Cite as: Moeller, K.: Investigating an alternative discretization of the gravitional force when simulating interfacial flows using the interIsoFoam solver. In Proceedings of CFD with OpenSource Software, 2021, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2021

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Investigating an alternative discretization of the gravitional force when simulating interfacial flows using the interIsoFoam solver

Developed for OpenFOAM-v1812
Requires: The TwoPhaseFlow library

Author:
Kasper Møller
Roskilde University
leenders@ruc.dk

Peer reviewed by:
Debarshee Ghosh
Johan Rønby
Mohammad H. A. Khanouki

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

January 18, 2022

Learning outcomes

The reader will learn:

How to use it:

- How to use the `interIsoFoam` solver.
- How to use the `setAlphaField` utility.

The theory of it:

- The theory of the isoAdvector algorithm and the reconstruction scheme, plicRDF, available to the `interIsoFoam` solver.
- The theory of the PIMPLE loop; the pressure-velocity coupling and the discretizations schemes used in the `interIsoFoam` solver.

How it is implemented:

- The implementation details of the advection of the interface using `isoAdvector`
- The implementaion of the plicRDF reconstruction scheme.
- How we may recognize the discretization schemes in the implementation

How to modify it:

- How to use the TwoPhaseFlow library
- How to add an extension to the library
- How to utilize the isoAdvector algorithm to ensure a hydrostatic balance between pressure and gravity

Prerequisites

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

- How to run standard document tutorials like a damBreak tutorial.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric
- How to customize a solver and do top-level application programming.
- Basic knowledge of C++ programming
- Basic knowledge of the Linux/Unix command line

Contents

Nomenclature

Acronyms

RDF	Reconstructed Distance Function

English symbols

$\hat{n}$	Interface unit normal
$\mathcal{H}$	$\mathcal{H}$ operator
A	Area
g	Gravitational acceleration
H	Heaviside function
n	Interface area normal
p	Total pressure
U	Cartesian velocity vector
V	Volume
w	Weight

Greek symbols

α	Volume fraction
δ	Dirac delta function
Γ	The interface between the fluids
Ω^+	Domain covered by the heavy fluid
Ω^-	Domain covered by the light fluid
ϕ	Volumetric flow rate
Ψ	Reconstructed distance function
ρ	Fluid density

Superscripts

$*$	Auxiliary velocity
$+$	Heavy fluid
$-$	Light fluid
n	Time step

Subscripts

C	Cell
d	Dynamic
f	Face
N	Neighbour
S	Surface

Chapter 1

Introduction

This project is concerned with the development of computational methods for multiphase flows, i.e., flows regarding two or more fluids co-existing in a common spatial domain. Examples could be the air and water phases in fire sprinklers or a wave hitting an offshore structure such as a wind turbine foundation. With the increase in computational power of modern computers, the interest in doing computations for these complex free surfaces has increased. However, the power of of modern computers are not enough. There is still a need for accurate and efficient computational methods and algorithms. Even though the governing equations of multiphase flows are well known and have been for a long time, the practical computations are still in a developing phase with several aspects suited for potential improvement.

In this project we consider an alternative numerical discretization of the gravitational force found in the governing equation for interfacial flows, or more specific, the `interIsoFoam` solver of OpenFOAM. The new method will utilize the interface advection algorithm, *isoAdvector*, developed by Roenby et al. [1].

A key challenge for multiphase flow is the discontinuities arising at the interface, e.g. the change in density which may be of the order 1:1000. Discontinuities are historically known to be difficult to handle for our numerical tools as the tools are developed with continuous functions in mind. With the development of the interfacial advection algorithm, isoAdvector, new opportunities arise but also new challenges. With isoAdvector, we may model and advect an interface accurately, efficiently and sharply. The sharpness of the interface will also be reflected in the discrete momentum equation. If this is not done properly, the discontinuity gives rise to spurious velocities which may cause a divergence from the true underlying solution as shown by Larsen et al. [2]. Here the authors consider the progression of nonlinear stream function waves from Fenton [3], where analytic solutions are known to the heavy phase.

If we use `interIsoFoam` for such simulations, we observe that the velocity behaves strange at the interface. This can be seen in Figure 1.1, where we visualize the volume fraction and the velocity magnitudes. These large velocities appear and disappear quickly hence they will not have an effect on the progression of the wave. However, they will put huge restrictions on the time step through the Courant number.

Regardless of the effect in the simulation, the spurious currents tells us that there is something fundamentally wrong in the current simulations using the `interIsoFoam` solver. In order to limit the scope, we will here anaylse the discretization errors found in the gravitational force. To isolate these errors, we will consider a hydrostatic test case. In this case we, for a consistent numerical scheme, should balance the pressure with the gravitational force hence it becomes a case where we may focus on the discretization of the pressure gradient and the gravitational force found in the momentum equation.

In this report, we will look at the current implementation of the `interIsoFoam` solver. We will dive into the theory behind it, show how to use the solver and how we may recognize the theory in the

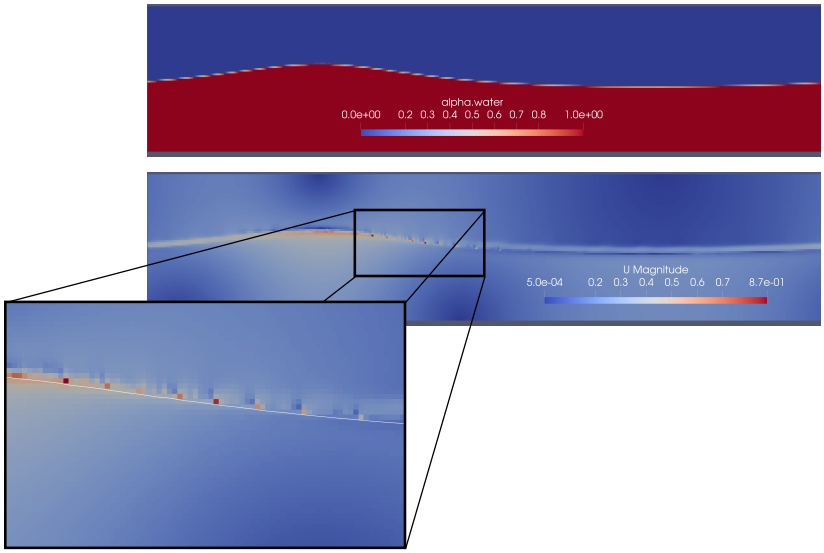


Figure 1.1: Simulation of a stream function wave using `interIsoFoam`. On top we have the volume fraction and below we have the velocity magnitude where the $\alpha = 0.5$ contour have been coloured in white. Notice how the spurious currents are present just above the interface.

implemented code. This will be followed by an analysis showing the hydrostatic issue of the solver. From here we will introduce a modification to the `interIsoFoam` solver through the *TwoPhaseFlow* library developed by Scheufler and Roenby [4]. This library offers a framework for faster and easier implementation of new models for interfacial flows. The modification, we will employ, will be through the `interFlow` solver of the TwoPhaseFlow library. This solver can be seen to be identical to the `interIsoFoam` solver. The modification, we employ to the TwoPhaseFlow library, is called `gravityRecon` and is developed by Henning Scheufler from German Aerospace Center (DLR). We will show how to add `gravityRecon` to the library and how to use the `interFlow` solver. Also, we will analyse the `gravityRecon` extension and show how it ensures a hydrostatic balance between the pressure and the gravitational force.

The performance of the `interIsoFoam` solver and the `interFlow` solver with `gravityRecon` will be shown on a simple hydrostatic test case. We will elaborate on how to set up the case using the two solvers and also highlight their similarities and differences.

Chapter 2

Theory

Let us start out the tour by looking into the theory of the interfacial flow solver, `interIsoFoam`. We will initially introduce the equations governing interfacial flows of two inviscid, incompressible fluids. We will then show how these equations will be solved and coupled in the `interIsoFoam` solver. The solver is shown to be composed of an advection step, using the isoAdvector algorithm, and a pressure-velocity coupling using the PIMPLE algorithm. In the advection step we will, with a focus on the plicRDF reconstruction scheme, elaborate on the explicit reconstruction of the interface as a planar interface. In the pressure-velocity coupling we will see how the pressure equation is formed using an auxiliary velocity field and in particular how we handle the gravitational force found in the governing equations.

2.1 Governing Equations

Let us first examine the governing equations for flows consisting of two immiscible, incompressible fluids. We will limit the scope to inviscid fluids and neglect the effect of surface tension. The governing equations will take a "one fluid" approach, see e.g. Tryggvason et al. [5], where the two fluids are modelled using one common density field with a sharp jump at the interface. The governing equations for the flow are then; the conservation of momentum

$$\frac{\partial(\rho\mathbf{U})}{\partial t} + \nabla\cdot(\rho\mathbf{U}\mathbf{U}) = -\nabla p + \rho\mathbf{g} \tag{2.1}$$

the incompressibility condition

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

and mass conservation

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{U}) = 0 \tag{2.3}$$

with $\mathbf{U}$ being the velocity field, p the pressure field, $\mathbf{g}$ the gravitational vector and ρ is the density field. The gravity vector will act in the negative z-direction hence we may write $\mathbf{g} = (0, 0, -g)^T$. The density field will be modelled using a Heaviside function

$$H(\mathbf{x}, t) = \begin{cases} 1 & ,\mathbf{x} \in \Omega^+ \\ 0 & ,\mathbf{x} \in \Omega^- \end{cases}$$

where Ω^+ and Ω^- denote the domains of the heavy and light fluid, respectively. We will occasionally use Γ to denote the shared interface. Note here that we have used the heavy fluid as reference fluid but the light fluid could be used likewise. With the above Heaviside function, we model the density field as

$$\rho(\mathbf{x}, t) = [\rho]H(\mathbf{x}, t) + \rho^- \tag{2.4}$$

where $[\rho] = \rho^+ - \rho^-$ denotes the jump in density with ρ^+ and ρ^- being the densities of the heavy and light fluid, respectively. Furthermore, as discussed by Rusche [6] and Popinet [7], we may improve the specification of boundary conditions by introducing a modified pressure, also named the dynamic pressure, as a dependent variable. The dynamic pressure is defined as

$$p_d = p - (\mathbf{g} \cdot \mathbf{x})\rho$$

thus the equation for conservation of momentum can be written as

$$\frac{\partial(\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U}\mathbf{U}) = -\nabla p_d - (\mathbf{g} \cdot \mathbf{x})\nabla\rho \tag{2.5}$$

Let us take a note on the gravitational force here as it is the main subject of this report. By using the formulation of the density field in Eq. (2.4), we may write the last term of Eq. (2.5) as

$$\begin{aligned}(\mathbf{g} \cdot \mathbf{x})\nabla\rho &= (\mathbf{g} \cdot \mathbf{x})\nabla\big([\rho]H + \rho^-\big) \\ &= [\rho](\mathbf{g} \cdot \mathbf{x})\nabla H \\ &= [\rho](\mathbf{g} \cdot \mathbf{x})\mathbf{n}_\Gamma \delta_\Gamma\end{aligned} \tag{2.6}$$

where $\mathbf{n}_\Gamma$ is the normal to the interface between the fluids and we use the Dirac delta function, δ_Γ which takes non-zero values on the interface only, (see e.g. the work of Popinet [7] for more derivation details). That is, by using the dynamic pressure as dependent variable, we change the gravitational force from a body force into a force acting on the interface only.

In the following sections we will elaborate on the `interIsoFoam` solver and its components. The main goal is to solve the above equations, (2.2), (2.3) and (2.5), simultaneously to have an expression of the velocity field, the density field and the (dynamic) pressure field for each point in space and time.

2.2 Numerical solution procedure

In OpenFOAM, hence also the `interIsoFoam` solver, we use the Finite Volume Method to discretize our governing equations. That is, we divide the computational domain into a finite number of volumes (or cells). In each cell we wish to satisfy the governing equations in the weak sense, i.e. in an integral form.

In order to solve the governing equations numerically, the `interIsoFoam` solver has, like many other solvers of OpenFOAM, adapted to the use of the PIMPLE algorithm. The PIMPLE algorithm may be seen as a combination of the SIMPLE and the PISO algorithms. The PIMPLE algorithm will consist of an outer loop (the SIMPLE part) and an inner loop (the PISO part). In the outer loop, we initially use conservation of mass, Eq. (2.3), to advect the density field or, as we will see, advect the volume fractions hence the interface. Then, as the fluids are assumed incompressible, we need to couple the pressure and the velocity. The coupling is done through an auxiliary velocity field based on the momentum equation, Eq. (2.5), where the effect from the pressure is neglected. With the auxiliary velocity field and the incompressibility condition, Eq. (2.2), we form a pressure equation from which we compute a pressure that ensures a divergence-free velocity field. The inner loop of the PIMPLE algorithm will then be composed of the construction and solution to the pressure equation together with the update of the velocity field. See the work of Moukalled et al. [8], Ferziger et al. [9] and Holzmann [10] for more information on the PIMPLE algorithm and its usage in OpenFOAM.

