerged( = -1), dry( = 1) or
    // on the surface ( = 0)
    // This is done during cutting to avoid using another tolerance check later
    // to dermine which points are on the surface, below or above it. By
    // definition, points that are a result of cutting are on the surface. (IG
    // 14/May/2019)
    labelList isSubmerged(facePointsAndIntersections_.size());

    // Loop through all edges
    forAll (edges, edgeI)
    {
        // Take reference to currentEdge
        const edge& curEdge = edges[edgeI];

        const label start = curEdge.start();
        const label end = curEdge.end();

        // Length of current edge
        const scalar edgeLength = curEdge.mag(localPoints);

        // Check if there is a legitimate cut to be found
        // Note: synced tolerances in ImmersedCell and ImmersedFace
        // HJ, 13/Mar/2019
        if
        (
            depth[start]*depth[end] < 0
         && edgeLength > SMALL
         && mag(depth[start]) > edgeLength*immersedPoly::tolerance_()
         && mag(depth[end]) > edgeLength*immersedPoly::tolerance_()
        )
        {
            // Prepare a new point to insert and determine its location
            point cutPoint;
            scalar depthAtCut = 0;

            if (!dist_.iterateDistance())
            {
                // Intersection is along the edge length (pf[end] - pf[start])
                // times the ratio of the depth at start and the difference
                // between depth at start and end; add to this the start point
                // and you have the location
                cutPoint =
                    localPoints[start]
                  + depth[start]/(depth[start] - depth[end])*
                    (localPoints[end] - localPoints[start]);
            }
            else
            {
                // Initialize bisection starting points
                point p0 = localPoints[start];
                point p1 = localPoints[end];

                // Depth at starting points
                scalar d0 = depth[start];
                scalar d1 = depth[end];

                // Initial guess of starting point same
                // as in non-iterative approach
                cutPoint = p0 + mag(d0)/(mag(d0) + mag(d1))*(p1 - p0);

                // Convergence criterion is the depth at newP
                depthAtCut = dist_.distance(cutPoint);

                // Initialize loop counter
                label iters = 0;

                while
                (
```

```
                (mag(depthAtCut) > immersedPoly::tolerance_())
             && (iters < immersedPoly::nIter_())
            )
            {
                // Is the guessed point on the same side of the surface
                // as p0? If yes, move p0 to the guessed point and thus
                // shorten the interval
                if (sign(depthAtCut) == sign(d0))
                {
                    d0 = depthAtCut;
                    p0 = cutPoint;
                }
                // otherwise, shorten the other side
                else
                {
                    d1 = depthAtCut;
                    p1 = cutPoint;
                }

                // determine new intersection point
                cutPoint =  p0 + mag(d0)/(mag(d0) + mag(d1))*(p1 - p0);

                // and calculate its depth
                depthAtCut = dist_.distance(cutPoint);

                iters++;
            }
        }

        // Store first point of edge
        facePointsAndIntersections_[nNewPoints] =
            localPoints[curEdge.start()];

        // Store first point depth
        newDepth[nNewPoints] = depth[curEdge.start()];

        // Determine whether it is above or below the surface.
        // NOTE: it must be one or the other since this is an original point
        // of the edge, and it passed the if statement above (IG
        // 14/May/2019)
        isSubmerged[nNewPoints] = sign(depth[curEdge.start()]);

        nNewPoints++;

        // Store the newly found cut point
        facePointsAndIntersections_[nNewPoints] = cutPoint;

        // Store newly found cut depth
        newDepth[nNewPoints] = depthAtCut;

        // The cut point is by definition on the surface and therefore
        // shared by the dry and wet face (IG 14/May/2019)
        isSubmerged[nNewPoints] = 0;

        nNewPoints++;
    }
    else
    {
        // No intersection: just copy first point of edge
        facePointsAndIntersections_[nNewPoints] =
            localPoints[curEdge.start()];

        // Store first point depth
        newDepth[nNewPoints] = depth[curEdge.start()];

        // Determine whether it is above, below or on the surface.
        // NOTE: now it can be any of the options since end or start is
        // sitting on the surface, otherwise the if statement above would
        // have been true.(IG 14/May/2019)
        // NOTE:
```