An overview of the `interIsoFoam` solver can be seen in Algorithm 1. Here we have included the time loop, the outer loop and the inner loop. However, for simplicity and with a focus on the relevant topics of this work, we have left out important aspects of the solver in this algorithm. This includes mesh motion, the non-orthogonal correction loop and turbulence computations. Also, we have assumed no momentum predictor step and no sub-cycles within each time step. In the following section, we elaborate on some of the steps of the algorithm and also go through the discretization

Algorithm 1 Overview of the `interIsoFoam` solver

```
1: Initialize fields: U, p_d and ρ
2: while Time loop do
3:     Compute Δt according to Courant number
4:     while Outer loop do
5:         Reconstruct the interface                    ▷ Compute interface center and normal
6:         Advect interface                             ▷ Update ρ
7:         while Inner loop do
8:             Build pressure equation                  ▷ With updated velocity
9:             Solve pressure equation                  ▷ Update p_d
10:            Update velocity using the pressure gradient  ▷ Update U
11:        end while
12:    end while
13: end while
```

and approximation procedures found in the `interIsoFoam` solver. For notational convenience, we introduce the notation of an average value in a cell

$$\langle \psi \rangle_C = \frac{1}{|V_C|} \int_C \psi \, \mathrm{d}V$$

with C denoting a computation cell, $|V_C|$ being the volume of the cell and ψ could be any field; scalar or vector.

2.2.1 Advection of the interface: isoAdvector

We will start by presenting the isoAdvector algorithm which will be the first step of the outer loop in the solver. The isoAdvector algorithm is a geometric volume-of-fluid (VoF) method. That is, we explicitly reconstruct the interface between the fluids (line 5 in Algorithm 1) and then advect the interface in a Lagrangian manner (line 6 in Algorithm 1) to have better estimates of the volume fluxes.

Let us start by considering an integral form of conservation of mass

$$\int_C \frac{\partial \rho}{\partial t} \, \mathrm{d}V + \int_C \nabla \cdot (\rho \mathbf{U}) \, \mathrm{d}V = 0$$

By using the expression of the density field in Eq. (2.4), we may write

$$\begin{aligned}
[\rho] \int_C \frac{\partial H}{\partial t} \, \mathrm{d}V + [\rho] \int_C \nabla \cdot (H\mathbf{U}) \, \mathrm{d}V + \rho^- \int_C \nabla \cdot \mathbf{U} \, \mathrm{d}V = 0 \\
[\rho] \int_C \frac{\partial H}{\partial t} \, \mathrm{d}V + [\rho] \int_C \nabla \cdot (H\mathbf{U}) \, \mathrm{d}V = 0 \\
\int_C \frac{\partial H}{\partial t} \, \mathrm{d}V + \int_C \nabla \cdot (H\mathbf{U}) \, \mathrm{d}V = 0
\end{aligned}$$

where we have utilized the incompressibility condition. With this, the equations for conservation of mass reduces to passive advection of the Heaviside function. We may reformulate the passive advection equation by considering Leibniz integral rule for the transient term and the divergence theorem for the advection term

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_C H \, \mathrm{d}V + \int_{\partial C} H\mathbf{U} \cdot \mathbf{n} \, \mathrm{d}A = 0$$

where ∂C denotes the surface of the cell and $\mathbf{n}$ is the normal pointing out of the cell. By restricting our selves to polygonal cells only, we can write the surface integral as a combination of faces

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_C H\,\mathrm{d}V + \sum_f \int_f H\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A = 0 \tag{2.7}$$

where f denotes a face of a polygonal cell. For clarity, we remark here that the above formulation leads to an advection scheme where only cells close to the interface needs attention. That is, in the bulk of the two fluids, the Heaviside function will not change between time steps, if we assume that the incompressibilty condition holds and there is no sources or sinks.

The above formulation, in Eq. (2.7), is the starting point of the isoAdvector algorithm. By introducing the volume fraction of the reference fluid (here the heavy fluid) in a cell,

$$\alpha_C(t) = \frac{1}{|V_C|}\int_C H\,\mathrm{d}V \tag{2.8}$$

and integrating forward in time (from time $t = t^n$ to time $t = t^{n+1}$), we get

$$\alpha_C(t^{n+1}) = \alpha_C(t^n) - \frac{1}{|V_C|}\sum_f \int_{t^n}^{t^{n+1}} \int_f H\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A\,\mathrm{d}\tau$$

The purpose of the isoAdvector algorithm is then to approximate the above double integral in a Lagrangian manner. That is, we will use an explicit reconstruction of the interface and then evaluate the double integral analytically. First, we will denote the double integral as

$$\Delta V_f = \int_{t^n}^{t^{n+1}} \int_f H\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A\,\mathrm{d}\tau \tag{2.9}$$

Then, let us define the volumetric face flux,

$$\phi_f(t) = \int_f \mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A$$

and approximate $\mathbf{U}\cdot\mathbf{n}$ as

$$\mathbf{U}\cdot\mathbf{n} \approx \frac{1}{|\mathbf{S}_f|}\int_f \mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A = \frac{\phi_f(t)}{|\mathbf{S}_f|}$$

which would be exact for constant $\mathbf{U}$ and $\mathbf{n}$. Here we have introduced $|\mathbf{S}_f|$ to denote the magnitude of the surface normal vector, i.e. the area of the face. With this approximation, we may write

$$\Delta V_f \approx \int_{t^n}^{t^{n+1}} \frac{\phi_f(\tau)}{|\mathbf{S}_f|}\int_f H\,\mathrm{d}A\,\mathrm{d}\tau$$

Then, as we only have information of the velocity, hence fluxes, at $t = t^n$, we will approximate the fluxes as constant through the time integration; i.e. we will set $\phi_f(t) = \phi_f(t^n)$ (denoted as ϕ_f^n) and write

$$\Delta V_f \approx \frac{\phi_f^n}{|\mathbf{S}_f|}\int_{t^n}^{t^{n+1}} \int_f H\,\mathrm{d}A\,\mathrm{d}\tau \tag{2.10}$$

In order to evaluate the integrals in Eq. (2.10), which gives the amount of the heavy fluid transported over the face, we need an explicit reconstruction of the interface. This responds to line 5 of Algorithm 1 and will be the topic of the following section.

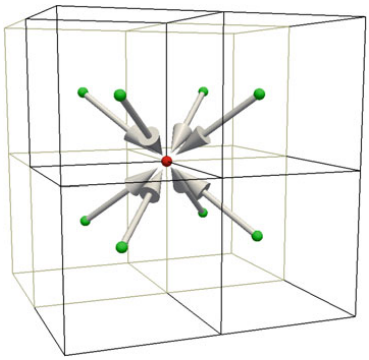
Reconstruction of the interface

The reconstruction will only happen in cells that contain an interface. That is, we should distinguish between interface cells and non-interface cells. Interface cells are cells occupied by both the heavy and light fluid thus we may define interface cells as cells satisfying

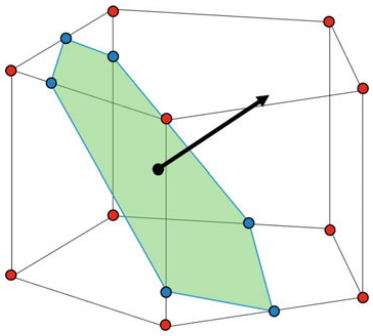
$$\epsilon < \alpha_C < 1 - \epsilon \tag{2.11}$$

where $\epsilon \ll 1$ is a user specified tolerance.

The interfaces, in the interface cells, will be estimated using a planar interface. That is, they will be described by an interface center, $\mathbf{x}_S$ and an interface area normal, $\mathbf{n}_S$ with length equal to the area of the interface in the cell. The question is now how we compute these two vectors. The `interIsoFoam` solver offers three options; the *isoAlpha* reconstruction method, the *gradAlpha* reconstruction method and the *plicRDF* reconstruction method. The isoAlpha method is the original reconstruction method for the isoAdvector algorithm described by Roenby et al. [1]. Here, for each interface cell, we interpolate the volume fraction data to the *vertices* of the cell using the volume fractions of the neighbouring cells. From the information on the vertices, we determine cut points on the edges. These cut points will be set in correspondence to the volume fraction of the given cell and will be used to construct the interface center and interface area normal. See Figure 2.1 for a visualization of the steps. The gradAlpha method will approximate the interface area normal using



(a) Interpolation of volume fractions to vertices using neighbouring cell values



(b) Construction of planar interface using cut points (blue points). The interface center (black point) and the interface area normal (black arrow) are also shown

Figure 2.1: Visualization of isoSurface reconstruction steps using the iso-Alpha method; Figures are from Roenby et al. [11]

the volume fractions. That is, we will reconstruct the interface unit normal, $\hat{\mathbf{n}}_S$ as

$$\hat{\mathbf{n}}_S = \frac{\nabla\alpha}{|\nabla\alpha|} \tag{2.12}$$

using a least square gradient. The interface area normal will then be the interface unit normal times the area of the interface in the cell. The interface center will then be set, along the direction of the interface normal, in correspondence to the volume fractions.

Scheuffler and Roenby [12] describes the plicRDF method and also compared it to the isoAlpha method. Here it is demonstrated that the plicRDF method shows improved convergence properties compared to the isoAlpha method; especially for the computation of the interface normals. In the following we will dive into the theory of the plicRDF scheme used by isoAdvector.

The plicRDF method uses an iterative procedure to calculate the interface area normals and interface centers. To run this, we need an initial estimate of the interface unit normal. This can be retrieved using either the interface normals from the previous time step or using the discrete gradient of the volume fractions in a cell as for the gradAlpha method. Using the initial normals, we will compute the interface location of the next iteration, $\mathbf{x}_S^{\text{new}}$, as the point, along the interface normal, that yields the correct volume fraction. See the left of Figure 2.2, where the new interface center (the blue dot) is placed along the interface normal, to ensure that the planar interface cuts the cell acording to the volume fraction, called α_i^{new} in the figure. With this new interface position we will create a Reconstructed Distance Function (RDF), Ψ, which, for each cell close to the interface, estimates the (signed) distance from the cell centers to the interface. How this value is computed will follow shortly. First let us remark that with a value of the RDF in the cells close to the interface, the interface unit normal will be computed as

$$\hat{\mathbf{n}}_S^{\text{new}} = \frac{\nabla\Psi}{|\nabla\Psi|} \tag{2.13}$$

where we use a least square gradient. Using this new interface normal, the procedure repeats until the change in interface normal is below a specified tolerance. A sketch of the computations for the new interface center, $\mathbf{x}_S^{\text{new}}$, and interface unit normal, $\hat{\mathbf{n}}_S^{\text{new}}$, can be seen in Figure 2.2. To the left, we, as mentioned earlier, find the interface center, that cuts the cell according to the sought volume fraction. To the right, we illustrate that with the new interface center (the blue dot), we compute the RDF and update the interface unit normal using Eq. (2.13) (the blue vector).

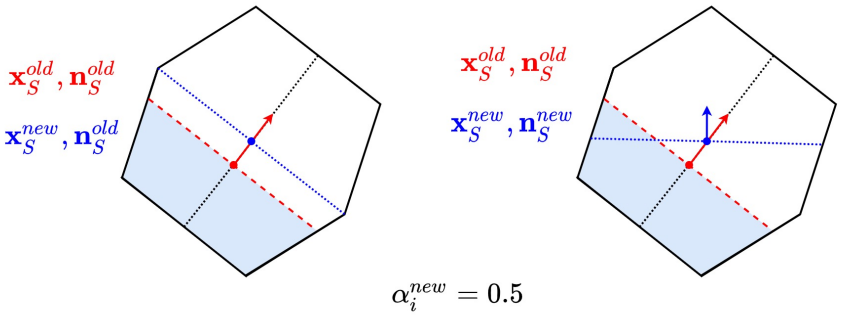


Figure 2.2: Difference between *old* and *new* interface location and interface normal: (Left) Computation of new interface center (blue dot) along old interface normal (red vector) in correspondence to new volume fraction, α_i^{new} (Right) Computation of new interface normal (blue vector) using the RDF computed from new interface centers.

Now, let us return to the RDF value in the cells. It will be computed as the distance from the cell center to the interface. The distance is computed differently for interface cells and non-interface cells. Non-interface cells may have several point neighbours[1] that contain an interface. By this, the distance to the interface, for a non-interface cell, will be computed using a weighted sum of the neighbouring interface cells

$$\Psi_i = \frac{\sum_j w_{ij}\tilde{\Psi}_{ij}}{\sum_j w_{ij}} \tag{2.14}$$

where $\tilde{\Psi}_{ij}$ denotes the normal distance between the center of cell i and the interface center of cell j

$$\tilde{\Psi}_{ij} = \hat{\mathbf{n}}_{S,j} \cdot (\mathbf{x}_i - \mathbf{x}_{S,j}) \tag{2.15}$$

and w_{ij} is a weighting factor defined as

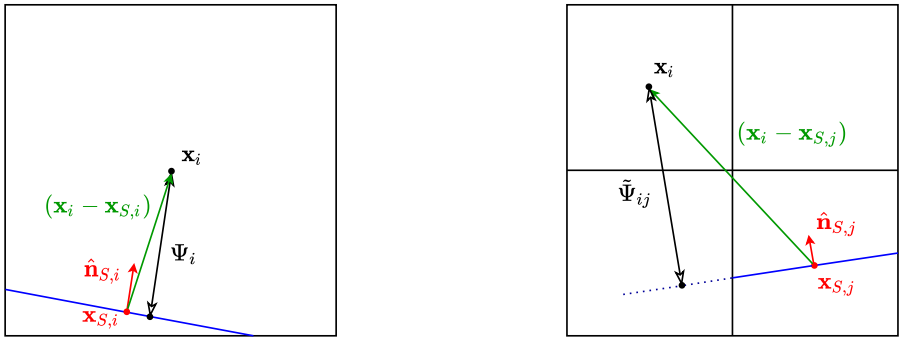
$$w_{ij} = \frac{|\hat{\mathbf{n}}_{S,j} \cdot (\mathbf{x}_i - \mathbf{x}_{S,j})|^2}{|\mathbf{x}_i - \mathbf{x}_{S,j}|^2} \tag{2.16}$$

[1] Cells with which it shares a vertex

Here $\mathbf{x}_i$ denotes the center of cell i. For interface cells, the distance will be measured using the interface center and interface normal found in the cell. The RDF value will here be

$$\Psi_i = \hat{\mathbf{n}}_{S,i} \cdot (\mathbf{x}_i - \mathbf{x}_{S,i})$$

A visualization of the measured distances, for both interface cells and non-interface cell, can be seen in Figure 2.3.



(a) Interface cell (b) Non-interface cell

Figure 2.3: Illustration of how to compute the distance to the interface using cell center, $\mathbf{x}_i$ interface center, $\mathbf{x}_{S,j}$ and interface unit normal, $\hat{\mathbf{n}}_{S,j}$

Finalize the advection of the interface

With the reconstruction of the interface, we may now evaluate the double integral in Eq. (2.9) thus advect the interface. Before doing so, let us remark here that the inner integral will be the area of the face that are occupied with the reference fluid and with the use of polygonal cells, this area, and its evolution within the entire time step, may be computed analytically. We will not go into these details here, but see the work of Roenby et al. [1] for an explanation on how this is done.

This leaves us with the evaluation of the double integral, ΔV_f, and we may now update the volume fraction as (line 6 of Algorithm 1)

$$\alpha_C(t^{n+1}) = \alpha_C(t^n) - \frac{1}{|V_C|} \sum_f \Delta V_f \tag{2.17}$$

where the only source of errors will be the estimation of the interface as planar within each cell and the use of the velocity face fluxes of the old time step, ϕ_f^n, which is integrated over the whole face and not just the heavy fluid.

2.2.2 Pressure-velocity coupling

With the advection of the interface completed, the following section introduce how we combine the pressure field with the velocity field in the `interIsoFoam` solver. This will correspond to line 7-11 of Algorithm 1. The solver uses the PIMPLE algorithm where we derive an equation for the pressure using the incompressibility condition. This pressure will then be used to correct an auxiliary velocity field thus making it incompressible. The pressure-velocity coupling in `interIsoFoam` is identical to the one found in `interFoam`. A detailed description of the pressure-velocity coupling in the `interFoam` solver can be found in the work of Deshpande et al. [13].

By spatially integrating the momentum equation over a cell, we may write

$$\int_C \frac{\partial(\rho \mathbf{U})}{\partial t}\,\mathrm{d}V + \int_C \nabla\cdot(\rho\mathbf{U}\mathbf{U})\,\mathrm{d}V = -\int_C \nabla p_d\,\mathrm{d}V - \int_C (\mathbf{g}\cdot\mathbf{x})\nabla\rho\,\mathrm{d}V$$

and similarly for the incompressibilty condition

$$\int_C \nabla\cdot\mathbf{U}\,\mathrm{d}V = 0$$

We will start by deriving an expression for an auxiliary velocity, derived from the momentum equation, followed by the pressure equation. As we only consider inviscid fluids, the auxiliary velocity equation will consist of the transient term, the convection term and the gravitational force. The discretization of these terms will be given, yielding the expression of the discrete auxiliary velocity field.

For the time derivative we utilize Leibniz integral rule

$$\int_C \frac{\partial(\rho\mathbf{U})}{\partial t}\,\mathrm{d}V = \frac{\mathrm{d}}{\mathrm{d}t}\int_C \rho\mathbf{U}\,\mathrm{d}V$$

We could then consider an Euler time integration scheme thus we would approximate the time derivative as

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_C \rho\mathbf{U}\,\mathrm{d}V \approx \frac{\langle\rho\mathbf{U}\rangle_C^{n+1} - \langle\rho\mathbf{U}\rangle_C^{n}}{\Delta t}|V_C|$$

where superscripts indicate the discrete time steps and Δt is the chosen time step size. Last, in order to get an expression for the velocity in the next time step, we do the approximation

$$\langle\rho\mathbf{U}\rangle_C^{n+1} \approx \langle\rho\rangle_C^{n+1}\langle\mathbf{U}\rangle_C^{n+1}$$

For the convective term, we will deviate from the usual discretization of the convective term. We utilize that we, from the advection step, have information about how the interface, hence the density, changes within a time step. The following description is base on the work of Roenby et al. [11].

Consider the following

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_C \rho\mathbf{U}\,\mathrm{d}V + \int_C \nabla\cdot(\rho\mathbf{U}\mathbf{U})\,\mathrm{d}V = ...$$

i.e. the left hand side of the integrated momentum equation. Now, let us integrate forward in time

$$\Big(\langle\rho\mathbf{U}\rangle_C^{n+1} - \langle\rho\mathbf{U}\rangle_C^{n}\Big)|V_C| + \int_{t^n}^{t^{n+1}}\int_C \nabla\cdot(\rho\mathbf{U}\mathbf{U})\,\mathrm{d}V\,\mathrm{d}\tau = ...$$

and focus on the convective term. Using the divergence theorem and assuming a use of polygonal cells yield

$$\begin{aligned}\int_{t^n}^{t^{n+1}}\int_C \nabla\cdot(\rho\mathbf{U}\mathbf{U})\,\mathrm{d}V\,\mathrm{d}\tau &= \int_{t^n}^{t^{n+1}}\int_{\partial C} \rho\mathbf{U}\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A\,\mathrm{d}\tau \\ &= \sum_f \int_{t^n}^{t^{n+1}}\int_f \rho\mathbf{U}\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A\,\mathrm{d}\tau\end{aligned}$$

To linearize the term, let us, as we did in the advection step, approximate $\mathbf{U}\cdot\mathbf{n}$ as

$$\mathbf{U}\cdot\mathbf{n} \approx \frac{1}{|\mathbf{S}_f|}\int_f \mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A = \frac{\phi_f(t)}{|\mathbf{S}_f|}$$

If we then assume that the velocity is constant at each face, call if $\mathbf{U}_f$, and evaluate the face flux, $\phi_f(t)$ at $t = t^n$, we may then write

$$\sum_f \int_{t^n}^{t^{n+1}}\int_f \rho\mathbf{U}\mathbf{U}\cdot\mathbf{n}\,\mathrm{d}A\,\mathrm{d}\tau \approx \sum_f \mathbf{U}_f\frac{\phi_f^n}{|\mathbf{S}_f|}\int_{t^n}^{t^{n+1}}\int_f \rho\,\mathrm{d}A\,\mathrm{d}\tau$$

Before we move into how we will evaluate $\mathbf{U}_f$, let us continue with the double integral. From the expression of the density field in Eq. (2.4), we may write

$$\sum_f \mathbf{U}_f \frac{\phi_f^n}{|\mathbf{S}_f|} \int_{t^n}^{t^{n+1}} \int_f \rho \, \mathrm{d}A \, \mathrm{d}\tau = \sum_f \mathbf{U}_f \frac{\phi_f^n}{|\mathbf{S}_f|} \int_{t^n}^{t^{n+1}} \int_f [\rho] H + \rho^- \, \mathrm{d}A \, \mathrm{d}\tau$$

by splitting the face integral, we get

$$= \sum_f \mathbf{U}_f \frac{\phi_f^n}{|\mathbf{S}_f|} \int_{t^n}^{t^{n+1}} [\rho] \int_f H \, \mathrm{d}A + \rho^- \int_f \mathrm{d}A \, \mathrm{d}\tau$$

and by splitting the time integral

$$= \sum_f \mathbf{U}_f \frac{\phi_f^n}{|\mathbf{S}_f|} \left([\rho] \int_{t^n}^{t^{n+1}} \int_f H \, \mathrm{d}A + \rho^- \int_{t^n}^{t^{n+1}} \int_f \mathrm{d}A \, \mathrm{d}\tau \right)$$

Now, using the expression of ΔV_f from Eq. (2.10), we may write

$$= \sum_f \mathbf{U}_f \left([\rho] \Delta V_f + \rho^- \phi_f^n \Delta t \right) \tag{2.18}$$

To finish the discussion of the convective term, we need to get the face value of the convected velocity, $\mathbf{U}_f$. This is obtained by an implicit interpolation between neighbouring cell values.

$$\mathbf{U}_f = \langle \mathbf{U} \rangle_C^{n+1} w_C + \langle \mathbf{U} \rangle_N^m w_N \tag{2.19}$$

where m denotes the latest update of the velocity field (from the inner loop) and subscripts C and N denote the current cell and its neighbour cell, i.e. the two cells adjacent to the face. Also, w_C and w_N denote the corresponding interpolation weights. We note here that this notation is crude as the interpolation weights contain spatial factors, limiters etc. However, the details will not be investigated further here. The main takeaway is the presence of an implicit velocity. See the work of Deshpande et al. [13] for a more thorough explanation.

Next, we will form the auxiliary velocity field although we have not covered the discretization of the dynamic pressure gradient and the gravitational force. Both terms have a similar discretization hence will be covered together when we get to the dynamic pressure equation.

The auxiliary velocity

In order to get a representation of the auxiliary velocity field, we initially combine the discretization of the transient term and the convection term and then, after some manipulations of these terms, we add the contribution of the gravitational force. Consider the left hand side of the discretized momentum equation

$$\frac{\langle \rho \rangle_C^{n+1} \langle \mathbf{U}^* \rangle_C - \langle \rho \mathbf{U} \rangle_C^n}{\Delta t} |V_C| + \sum_f \mathbf{U}_f \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) = ...$$

Note here that we have used the superscript, *, to denote the auxiliary velocity; not the velocity for the next time step. Furthermore, the terms in the parenthesis of the second term above deviates slightly from the one obtained in Eq. (2.18). This is due to the analytic time integration in Eq. (2.18) and an Euler time integration in the equation above. Then we use the expression of the velocity on the face in Eq. (2.19), we get

$$\frac{\langle \rho \rangle_C^{n+1} \langle \mathbf{U}^* \rangle_C - \langle \rho \mathbf{U} \rangle_C^n}{\Delta t} |V_C| + \sum_f \left(\langle \mathbf{U} \rangle_C^* w_C + \langle \mathbf{U} \rangle_N^m w_N \right) \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) = ...$$

followed by moving the explicit terms to the right hand side

$$\langle \mathbf{U}^* \rangle_C \left[\frac{\langle \rho \rangle_C^{n+1}}{\Delta t} |V_C| + \sum_f w_C \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) \right] = - \sum_f \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) w_N \langle \mathbf{U} \rangle_N^n + \frac{\langle \rho \mathbf{U} \rangle_C^m}{\Delta t} |V_C| + ...$$

and introducing a more compact notation

$$a_C \langle \mathbf{U}^* \rangle_C = \sum_f a_N \langle \mathbf{U} \rangle_N^m + e_C + ... \tag{2.20}$$

where

$$\begin{aligned} a_C &= \frac{\langle \rho \rangle_C^{n+1}}{\Delta t} |V_C| + \sum_f w_C \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) \\ a_N &= - \left([\rho] \frac{\Delta V_f}{\Delta t} + \rho^- \phi_f^n \right) w_N \\ e_C &= \frac{\langle \rho \mathbf{U} \rangle_C^n}{\Delta t} |V_C| \end{aligned}$$

For convenience we introduce the $\mathcal{H}$ operator

$$\mathcal{H} \left(\langle \mathbf{U} \rangle_N^m \right) = \sum_f a_N \langle \mathbf{U} \rangle_N^m + e_C \tag{2.21}$$

hence we may write Eq. (2.20) as

$$a_C \langle \mathbf{U}^* \rangle_C = \mathcal{H} \left(\langle \mathbf{U} \rangle_N^m \right) + ...$$

Then by adding the gravitational force, the auxiliary velocity field may be written as

$$\langle \mathbf{U}^* \rangle_C = \frac{\mathcal{H} \left(\langle \mathbf{U} \rangle_N^m \right)}{a_C} - \frac{\langle (\mathbf{g} \cdot \mathbf{x}) \nabla \rho \rangle_C^{n+1}}{a_C}$$

i.e. without the effect of the pressure gradient. Note that the gravitational force takes value at the new time step, as we assume that an advection of the interface already has been made. Furthermore, we note that the gravitational force has not been discretized yet. The discretization will be elaborated in the following section.