```
            // Old check depended on the length of the current edge, meaning
            // that the tolerance depends on the order the face is visited
            // (consider pair of faces on the processor boundary.
            // This is incorrect: use absolute tolerance instead, consistent
            // with the wet/dry test in the constructor
            // HJ, 10/May/2022
            if (mag(depth[curEdge.start()]) < absTol_)
            {
                isSubmerged[nNewPoints] = 0;
            }
            else
            {
                isSubmerged[nNewPoints] = sign(depth[curEdge.start()]);
            }

            nNewPoints++;
        }
    }

    // Point list should now be complete because last point of last edge should
    // be the starting point of the first edge
    facePointsAndIntersections_.setSize(nNewPoints);
    newDepth.setSize(nNewPoints);
    isSubmerged.setSize(nNewPoints);

    // Count the number of points on wet and dry parts of the face and create
    // the faces
    {
        // Face is intersected by surface

        // Initialise both faces to full size of intesection points
        // to be truncated after completion

        drySubface_.setSize(facePointsAndIntersections_.size());
        label nDry = 0;

        wetSubface_.setSize(facePointsAndIntersections_.size());
        label nWet = 0;

        forAll (facePointsAndIntersections_, pointI)
        {
            if (isSubmerged[pointI] == 1)
            {
                // Point is dry, add to dry sub-face
                drySubface_[nDry] = pointI;
                nDry++;
            }
            else if (isSubmerged[pointI] == -1)
            {
                // Point is submerged, add to wet sub-face
                wetSubface_[nWet] = pointI;
                nWet++;
            }
            else
            {
                // Point is on surface, add to both dry and wet sub-face
                drySubface_[nDry] = pointI;
                nDry++;

                wetSubface_[nWet] = pointI;
                nWet++;
            }
        }

        // Check if surface is merely touching the face
        // in that case, either dry or wet sub-face have less
        // than 3 points
        if (nDry < 3)
        {
            // The face is wet
```

```
                isAllWet_ = true;
                isAllDry_ = false;

                drySubface_.clear();
            }
            else
            {
                drySubface_.setSize(nDry);

                // Since cell cut is adjusted, face cut cannot be.
                // HJ, 5/Apr/2019
            }

            if (nWet < 3)
            {
                // The face is dry
                isAllWet_ = false;
                isAllDry_ = true;

                wetSubface_.clear();
            }
            else
            {
                wetSubface_.setSize(nWet);

                // Since cell cut is adjusted, face cut cannot be.
                // HJ, 5/Apr/2019
            }
        }
}


template<class Distance>
void Foam::ImmersedFace<Distance>::init()
{
    face localFace(facePointsAndIntersections_.size());

    // Local face addresses into local points
    forAll (localFace, pointI)
    {
        localFace[pointI]  = pointI;
    }

    // Distance from the surface for every point of face
    scalarField depth = dist_.distance(facePointsAndIntersections_);

    // Calculating absolute tolerances based on minimum edge length
    absTol_ = 0;

    {
        // Use local edges
        const edgeList edges = localFace.edges();

        // Calculate min edge length for a quick check
        scalar minEdgeLength = GREAT;

        // Note: expensive calculation of min length.  HJ, 28/May/2015
        forAll (edges, edgeI)
        {
            minEdgeLength =
                Foam::min
                (
                    minEdgeLength,
                    edges[edgeI].mag(facePointsAndIntersections_)
                );
        }

        absTol_ = minEdgeLength*immersedPoly::tolerance_();
    }

```

```
    // Check if all points are wet or dry, using absolute tolerance
    if (max(depth) < absTol_)
    {
        // All points are wet within a tolerance: face is wet
        isAllWet_ = true;
        isAllDry_ = false;

        wetSubface_ = localFace;
    }
    else if (min(depth) > -absTol_)
    {
        // All points are dry within a tolerance: face is dry
        isAllWet_ = false;
        isAllDry_ = true;

        drySubface_ = localFace;
    }
    else
    {
        // Face appears to be cut by the free surface.
        // Perform detailed analysis to create dry and wet sub-face
        createSubfaces(localFace, depth);
    }
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

template<class Distance>
Foam::ImmersedFace<Distance>::ImmersedFace
(
    const pointField& p,
    const Distance& dist
)
:
    dist_(dist),
    facePointsAndIntersections_(p),
    wetSubface_(),
    drySubface_(),
    isAllWet_(false),
    isAllDry_(false)
{
    init();
}


template<class Distance>
Foam::ImmersedFace<Distance>::ImmersedFace
(
    const label faceID,
    const polyMesh& mesh,
    const Distance& dist
)
:
    dist_(dist),
    facePointsAndIntersections_(mesh.faces()[faceID].points(mesh.points())),
    wetSubface_(),
    drySubface_(),
    isAllWet_(false),
    isAllDry_(false)
{
    // Initialised immersed face
    init();
}


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

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

AVANCEZ
1829

CHALMERS
UNIVERSITY OF TECHNOLOGY