The pressure correction

With a representation of the auxiliary velocity, we can move towards computing the dynamic pressure thus also computing a divergence-free velocity field. This is done in the inner loop of the algorithm. Here we construct and solve the pressure equation (line 8 and 9 of Algorithm 1) and update the velocity field (line 10 of Algorithm 1).

Let us start by writing the incompressibility condition in integral form

$$\int_C \nabla \cdot \mathbf{U} \, \mathrm{d}V = 0$$

In order to have a divergence free velocity field, it should then satisfy

$$\begin{aligned} \int_C \nabla \cdot \mathbf{U}^{m+1} \, \mathrm{d}V &= \sum_f \int_f \mathbf{U}^{m+1} \cdot \mathbf{n} \, \mathrm{d}A \\ &= \sum_f \phi_f^{m+1} = 0 \end{aligned}$$

where m highlights the update of the velocity is done in the inner loop and we, again, have used the divergence theorem and assumed polygonal cells. The question is now, how to represent the face fluxes.

We start by using a mean value approximation of the face integral

$$\phi_f^{m+1} = \int_f \mathbf{U}^{m+1} \cdot \mathbf{n} \, \mathrm{d}A \approx \mathbf{U}_f^{m+1} \cdot \mathbf{S}_f$$

where $\mathbf{S}_f$, as earlier, is the surface area vector pointing in the normal direction of the face and with a magnitude equal the face area. In order to find an expression for the mean value on the face, $\mathbf{U}_f^{m+1}$, we will consider the cell valued velocity field of the next time step. We will represent it as a combination of the auxiliary velocity and the pressure gradient

$$\begin{aligned}\langle \mathbf{U} \rangle_C^{m+1} &= \langle \mathbf{U}^* \rangle_C - \frac{\langle \nabla p_d \rangle_C^{n+1}}{a_C} \\ &= \frac{\mathcal{H}\left(\langle \mathbf{U} \rangle_N^m\right)}{a_C} - \frac{\langle (\mathbf{g} \cdot \mathbf{x}) \nabla \rho \rangle_C^{n+1}}{a_C} - \frac{\langle \nabla p_d \rangle_C^{n+1}}{a_C}\end{aligned}$$

such that the corresponding face value can be found as

$$\mathbf{U}_f^{m+1} = \left(\frac{\mathcal{H}\left(\langle \mathbf{U} \rangle_N^m\right)}{a_C}\right)_f - \left(\frac{1}{a_C}\right)_f \left((\mathbf{g} \cdot \mathbf{x}) \nabla \rho\right)_f^{n+1} - \left(\frac{1}{a_C}\right)_f \left(\nabla p_d\right)_f^{n+1}$$

and the face flux becomes

$$\begin{aligned}\phi_f^{m+1} &\approx \mathbf{U}_f^{m+1} \cdot \mathbf{S}_f \\ &= \left(\frac{\mathcal{H}\left(\langle \mathbf{U} \rangle_N^m\right)}{a_C}\right)_f \cdot \mathbf{S}_f - \left(\frac{1}{a_C}\right)_f \left((\mathbf{g} \cdot \mathbf{x}) \nabla \rho \cdot \mathbf{S}_f\right)_f^{n+1} - \left(\frac{1}{a_C}\right)_f \left(\nabla p_d \cdot \mathbf{S}_f\right)_f^{n+1}\end{aligned} \tag{2.22}$$

where $\left(\frac{\mathcal{H}(\langle \mathbf{U} \rangle_N^m)}{a_C}\right)_f$ and $\left(\frac{1}{a_C}\right)_f$ are found using interpolation of neighbouring cells, e.g. linear interpolation. The discretization of the gravitational force and the pressure gradient, on the faces, will be discussed shortly. First, let us note that the incompressibility condition now becomes

$$\sum_f \phi_f^{m+1} = 0$$

$$\Leftrightarrow$$

$$\sum_f \left(\frac{1}{a_C}\right)_f \left(\nabla p_d \cdot \mathbf{S}_f\right)_f^{n+1} = \sum_f \left(\frac{\mathcal{H}\left(\langle \mathbf{U} \rangle_N^m\right)}{a_C}\right)_f \cdot \mathbf{S}_f - \left(\frac{1}{a_C}\right)_f \left((\mathbf{g} \cdot \mathbf{x}) \nabla \rho \cdot \mathbf{S}_f\right)_f^{n+1}$$

i.e. an equation yielding a dynamic pressure that ensures a velocity field satisfying the incompressibility constraint. Remark here that the $\mathcal{H}$ operator gets updated using the updated velocity field each time we construct the pressure equation (line 8 of Algorithm 1).

In order to solve the pressure equation, we need to discretize the pressure gradient and the gravitational force at the faces. The pressure gradient will be discretized using linear approximation

$$\left(\nabla p_d \cdot \mathbf{S}_f\right)_f^{n+1} \approx \frac{(p_d)_N^{n+1} - (p_d)_C^{n+1}}{|\mathbf{d}_{CN}|} |\mathbf{S}_f|$$

where $(p_d)_C$ and $(p_d)_N$ denotes cell values of the pressure and $\mathbf{d}_{CN}$ denotes the distance vector between the corresponding cell centers. The gravitational force will be discretized in a similar manner. First, we use a mean value approximation for the inner product

$$\left((\mathbf{g} \cdot \mathbf{x}) \nabla \rho \cdot \mathbf{S}_f\right)_f^{n+1} \approx (\mathbf{g} \cdot \mathbf{x}_f)\left(\nabla \rho \cdot \mathbf{S}_f\right)_f^{n+1}$$

where $\mathbf{x}_f$ denotes the position of the face center. The discretization of the face gradient will be similar as before hence

$$(\mathbf{g}\cdot\mathbf{x}_f)\big(\nabla\rho\cdot\mathbf{S}_f\big)_f^{n+1} \approx (\mathbf{g}\cdot\mathbf{x}_f)\frac{\rho_N^{n+1}-\rho_C^{n+1}}{|\mathbf{d}_{CN}|}|\mathbf{S}_f| \tag{2.23}$$

From this, the discrete pressure equation reads

$$\begin{aligned}\sum_f\left(\frac{1}{a_C}\right)_f\left(\frac{(p_d)_N^{n+1}-(p_d)_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| = \\ \sum_f\left(\frac{\mathcal{H}\left(\langle\mathbf{U}\rangle_N^m\right)}{a_C}\right)_f\cdot\mathbf{S}_f-\left(\frac{1}{a_C}\right)_f\left((\mathbf{g}\cdot\mathbf{x}_f)\frac{\rho_N^{n+1}-\rho_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f|\end{aligned} \tag{2.24}$$

After computing the discrete pressure field, we update the discrete velocity field as

$$\langle\mathbf{U}^{m+1}\rangle_C = \frac{\mathcal{H}(\mathbf{U}^m)}{a_C}+\frac{1}{a_C}\mathcal{R}\left(\zeta_f^{n+1}\right) \tag{2.25}$$

where ζ_f^{n+1} denotes the face fluxes consisting of the discretized gravitational force and the discretized pressure gradient, i.e.

$$\zeta_f^{n+1} = -\big((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f\big)_f^{n+1}-\big(\nabla p_d\cdot\mathbf{S}_f\big)_f^{n+1}$$

and $\mathcal{R}$ is a linear reconstruction operator that reconstructs cell values from face fluxes, and is defined as

$$\langle\psi\rangle_C \approx \mathcal{R}(\psi_f) \equiv \left(\sum_f\frac{\mathbf{S}_f\mathbf{S}_f^T}{|\mathbf{S}_f|}\right)^{-1}\left(\sum_f\frac{\mathbf{S}_f}{|\mathbf{S}_f|}\psi_f\right) \tag{2.26}$$

with ψ_f denoting all the fluxes at the faces of the cell and $\mathbf{S}_f\mathbf{S}_f^T$ being the outer product of the surface area vectors. Consult the work of Aguerra et al. [14] for a nice motivation and discussion of the reconstruction operator found in OpenFOAM.

With this, Eq. (2.25) ends the inner loop (line 10 of Algorithm 1). If several outer loops are used, we would repeat the advection of the interface using the updated velocity field and repeat the inner loop with a new pressure equation. When the outer loop then terminates, we are finished with one time iteration and ready to move on in the time loop.

Chapter 3

Implementation details

Let us consider how we may recognize the theory in the implementation of the `interIsoFoam` solver. We will have a focus on how we create the object holding the gravity vector and how we compute the inner product between the gravity vector and the face centers. We will look into the isoAdvector algorithm with a focus on the reconstruction of the interface using the reconstruction scheme, `plicRDF`. This will be followed by the pressure-velocity coupling (the PIMPLE loop) of the `interIsoFoam` solver. Here we will state where and how the inner product between the gravity vector and the face centers is used.

Note here that the chapter includes several code snippets. These will always include a title redirecting the reader to the actual file. Depending on the uniqueness of the file name in the OpenFOAM code library, the actual title may include the full path or only the name of the file.

3.1 The `interIsoFoam` solver

Assuming we have sourced the OpenFOAM environmental variables, the source code of the `interIsoFoam` solver can be found at

`$FOAM_SOLVERS/multiphase/interIsoFoam/interIsoFoam.C`

This chapter will take this particular file as a starting point. From here we will move into various classes and functions that will help us describe the implementation details of the theory from the previous chapter.

Going into the file, `interIsoFoam.C`, the first thing we notice, is the inclusion of a lot header files. These are all included prior to the `main` function of the file.

`interIsoFoam.C`

```
54 #include "fvCFD.H"
55 #include "dynamicFvMesh.H"
56 #include "isoAdvection.H"
57 #include "EulerDdtScheme.H"
58 #include "localEulerDdtScheme.H"
59 #include "CrankNicolsonDdtScheme.H"
60 #include "subCycle.H"
61 #include "immiscibleIncompressibleTwoPhaseMixture.H"
62 #include "turbulentTransportModel.H"
63 #include "pimpleControl.H"
64 #include "fvOptions.H"
65 #include "CorrectPhi.H"
66 #include "fvcSmooth.H"
67 #include "dynamicRefineFvMesh.H"
```

All these includes will simply introduce the needed classes for the solver. One to mention here is the `isoAdvector` class; declared in `isoAdvection.H`.

Then, in the main function, we initially read

interIsoFoam.C

```
71 int main(int argc, char *argv[])
72 {
73     argList::addNote
74     (
75         "Solver for two incompressible, isothermal immiscible fluids"
76         " using isoAdvector phase-fraction based interface capturing.\n"
77         "With optional mesh motion and mesh topology changes including"
78         " adaptive re-meshing.\n"
79         "The solver is derived from interFoam"
80     );
81
82     #include "postProcess.H"
83
84     #include "addCheckCaseOptions.H"
85     #include "setRootCaseLists.H"
86     #include "createTime.H"
87     #include "createDynamicFvMesh.H"
88     #include "initContinuityErrs.H"
89     #include "createDyMControls.H"
90     #include "createFields.H"
91     #include "initCorrectPhi.H"
92     #include "createUfIfPresent.H"
93
94     turbulence->validate();
95
96     #include "CourantNo.H"
97     #include "setInitialDeltaT.H"
```

That is, we start by adding the postProcess option to the solver, creating relevant object of the various classes and initialize the time step size using the initial Courant number. One important step here is the creation of the objects related to the computational fields. This is done in the createFields.H file. Here we initialize the dynamic pressure, p_d, the velocity field, $\mathbf{U}$, the face fluxes, ϕ and the volume fractions, α. One other important object, also initialized in createFields.H, will be the gravity vector. In createFields.H we read

$FOAM_SOLVERS/multiphase/interIsoFoam/createFields.H

```
81 #include "readGravitationalAcceleration.H"
82 #include "readhRef.H"
83 #include "gh.H"
```

We will not show the details of the file readGravitationalAcceleration.H, but note that this file reads the gravity vector specified in the constant folder and allows the user to modify the gravity vector during run time. We will skip the readhRef.H file as it has no interest for us and can be neglected. The next file, gh.H, is important to us. This is where we evaluate the inner product, $(\mathbf{g} \cdot \mathbf{x})$, found in the gravitational force. In gh.H we read

gh.H

```
1     Info<< "Calculating field g.h\n" << endl;
2     dimensionedScalar ghRef
3     (
4         mag(g.value()) > SMALL
5       ? g & (cmptMag(g.value())/mag(g.value()))*hRef
6       : dimensionedScalar("ghRef", g.dimensions()*dimLength, 0)
7     );
8     volScalarField gh("gh", (g & mesh.C()) - ghRef);
9     surfaceScalarField ghf("ghf", (g & mesh.Cf()) - ghRef);
```

If we ignore hRef (set it equal to 0), we observe that we create a volScalarField, gh and a surfaceScalarField, ghf. The volScalarField, gh takes, in each cell, the value

```
g & mesh.C()
```

and the surfaceScalarField, ghf takes, at each face, the value

```
g & mesh.Cf()
```

i.e. the inner product between the gravity vector and the cell centers and the inner product between the gravity vector and face centers, respectively. That is, we have an approximation of the inner product, present in the gravitational force (see Eq. (2.5)), in the cells and on the faces. We need the face approximation when creating the pressure equation and the cell approximation is used when we compute the total pressure from the dynamic pressure.

3.1.1 The advection step

With the construction and use of the gravity vector, we will return to the last line of the createFields.H file

$FOAM_SOLVERS/multiphse/interIsoFoam/createFields.H

```
127 isoAdvection advector(alpha1, phi, U);
```

i.e. the construction of an isoAdvection object called advector. This object is used for advecting the volume fraction (the interface) using the isoAdvector algorithm.

Now, leaving the createFields.H file and return to the source file of the interIsoFoam solver, we read

interIsoFoam.C

```
100     Info<< "\nStarting time loop\n" << endl;
101
102     while (runTime.run())
103     {
104         #include "readDyMControls.H"
105         #include "CourantNo.H"
106         #include "alphaCourantNo.H"
107         #include "setDeltaT.H"
108
109         ++runTime;
110
111         Info<< "Time = " << runTime.timeName() << nl << endl;
112
113         // --- Pressure-velocity PIMPLE corrector loop
114         while (pimple.loop())
115         {
```

We see that we move into the time loop of the PIMPLE algorithm. Each time step will begin with computing the maximum and mean Courant number for the entire domain, Courant.H and near the interface, alphaCourantNo.H. This information will be used to compute the size of next time step in setDeltaT. Afterwards, we move into the PIMPLE loop.

The initial part of the PIMPLE loop handles the case of a moving mesh. We will skip these details and move into the part of the implementation where the advection of the interface takes place. In interIsoFoam.C we read

interIsoFoam.C

```
164             #include "alphaControls.H"
165             #include "alphaEqnSubCycle.H"
```

We will not explicitly state the details of these header files, but state the outcome of these files. In alphaControls.H, we will create a constant reference to the alpha.water dictionary found in fvSolution. From this dictionary we will read the number of sub cycles used. In alphaEqnSubCycle.H we handle the case of several sub cycles in each time step. However, the main take away from this file is, regardless of having sub cycles, that we will include the header file alphaEqn.H, where the

actual advection of the interface occurs. In alphaEqn.H we, again, omit details concerned with a moving mesh and read the following line

$FOAM_SOLVERS/multiphase/interIsoFoam/alphaEqn.H

```
10    advector.advect(Sp, Su);
```

To understand what the member function, advect(Sp, Su) does, we recall from createFields.H that advector is an object of the class isoAdvection. In isoAdvection.H we read the declaration of the member function

isoAdvection.H

```
303         template < class SpType, class SuType >
304         void advect(const SpType& Sp, const SuType& Su);
```

Thus the member function is declared as a templated function with SpType and SuType as templated parameters. Although these terms will not be further investigated here, we note that they are used to include linear and constant source terms.

The definition of the advect function is found in isoAdvectionTemplates.H. The initial part is shown here

isoAdvectionTemplates.H

```
365 template<class SpType, class SuType>
366 void Foam::isoAdvection::advect(const SpType& Sp, const SuType& Su)
367 {
368     DebugInFunction << endl;
369
370     scalar advectionStartTime = mesh_.time().elapsedCpuTime();
371
372     scalar rDeltaT = 1/mesh_.time().deltaTValue();
373
374     // reconstruct the interface
375     surf_->reconstruct();
```

Here we have a scalar for the current CPU time and a scalar for the reciprocal of the time step size. This is followed by a reconstruction of the interface, i.e. the interface area normals and the interface centers. The reconstruction is made by the member data called surf_. Before continuing with the advection of the interface, i.e. the isoAdvector algorithm, we will go into details of how we create the surf_ member data and how we reconstruct the interface using the plicRDF reconstruction scheme.

The surf_ object is created in the isoAdvection.H file

isoAdvection.H

```
140             //- Pointer to reconstruction scheme
141             autoPtr<reconstructionSchemes> surf_;
```

saying that we create an object of the templated class autoPtr with the templated parameter as reconstructionSchemes. The objects of the autoPtr class are so called *smart pointers*, which ensures that the memory associated to the pointer is automatically deallocated at the end of the solver. See e.g. the work of Marić et al. [15] for more thorough discription and examples. The main take away for us here, is that we create a (smart) pointer to an object of the reconstructionSchemes class. The surf_ object is initialized in the constructor of the isoAdvection class. In isoAdvection.C we read

isoAdvection.C

```
100     surf_(reconstructionSchemes::New(alpha1_, phi_, U_, dict_))
```

i.e. the surf_ object gets initialized using the selector, New, of the reconstructionSchemes class where dict_ will be the dictionary of the volume fraction field, e.g. alpha.water, found in the fvSolution file.

Assuming that we have specified the plicRDF reconstruction scheme in the dictionary, the selector calls the constructor of the plicRDF class, which is a subclass of the reconstructionSchemes class. Frist thing to note is that this constructor will initialize important member data of the base class. In reconstructionSchemes.H we have the following member data

reconstructionSchemes.H

```
82        //- Interface area normals
83        volVectorField normal_;
84
85        //- Interface centres
86        volVectorField centre_;
```

i.e. the interface area normal and interface center member data, which will be objects of the type volVectorField thus taking vector values in each cell. These objects will be initialized to zero and will be written to the time directories when specified by the user. The constructor of the plicRDF class will also initialize the RDF_ object, which will be written to the time directories as well, and the tolerances used in the reconstruction. The tolerances will also be read form the alpha.water dictionary.

3.1.1.1 The reconstruction step

With an introduction to the surf_ object, let us discuss the reconstruct() member function. With surf_ being a (smart) pointer of the type reconstructionSchemes, we should search reconstructionSchemes.H for implementation of the reconstruct() function. In reconstructionSchemes.H we read

reconstructionSchemes.H

```
224       //- Reconstruct the interface
225       virtual void reconstruct(bool forceUpdate = true) = 0;
```

i.e. reconstruct() is a *pure* virtual function with the boolean forceUpdate set to true by default. With the function being a pure virtual function, it makes reconstructionSchemes an abstract class and we should look for the implementation of the reconstruct() member function in its sub classes. Let us again stay with the assumption of using the plicRDF reconstruction scheme, thus the pointer, surf_ will point to an object of the plicRDF class. This means that the member function reconstruct() should be implemented in the plicRDF class. In plicRDF.C we find the definition of the reconstruct() member function. The initial part reads

plicRDF.C

```
374 void Foam::reconstruction::plicRDF::reconstruct(bool forceUpdate)
375 {
376     const bool uptodate = alreadyReconstructed(forceUpdate);
377
378     if (uptodate && !forceUpdate)
379     {
380         return;
381     }
382
383     if (mesh_.topoChanging())
384     {
385         // Introduced resizing to cope with changing meshes
386         if (interfaceCell_.size() != mesh_.nCells())
387         {
388             interfaceCell_.resize(mesh_.nCells());
389         }
390     }
391     interfaceCell_ = false;
392
393     // Sets interfaceCell_ and interfaceNormal
394     setInitNormals(interpolateNormal_);
```

saying that, initially, we call the function, `alreadyReconstructed`, which is inherited from the base class hence its definition is found in `reconstructionSchemes.C`. It simply checks if the reconstrution step is needed or not. The following `if`-statement says that, if the reconstruction is up to date and we will not force an update, we will end the reconstruction step before we have begun. Next, in lines 383-390, we have some lines regarding mesh changes, which we will leave here as it is. This is followed by setting the list of booleans, `interfaceCell_` to `false`; meaning that no cells are marked as interface cell. Last line here is the call to the function `setInitNormals` with the argument `interpolateNormal_`. This argument is a boolean read from the `alpha.water` dictionary in the constructor of the `plicRDF` class. The `setInitNormals` will, with the argument set to true, initializing the interface area normals using the interface area normals from the previous time step. Also this function will update the list `interFaceCell` to return true for interface cells.

The `reconstruct()` function is then followed by

plicRDF.C

```
396     centre_ = dimensionedVector("centre", dimLength, Zero);
397     normal_ = dimensionedVector("normal", dimArea, Zero);
398
399     // nextToInterface is update on setInitNormals
400     const boolList& nextToInterface_ = RDF_.nextToInterface();
401
402     labelHashSet tooCoarse;
403
404     for (int iter=0; iter<iteration_; ++iter)
405     {
```

The main thing here is setting `centre_` and `normal_` to zero in every cell and then the beginning of a `for`-loop that runs with the number of iterations set to `iteration_`, specified by the user in `fvSolution` and read in the constructor of the `plicRDF` class. This loop will handle the iterative procedure of updating the interface normal and interface center of the interface cells (line 5 of Algorithm 1). The first part of the `for`-loop reads

plicRDF.C

```
406     forAll(interfaceLabels_, i)
407     {
408         const label celli = interfaceLabels_[i];
409         if (mag(interfaceNormal_[i]) == 0)
410         {
411             continue;
412         }
413         sIterPLIC_.vofCutCell
414         (
415             celli,
416             alpha1_[celli],
417             isoFaceTol_,
418             100,
419             interfaceNormal_[i]
420         );
421
422         if (sIterPLIC_.cellStatus() == 0)
423         {
424             normal_[celli] = sIterPLIC_.surfaceArea();
425             centre_[celli] = sIterPLIC_.surfaceCentre();
426             if (mag(normal_[celli]) == 0)
427             {
428                 normal_[celli] = Zero;
429                 centre_[celli] = Zero;
430             }
431         }
432         else
433         {
434             normal_[celli] = Zero;
435             centre_[celli] = Zero;
436         }
```

```
437     }
438 }
```

which resembles a `forAll` loop, which loops over all interface cells. The first thing, in this loop, we draw attention to is the lines 413-420. Here we use the `vofCutCell` member function of the `sIterPLIC_` object, which is initialized in the `plicRDF_` constructor. This function will place the interface center along the interface normal such that the interface will cut the interface cell in correspondence with the volume fraction. Revisit the left of Figure 2.2 for a visualization. In the next part of the loop, lines 422-436, we will set the member data `normal_` and `center_` as the interface area normals (line 424) and the interface center (line 425), respectively.

The next part of the `for` loop of the `reconstruct()` function, we will show, will be the update of the reconstructed distance function (RDF). From the body of the `reconstruct()` function, we read

plicRDF.C

```
449             RDF_.constructRDF
450             (
451                 nextToInterface_,
452                 centre_,
453                 normal_,
454                 exchangeFields_,
455                 false
456             );
457             RDF_.updateContactAngle(alpha1_, U_, nHatb);
458             gradSurf(RDF_);
459             calcResidual(residual, avgAngle);
```

What we will note here is the update of the RDF (lines 449-456) and the calculation of its gradient (line 458). The gradient will be calculated using `gradSurf` from the `plicRDF` class, which boils down to a least square gradient. The RDF is constructed using the `constructRDF` member function of the `reconstructedDistanceFunction` class. Its definition is found in `reconstructedDistanceFunction.C`. As we will see, the function distinguishes between two cases; one for interface cells and one for non-interface cells which are point neighbours to interface cells. We will here go through the main parts of the implementation. In the body of the function, `constructRDF`, we read

reconstructedDistanceFunction.C

```
267     forAll(nextToInterface,celli)
268     {
269         if (nextToInterface[celli])
270         {
271             if (mag(normal[celli]) != 0) // interface cell
272             {
273                 vector n = -normal[celli]/mag(normal[celli]);
274                 scalar dist = (centre[celli] - mesh_.C()[celli]) & n;
275                 reconDistFunc[celli] = dist;
276             }
```

where `nextTointerface` is a list containing interface cells and their point neighbours (not neccesarily interface cells). The first case, we handle, is the case of an interface cell, where we recognize interface cells with a non-zero magnitude of the interface area normal. In the above code, we see that the RDF value, `reconDistFunc`, of interface cells will be computed as the normal distance between the interface center and the cell center, i.e. similarly to Eq. (2.15), using the interface normal. In the second case, for the non-interface cells, we have the following code

reconstructedDistanceFunction.C

```
277             else // nextToInterfaceCell or level == 1 cell
278             {
279                 scalar averageDist = 0;
280                 scalar avgWeight = 0;
281                 const point p = mesh_.C()[celli];
```

```
282
283                 forAll(stencil[celli], i)
284                 {
285                     const label gblIdx = stencil[celli][i];
286                     vector n = -distribute.getValue(normal, mapNormal, gblIdx);
287                     if (mag(n) != 0)
288                     {
289                         n /= mag(n);
290                         vector c = distribute.getValue(centre,mapCentres,gblIdx);
291                         vector distanceToIntSeg = (c - p);
292                         scalar distToSurf = distanceToIntSeg & (n);
293                         scalar weight = 0;
294
295                         if (mag(distanceToIntSeg) != 0)
296                         {
297                             distanceToIntSeg /= mag(distanceToIntSeg);
298                             weight = sqr(mag(distanceToIntSeg & n));
299                         }
300                         else // exactly on the center
301                         {
302                             weight = 1;
303                         }
304                         averageDist += distToSurf * weight;
305                         avgWeight += weight;
306                     }
307                 }
308
309                 if (avgWeight != 0)
310                 {
311                     reconDistFunc[celli] = averageDist / avgWeight;
312                 }
313             }
```

We will look a bit closer to the `forAll` loop (lines 283-307), where we run through the stencil of the current cell. Note that the stencil is all point neighbours of the current cell. If a neighbouring cell has an interface area normal with magnitude different from zero (line 287), we will compute the normal distance from the cell center of the non-interface cell to the interface center of the neighbouring interface cells using Eq. (2.15). This distance is called `distToSurf` above and is found in line 292. As mentioned in the theory part, the RDF value of non-interface cells will be computed through a weighted sum. The computation of the corresponding weights are also seen above (lines 297-298) and are given the name `weight`. Here `distanceToIntSeg` corresponds to the vector $(\mathbf{x}_i - \mathbf{x}_{S,j})$ of Eq. (2.16). We see that this vector is normalized, dotted with the interface unit normal and then squared in the end. The contribution of the weights and the computed distances for the cells will be collected in `averageDist` and `avgWeight` (lines 304-305) such that the RDF value in the cell can be computed using Eq. (2.14) and saved in `reconDistFunc[celli]` in line 311.

We recall that the reconstruction of the interface is an iterative procedure. That is, we recalculate the interface center and interface normal iteratively. The process will stop after a number of iterations or when the normals will not change significantly between iterations. We will not elaborate on these details and therefore we will leave the reconstruction step here and return to the advection step, which we left when we ran into `surf_->reconstruct()` in the member function, `advect` of the class `isoAdvection`.

3.1.1.2 The isoAdvection Step

From this small detour in the `plicRDF` implementation, let us recall that the isoAdvector algorithm is about evaluating the double integral in Eq. (2.9) using the reconstructed interface in the cells.

After the reconstruction step, in the `advect` function of the `isoAdvection` class, we have the following code

`isoAdvectionTemplates.H`

```
377     // Initialising dVf with upwind values
378     // i.e. phi[facei]*alpha1[upwindCell[facei]]*dt
379     dVf_ = upwind<scalar>(mesh_, phi_).flux(alpha1_)*mesh_.time().deltaT();
380
381     // Do the isoAdvection on surface cells
382     timeIntegratedFlux();
```

Before we move into the actual commands, we will note here that the object, `dVf_`, is of the type `surfaceScalarField` and declared in `isoAdvection.H`.

The first command, we see in the code above, is the initialization of the `dVf_` values using the upwind values of the volume fractions. That is, the expression in Eq. (2.10) will initially be approximated as

$$\Delta V_f \approx \frac{\phi_f^n}{|\mathbf{S}_f|}\int_{t_n}^{t_{n+1}}\int_f H\,\mathrm{d}A\,\mathrm{d}\tau \approx \frac{\phi_f^n}{|\mathbf{S}_f|}\int_{t_n}^{t_{n+1}}\alpha_C^{n,\mathrm{up}}(t)\,\mathrm{d}\tau \approx \frac{\phi_f^n}{|\mathbf{S}_f|}\alpha_C^{n,\mathrm{up}}\Delta t$$

where $\alpha_C^{n,\mathrm{up}}$ denotes the volume fraction of the upwind cell for the face from the previous time step.

The second command, in the above code, is the call to a function called `timeIntegratedFlux`. This is a private member function of the `isoAdvection` class and it is defined in `isoAdvection.C`. This function is all about evaluating the double integral, ΔV_f, properly; i.e. evaluate it analytically using the reconstructed interface. Although, this function is the corner stone of the isoAdvector algorithm, we will skip the implementation details as it is not our main concern here.

With the evaluation of ΔV_f at all the downwind faces of the interface cells, we are ready to do the actual advection (line 6 of Algorithm 1). In the `advect` function we read

isoAdvectionTemplates.H

```
390     // Advect the free surface
391     alpha1_.primitiveFieldRef() =
392     (
393         alpha1_.oldTime().primitiveField()*rDeltaT
394       + Su.field()
395       - fvc::surfaceIntegrate(dVf_)().primitiveField()*rDeltaT
396     )/(rDeltaT - Sp.field());
```

First let us ignore the `Su` and `Sp` values as they relate to additional source terms and they are not interesting for us here. Next thing we notice is the `fvc::surfaceIntegrate` function. This will simply sum over the face values and add them to the corresponding cell values. Hence the above code reduces to

$$\alpha_C(t+\Delta t) = \alpha_C(t) - \frac{1}{|V_C|}\sum_f \Delta V_f$$

similarly to the expression in Eq. (2.17) in the Theory chapter.

This completes the advection step. Hereafter, we compute the updated density and return to the `interIsoFoam.C` source file where we will begin the pressure-velocity coupling.

3.1.2 Pressure-Velocity coupling

Before going in to the construction of the various equations in the pressure-velocity coupling, we remember the following important detail in `interIsoFoam.C`

interIsoFoam.C

```
113         // --- Pressure-velocity PIMPLE corrector loop
114         while (pimple.loop())
115         {
```

which marks the outer loop of the PIMPLE algorithm. We have gone through the first part of the PIMPLE loop, the update of the volume fraction, and now we will go into the last part, the pressure velocity coupling. In `interIsoFoam.C` we read

interIsoFoam.C

```
113             #include "UEqn.H"
114
115             // --- Pressure corrector loop
116             while (pimple.correct())
117             {
118                 #include "pEqn.H"
119             }
120
121             if (pimple.turbCorr())
122             {
123                 turbulence->correct();
124             }
125         }
```

That is, in the outer loop, we include the `UEqn.H`-file. This is followed by the pressure corrector loop (the inner loop of Algorithm 1), where we include the `pEqn.H`-file. In the last part of the outer loop, we will correct for turbulence, if specified.

Both the `UEqn.H` file and the `pEqn.H` file are found in the folder of the `interFoam` solver. In the initial part of `UEqn.H` we construct the `UEqn` object

$FOAM_SOLERS/multiphase/interFoam/UEqn.H

```
6       fvVectorMatrix UEqn
7       (
8           fvm::ddt(rho, U) + fvm::div(rhoPhi, U)
9         + MRF.DDt(rho, U)
10        + turbulence->divDevRhoReff(rho, U)
11       ==
12          fvOptions(rho, U)
13      );
```

We see that `UEqn` is an object of the `fvVectorMatrix` class, which is a type definition of `fvMatrix<vector>` i.e. an object of the `fvMatrix` class with `vector` as template parameter. Due to the assumption of inviscid fluids, we will only mention the time derivative, `fvm::ddt(rho, U)` and the nonlinear convection term, `fvm:div(rhoPhi, U)`, which we recognize as the left hand side of the momentum equation in Eq. (2.5). Let us spend some time at the argument, `rhoPhi`. This is one of the places where the `interIsoFoam` deviates from the `interFoam` solver. The object `rhoPhi` is an object of the type `surfaceScalarField` and is computed in the `alphaEqn.H` file. Here we read

$FOAM_SOLVERS/multiphase/interIsoFoam/alphaEqn.H

```
18      #include "rhofs.H"
19      rhoPhi = advector.getRhoPhi(rho1f, rho2f);
```

saying that we should consult the function, `getRhoPhi` of the `isoAdvection` class. In here we have

isoAdvection.H

```
333             tmp<surfaceScalarField> getRhoPhi
334             (
335                 const dimensionedScalar rho1,
336                 const dimensionedScalar rho2
337             ) const
338             {
339                 return tmp<surfaceScalarField>
340                 (
341                     new surfaceScalarField
342                     (
343                         "rhoPhi",
344                         (rho1 - rho2)*dVf_/mesh_.time().deltaT() + rho2*phi_
345                     )
346                 );
347             }
```

This expression is somewhat similar to the one in Eq. (2.18). If we rewrite the expression in Eq. (2.18) as

$$\sum_f \mathbf{U}_f \left([\rho] \Delta V_f + \rho^- \phi_f^n \Delta t \right) = \Delta t \sum_f \mathbf{U}_f \left(\frac{[\rho] \Delta V_f}{\Delta t} + \rho^- \phi_f^n \right)$$

the expression in the parentheses of the right hand side above will correspond to the one returned in `getRhoPhi`.

Then, if we return to the construction of `UEqn` object in the `UEqn.H` file, we note that the pressure gradient and the gravitational force is missing from the equation. These terms will be added later; in a momentum predictor step, where velocity is predicted using the old pressure values, and in the pressure equation, where we compute a pressure, that ensures a divergence free velocity field. We will skip the momentum predictor step, which is found in the `UEqn.H` file, and then move into creating and solving the pressure equation. This happens in the `pEqn.H` file, which reads

`$FOAM_SOLVERS/multiphase/interFoam/pEqn.H`

```
2   if (correctPhi)
3   {
4       rAU.ref() = 1.0/UEqn.A();
5   }
6   else
7   {
8       rAU = 1.0/UEqn.A();
9   }
10
11  surfaceScalarField rAUf("rAUf", fvc::interpolate(rAU()));
12  volVectorField HbyA(constrainHbyA(rAU()*UEqn.H(), U, p_rgh));
13  surfaceScalarField phiHbyA
14  (
15      "phiHbyA",
16      fvc::flux(HbyA)
17    + MRF.zeroFilter(fvc::interpolate(rho*rAU())*fvc::ddtCorr(U, phi, Uf))
18  );
```

The first part is the assignment of the `rAU` object (a `volScalarField`) using the `UEqn` object. This assignment will correspond to the reciprocal of the implicit coefficients for the velocities ($1/a_C$) in Eq. (2.20). As mentioned in the Theory chapter, we need these coefficients in the pressure equation; i.e. they should be evaluated on the faces. Due to this, we create the `surfaceScalarField` object, `rAUf`, which is a simple interpolation of the `rAU` object. The type of interpolation should be specified in the `fvSchemes` dictionary. The interpolation is followed by the construction of the object, `HbyA`, which corresponds to the $\mathcal{H}$ operator, defined in Eq. (2.21), divided by the implicit coefficients of the velocities, i.e. we recognize this term as $\mathcal{H}\left(\langle \mathbf{U} \rangle^m\right)/a_C$. With this operator we create `surfaceScalarField`, `phiHbyA`, corresponding to $\left(\frac{H(\langle \mathbf{U} \rangle^n)}{a_C} \right)_f \cdot \mathbf{S}_f$, which is computed using the `fvc::flux` function. Although not explicitly shown here, the `fvc::flux` function will handle the interpolation to the faces and the inner product with the face area vector, $\mathbf{S}_f$. This completes the first part of the right hand side of the pressure equation, Eq. (2.24). The last part of the right hand side is the gravitational force. From `pEqn.H` we read

`$FOAM_SOLVERS/multiphase/interFoam/pEqn.H`

```
28  surfaceScalarField phig
29  (
30      (
31          mixture.surfaceTensionForce()
32        - ghf*fvc::snGrad(rho)
33      )*rAUf*mesh.magSf()
34  );
35
36  phiHbyA += phig;
```

Here we construct the object, `phig`, which is composed of the surface tension (neglected here) and the gravitational force. The gravitational force is computed in line 32 and consists of `ghf` and

`fvc::snGrad(rho)`. We have discussed `ghf` earlier as the inner product between the gravity vector and the face center. The second part, `fvc::snGrad(rho)`, will compute the difference between the cells neighbouring the face i.e. $\frac{\rho_N^{n+1}-\rho_C^{n+1}}{|\mathbf{d}_{CN}|}$. Last we multiply by `rAUf` and the magnitude of the face area vector, `magSf`. Combining everything, we have computed

$$\left(\frac{1}{a_C}\right)_f (\mathbf{g}\cdot\mathbf{x}_f)\frac{\rho_N^{n+1}-\rho_C^{n+1}}{|\mathbf{d}_{CN}|}|\mathbf{S}_f|$$

The last line, we see in the above code snippet, is that we add the contribution of gravity to `phiHbyA`, i.e. we complete the right hand side of the pressure equation.

With this, we are ready construct the pressure equation. In `pEqn.H` we read the following

`$FOAM_SOLVERS/multiphase/interFoam/pEqn.H`

```
41      while (pimple.correctNonOrthogonal())
42      {
43          fvScalarMatrix p_rghEqn
44          (
45              fvm::laplacian(rAUf, p_rgh) == fvc::div(phiHbyA)
46          );
47
48          p_rghEqn.setReference(pRefCell, getRefCellValue(p_rgh, pRefCell));
49
50          p_rghEqn.solve(mesh.solver(p_rgh.select(pimple.finalInnerIter())));
51
52          if (pimple.finalNonOrthogonalIter())
53          {
54              phi = phiHbyA - p_rghEqn.flux();
55
56              p_rgh.relax();
57
58              U = HbyA + rAU()*fvc::reconstruct((phig - p_rghEqn.flux())/rAUf);
59              U.correctBoundaryConditions();
60              fvOptions.correct(U);
61          }
62      }
```

First we note the `while` loop, which is used for an explicit correction if a non-orthogonal mesh is used. We use an uniform Cartesian mesh, hence we will leave this here. What is more important for us, is the construction of the `p_rghEqn` object in lines 43-46. It is an object of the type `fvScalarMatrix` and is constructed using `fvm::laplacian(rAUf, p_rgh)` on the left hand side and `fvc::div(phiHbyA)` on the right hand side. We recognize this as the discrete pressure equation in Eq. (2.24). After the construction of the `p_rghEqn` object, we set the reference (dynamic) pressure followed by solving the equation, i.e. obtain a dynamic pressure in each cell of the mesh. With the assumption of no non-orthogonal corrections, we will enter the `if` statement above. That is, we update the face fluxes, `phi`, as in Eq. (2.22), and compute the velocity in each cell, as in Eq. (2.25), using the reconstruction operator, `fvc::reconstruct`.

This finalizes the inner loop of the algorithm. If more outer loops will follow, we will return to the advection of the interface (using the updated velocities), the construction of the `UEqn.H` object and again the inner loop.

3.2 Summary

We have now covered the theory and the implementation of the `interIsoFoam` solver. Within one iteration we have seen how we update the volume fraction using the isoAdvector algorithm and a reconstruction of the interface using the plicRDF algorithm. This was followed by how we couple the pressure and the velocity in order to ensure a divergence free velocity field.

To summarize and give an broader overview of the `interIsoFoam` solver, a flow chart of the processes in the solver is seen in Figure 3.1. Note here that several assumptions have been made in

this chart. That is, we have left out important aspects in order to make room for the topics discussed here. We have for instance skipped any aspects of mesh motion, the non-orthogonal correction loop and any turbulence computations. Likewise, we have made an assumption of the use of the plicRDF reconstruction scheme, no momentum predictor step and no sub cycles within each time step. Also, the flow chart contains dashed arrows. These are used to point to the subprocesses of the associated process.

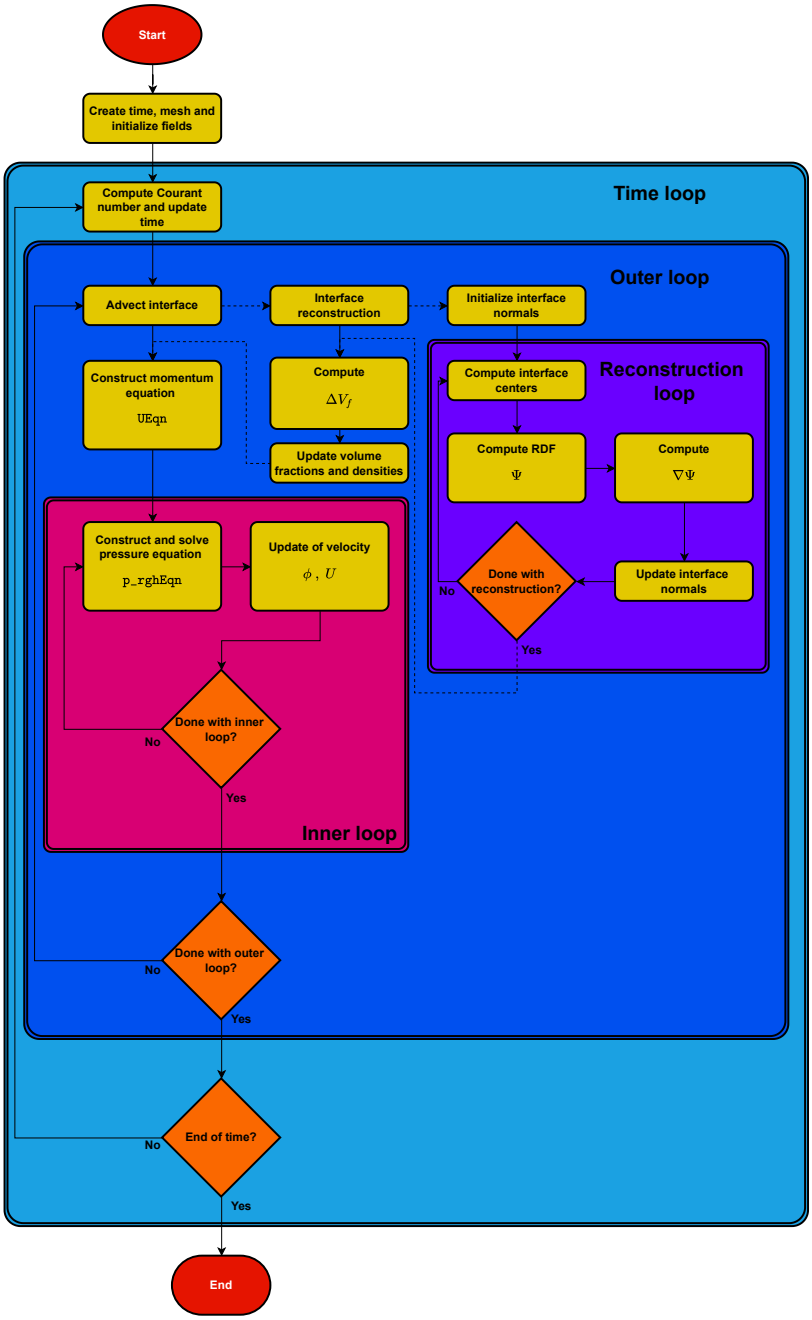


Figure 3.1: Flow chart visualizing the processes in the `interIsoFoam` solver

Chapter 4

Run a case using `interIsoFoam`

We will here specify how to use the `interIsoFoam` solver. We set up a hydrostatic test case in 2D which we call *the tilted box*. The tilted box contains two incompressible, inviscid fluids at rest. See Figure 4.1 for a visualization of the initial volume fractions resembling two fluids at rest with an planar interface between them. This case is chosen, as it illustrates an issue with the `interIsoFoam` solver. In the tilted box, the dynamic pressure should balance the gravitational forces thus leading to a zero velocity field. However, as shown later in Chapter 6, the `interIsoFoam` solver fails to retain this balance leading to spurious velocities at the interface.

The following will describe the tutorial briefly. That is, we will not show the content of all the files. Consult Appendix A to view the actual files used to set up and run the case.

With the fluids being inviscid, we will set the viscosity of each fluid equal to 0 in the `transportProperties` file found in the `constant` directory. Here we also set the density of each fluid and set the surface tension coefficient. We will use a density ratio of 1:1000 and set the surface tension coefficient to zero.

The gravity vector is specified in the `g` file of the `constant` folder. We have set the gravity vector to point in the negative z-direction with a magnitude of 9.81 m/s^2.

The simulation is laminar thus set in the `turbulenceProperties` file.

For the mesh we use a simple Cartesian uniform mesh utilizing the `blockMesh` utility. We construct a unit cube centered at $(x, y, z) = (0.5, 0, 0.5)$. The mesh will contain one `block`, which, as the problem is in 2D, is resolved only in the x- and z-direction. With this initial mesh, the cube will be tilted using the utility `transformPoints`. We will specify the option `yawPitchRoll` and then rotate the cube 30 degrees around the y axis.

The boundaries of the box will be considered as walls. For the velocity field we will employ slip boundary condition. For the volume fraction, here called `alpha.water`, we will use the `zeroGradient` boundary condition and for the dynamic pressure we use the `fixedFluxPressure` boundary condition with a value set to zero.

The initial velocity and dynamic pressure will be set to zero everywhere. The volume fraction will also, initially, be set to zero but as we consider an interfacial flow (between two phases), we should initialize the interface, i.e. the two phases. In OpenFOAM we have two utilities that may handle this initialization: `setFields` and `setAlphaField`. With these we divide the domain into sub domains in various ways; e.g. the inside/outside of a cylinder or box or with the use of a plane. In our case, the volume fractions should reflect the presence of an planar interface between the two steady fluids thus we use a plane. We will use the `setAlphaField` utility as it allows volume fractions between 0 and 1. The `setFields` utility only sets the volume fractions to 0 or 1; depending on the position of the cell center. The `setAlphaField` utility needs a `setAlphafieldDict` file in the `system` folder. It will, in our case, look like

system/setAlphafieldDict

```
18 field   alpha.water;
19 type        plane;
```

```
20 normal  (0 0 -1);
21 origin  (0 0 0.405);
```

That is, we apply the utility to the `alpha.water` field where we have specified the `type` to a `plane`. This type needs two additional settings. A normal, which points towards the reference field and sets the direction of the plane, and an origin, which sets the position of the plane. With this, we have a well-defined plane from which we may distinguish between the two fluids; everything in the normal direction will be considered as the reference fluid, e.g. the water phase. Then cells completely beneath the plane will be set to 1 and cells which are cut by the plane will have volume fraction set according to the cut. In Figure 4.1 we see the initial volume fraction after the use of the `setAlphaField` utility. Notice how some volume fraction are in between 0 and 1.

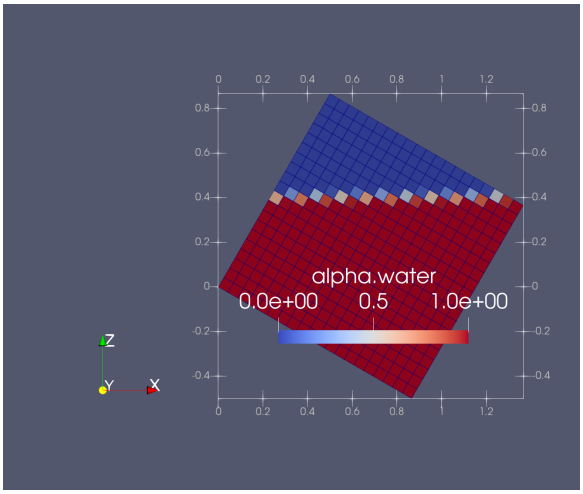


Figure 4.1: Initialization of the volume fractions using the `setAlphaField` utility

Next, we will look into the `interIsoFoam` solver settings. These should be specified in the sub dictionary, `alpha.*`[1], of the `solver` dictionary found in the `fvsolution` file. Here we read

system/fvSolution

```
18 solvers
19 {
20     alpha.water
21     {
22         isoFaceTol      1e-8;
23         surfCellTol     1e-8;
24         snapTol         1e-12;
25         clip            true;
26         reconstructionScheme plicRDF;
27         nAlphaSubCycles 1;
28         cAlpha          1;
29     }
```

What we notice is that we have the possibilities to specify the tolerances used by the solver. One is `surfCellTol`, which specifies a tolerance for when a cell is considered as an interface cell, see Eq. (2.11). Also, we have `isoFaceTol`, which is used in the construction of the interfaces. As the interfaces is constructed in accordance to the volume fraction, the `isoFaceTol` tolerance will specify the corresponding precision in which we cut the cells. Both these tolerances have a default value of `1e-8`. We also see the `snapTol` parameter and the `clip` parameter. The second specifies if we will apply brute force clipping of the volume fraction. That is, we will force the volume fractions to satisfy

$$0 \leq \alpha \leq 1$$

[1] here `*` would indicate the name of reference phase, e.g. `water`

Then, if brute force clipping is used, we use the `snapTol` in this clipping. Next, we notice the `reconstrutionScheme` parameter, which specifies which method we use for reconstructing the interface in interface cells. Here we have set it to `plicRDF`. This is followed by the `nAlphaSubCycles` parameter which sets the number of sub cycles within each time step. This is here set to 1, i.e. no sub cycles. Last we have `cAlpha`. This is actually not used by the `interIsoFoam` solver, but must be specified as it is read by the `interfaceProperties` constructor.

With the solver settings described, we will elaborate on a corresponding `Allrun` script; i.e. a script that describe the commands needed to run the case. It reads

`Allrun`

```
#!/bin/sh
cd ${0%/*} || exit 1  # Run from this directory

. $WM_PROJECT_DIR/bin/tools/RunFunctions

runApplication blockMesh
restore0Dir
cp 0/alpha.org 0/alpha.water

runApplication transformPoints -yawPitchRoll '(0 30 0)'
runApplication setAlphaField

runApplication $(getApplication)
```

From the above we see that in order to run the hydrostatic test case, we will

1. run the `blockMesh` utility
2. copy the `0.orig` folder to the `0` folder
3. rename the `alpha` file
4. rotate the mesh using the `transformPoints` utility
5. set the alpha values using the `setAlphaField` utility
6. run the case using the application found in the `controlDict` file: here `interIsoFoam`

Chapter 5

Modification of the `interIsoFoam` solver

This chapter will describe how we modify the `interIsoFoam` solver by changing the discretization of the gravitational force. The modification will be done through the `interFlow` solver of the TwoPhaseFlow library which resembles the `interIsoFoam` solver. We will initially describe how to compile the `TwoPhaseFlow` library and how to add the extension, `gravityRecon`, to it. With this extension we may use the location of the interface when computing the inner product of the gravitational force thus leading to a well-balanced solver.

The most important part of this chapter, will be an analysis highlighting the need for a proper estimation of the inner product found in the gravitational force, i.e. $(\mathbf{g} \cdot \mathbf{x})$. We will see, in the hydrostatic case, how the current implementation creates spurious velocities as it can not balance the dynamic pressure with the gravitational force. From this we will elaborate on the theory of the `gravityRecon` model, which uses ideas from the `plicRDF` reconstruction scheme to estimate the interface position properly. This interface position will then be used to evaluate the inner product of the gravitational force.

This will be followed by the implementation of the `interFlow` solver and the `gravityRecon` model. We see the generic behaviour of the `interFlow` solver and how `gravityRecon` computes the interface position using the information from the isoAdvector algorithm.

We last elaborate on how to use the `interFlow` solver of the TwoPhaseFlow library. This solver is similar to the `interIsoFoam` solver but more generic as it offers an easy way to implement new models and test them.

5.1 Compiling TwoPhaseFlow and set up the extension

First let us look into the library, TwoPhaseFlow. It can be found here: https://github.com/DLR-RY/TwoPhaseFlow. We remark that the library is developed in OpenFOAM-v1812 hence this version is needed.

The library can be fetched from the archive accompanying this report or cloned using the repository with the address given above. We have here provided a guide for cloning the repository. First we should clone the TwoPhaseFlow library from the GitHub repository

```
git clone https://github.com/DLR-RY/TwoPhaseFlow
```

Then we should add `gravityRecon` to the library. That is, we should collect the files, `gravityRecon.H` and `gravityRecon.C` (see Appendix C for these files) in a common folder, called `gravityRecon`. This folder should then be placed in the dynamic library, `SurfaceForces` in the `src` folder of `TwoPhaseFlow`

```
mv gravityRecon TwoPhaseFlow/src/surfaceForces/accelerationForce/
```

Next, we need to let the dynamic library, surfaceForces, be aware of the source code of gravityRecon. We do this by editing the Make/files file located in the SurfaceForces folder. Before the last line of the Make/files file, containing LIB = $(FOAM_USER_LIBBIN)/libsurfaceForces, we should add the following line

```
accelerationForce/gravityRecon/gravityRecon.C
```

Last, we may then compile the library by executing the Allwmake script in the TwoPhaseFlow-folder.

```
TwoPhaseFlow/Allwmake
```

and we are ready to use the interFlow solver with the gravityRecon extension. How to use and set up the solver, will be shown later in Chapter 5.4.

5.2 Theory

In the following, we will consider the motivation behind the need for a new discretization of the gravitational force in interIsoFoam. We will initially describe the hydrostatic test case which will include an analytic expression for the dynamic pressure in a control volume.

Consider the hydrostatic case of two steady, incompressible fluids at rest in a box. We will assume a horizontal interface located at the height $z = z_\Gamma$. The distance from the interface to the top of the box, where a reference pressure will be specified, will be denoted by h^- and the distance from the interface to the bottom of the box, is denoted by h^+, see Figure 5.1. The dynamic pressure can then be found analytically to be

$$p_d = p_0 + \rho^- g h^- + \begin{cases} \rho^- g z_\Gamma & , \ z > z_\Gamma \\ \rho^+ g z_\Gamma & , \ z < z_\Gamma \end{cases}$$

That is, it will be piecewise constant with a sharp jump, with the size of the density difference, at the interface. By letting $p_0 = -\rho^- g h^-$ and consider an average value in a control volume covering the interface, we can determine

$$\begin{aligned} \langle p_d \rangle_C &= \frac{1}{|V_C|} \int_C p_d \, \mathrm{d}V \\ &= \frac{1}{|V_C|} \left(\int_{C^-} \rho^- g z_\Gamma \, \mathrm{d}V + \int_{C^+} \rho^+ g z_\Gamma \, \mathrm{d}V \right) \\ &= \frac{1}{|V_C|} \left(\rho^- g z_\Gamma \int_{C^-} \mathrm{d}V + \rho^+ g z_\Gamma \int_{C^+} \mathrm{d}V \right) \\ &= \frac{1}{|V_C|} \left(\rho^- g z_\Gamma |V_{C^-}| + \rho^+ g z_\Gamma |V_{C^+}| \right) \\ &= \rho^- g z_\Gamma \frac{|V_{C^-}|}{|V_C|} + \rho^+ g z_\Gamma \frac{|V_{C^+}|}{|V_C|} \end{aligned}$$

where C^+ and C^- denote the part of the volume filled with the heavy fluid and the light fluid, respectively. Also, $|V_{C^+}|$ and $|V_{C^-}|$ denote the corresponding volumes. We may recognize $|V_{C^+}|/|V_C|$ as the part of the volume filled with the heavy fluid, i.e. the volume fraction defined in Eq. (2.8)

$$\frac{|V_{C^+}|}{|V_C|} = \alpha_C$$

similarly we have

$$\frac{|V_{C^-}|}{|V_C|} = 1 - \alpha_C$$

This yields a general expression for the average dynamic pressure in the control volume

$$\begin{aligned}\langle p_d \rangle_C &= \rho^- g z_\Gamma (1 - \alpha_C) + \rho^+ g z_\Gamma \alpha_C \\ &= [\rho] g z_\Gamma \alpha_C + \rho^- g z_\Gamma\end{aligned} \tag{5.1}$$

Regarding the velocity field, it will be zero at every point in space and time, i.e.

$$\mathbf{U} \equiv \mathbf{0}$$

This means that conservation of momentum in the hydrostatic case would be

$$\frac{\partial(\rho \mathbf{U})}{\partial t} = -\nabla p_d - (\mathbf{g} \cdot \mathbf{x}) \nabla \rho = 0$$

or in words, in order to keep the momentum constant, the dynamic pressure gradient should be balanced by the gravitational force.

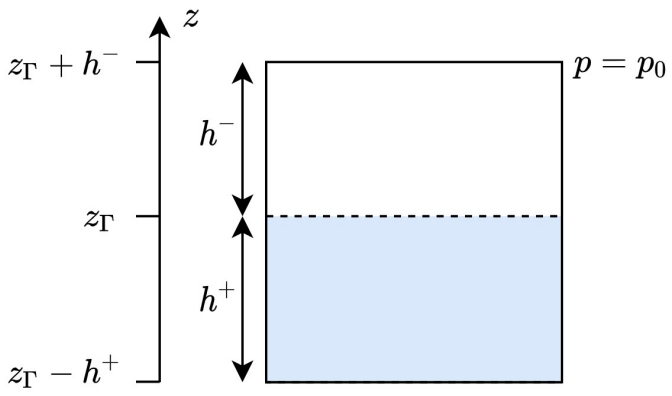


Figure 5.1: Sketch of a box containing two steady fluids at rest

5.2.1 Issue with the current implementation

We recall from Chapter 2.2, that the interIsoFoam solver couples pressure and velocity using the PIMPLE algorithm; i.e. we compute the discrete pressure equation from Eq. (2.24) and correct the velocity using Eq. (2.25). Assuming an initial velocity field of zero, $\mathbf{U}^0 = \mathbf{0}$, the discrete velocity field would be computed as

$$\langle \mathbf{U}^{n+1} \rangle_C = \frac{1}{a_C} \mathcal{R}\left(\zeta_f^{n+1}\right)$$

since $\mathcal{H}(\mathbf{U}^0) = \mathbf{0}$. Thus, in order to obtain a zero velocity field in the next time step, we should have

$$\begin{aligned}\langle \mathbf{U}^{n+1} \rangle_C = \mathbf{0} \;&\Leftrightarrow\; \mathcal{R}\left(\zeta_f^{n+1}\right) = 0 \\ &\Leftrightarrow \sum_f \mathbf{S}_f \zeta_f^{n+1} = 0\end{aligned}$$

using the expression of the reconstruction operator in Eq. (2.26). Symbolically, we may then write the last equation as

$$\begin{aligned}
&\sum_f \mathbf{S}_f \left(-\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f \right)_f^{n+1} - \left(\nabla p_d \cdot \mathbf{S}_f \right)_f^{n+1} \right) = 0 \\
&\qquad\qquad \Leftrightarrow \\
&\sum_f \mathbf{S}_f \left(\nabla p_d \cdot \mathbf{S}_f \right)_f^{n+1} + \sum_f \mathbf{S}_f \left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f \right)_f^{n+1} = 0 \\
&\qquad\qquad \Leftrightarrow \\
&\mathcal{R}\left(\left(\nabla p_d \cdot \mathbf{S}_f \right)_f^{n+1} \right) + \mathcal{R}\left(\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f \right)_f^{n+1} \right) = 0
\end{aligned} \tag{5.2}$$

saying that the reconstruction of the pressure gradient should equal the negative reconstruction of the gravitational force. This establishes a condition for obtaining a zero velocity field in the hydrostatic case. The question is then, which dynamic pressure do we compute in the discrete pressure equation?

In the hydrostatic case, the discrete pressure equation would reduce to

$$\sum_f \left(\frac{1}{a_C} \right)_f \left(\frac{\langle p_d \rangle_N^{n+1} - \langle p_d \rangle_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| = -\sum_f \left(\frac{1}{a_C} \right)_f \left((\mathbf{g}\cdot\mathbf{x}_f) \frac{\langle \rho \rangle_N^{n+1} - \langle \rho \rangle_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| \tag{5.3}$$

Then, using the expression of the density field in Eq. (2.4), we observe that the cell values of the densities can be written as

$$\begin{aligned}
\langle \rho \rangle_C^{n+1} &= \frac{1}{|V_C|} \int_C [\rho]\mathcal{H}^{n+1} + \rho^- \, \mathrm{d}V \\
&= [\rho] \frac{|V_{C^+}^{n+1}|}{|V_C|} + \rho^- \\
&= [\rho]\alpha_C^{n+1} + \rho^-
\end{aligned}$$

and if we use this expression in the discretization of the flux of the gravitational force at the faces, we get

$$\begin{aligned}
(\mathbf{g}\cdot\mathbf{x}_f) \frac{\langle \rho \rangle_N^{n+1} - \langle \rho \rangle_C^{n+1}}{|\mathbf{d}_{CN}|} |\mathbf{S}_f| &= (\mathbf{g}\cdot\mathbf{x}_f) \frac{[\rho]\alpha_N^{n+1} + \rho^- - [\rho]\alpha_C^{n+1} - \rho^-}{|\mathbf{d}_{CN}|} |\mathbf{S}_f| \\
&= [\rho](\mathbf{g}\cdot\mathbf{x}_f) \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| \\
&= -[\rho] g z_f \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f|
\end{aligned}$$

where we have evaluated the inner product in the last equality; with z_f denoting the z-coordinate (the height) of the face center. That is, the discrete pressure equation, Eq. (5.3), can be written as

$$\sum_f \left(\frac{1}{a_C} \right)_f \left(\frac{\langle p_d \rangle_N^{n+1} - \langle p_d \rangle_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| = \sum_f \left(\frac{1}{a_C} \right)_f [\rho] g z_f \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| \tag{5.4}$$

Now, the pressure we seek, in each cell, is the analytic one shown in Eq. (5.1). With this analytic expression, the discretization of the flux of the dynamic pressure gradient at a face becomes

$$\begin{aligned}
\left(\frac{\langle p_d \rangle_N^{n+1} - \langle p_d \rangle_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| &= \left(\frac{[\rho] g z_\Gamma \alpha_N^{n+1} + \rho^- g z_\Gamma - [\rho] g z_\Gamma \alpha_C^{n+1} - \rho^- g z_\Gamma}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f| \\
&= [\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|} \right) |\mathbf{S}_f|
\end{aligned}$$

Inserting this into the left hand side of Eq. (5.3) yields

$$\sum_f \left(\frac{1}{a_C}\right)_f \left(\frac{\langle p_d\rangle_N^{n+1} - \langle p_d\rangle_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| = \sum_f \left(\frac{1}{a_C}\right)_f [\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| \tag{5.5}$$

which would resemble a pressure equation that would produce the true, analytic dynamic pressure in each cell.

Let us compare the right hand side of Eq. (5.5) with the right hand side of Eq. (5.4). We observe that they are quite similar with one major difference; namely the evaluation of the inner product, $(\mathbf{g}\cdot\mathbf{x}) = -gz$. What Eq. (5.5) tells us is that to retrieve the true, analytic dynamic pressure, we should evaluate the inner product using the interface height, z_Γ, i.e. we should have

$$(\mathbf{g}\cdot\mathbf{x}) = -gz_\Gamma$$

What we observe in the current discrete pressure equation, Eq. (5.4), is that we use

$$(\mathbf{g}\cdot\mathbf{x}) = -gz_f$$

This tells us that the current pressure equation is constructed incorrectly thus it can not compute the true analytic pressure; simply because we estimate the inner product using the location of the face centers and not the true height of the interface.

This analysis does not say, that the current discrete pressure equation creates a dynamic pressure that fails to satisfy the condition in Eq. (5.2) (although we observe this particular problem in practice through simulations). The above analysis tells us only how to obtain the true, analytic dynamic pressure. But what if we had the true dynamic pressure? And also evaluated the inner product, $(\mathbf{g}\cdot\mathbf{x})$, using the interface position?

Before answering this, let us do some preliminary derivations. We recall that the true dynamic pressure yielded

$$\begin{aligned}\left(\nabla p_d\cdot\mathbf{S}_f\right)_f^{n+1} &\approx \left(\frac{\langle p_d\rangle_N^{n+1} - \langle p_d\rangle_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| \\ &= [\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f|\end{aligned}$$

hence the corresponding reconstruction becomes

$$\begin{aligned}\mathcal{R}\left(\left(\nabla p_d\cdot\mathbf{S}_f\right)_f^{n+1}\right) &= \sum_f \mathbf{S}_f\left(\nabla p_d\cdot\mathbf{S}_f\right)_f^{n+1} \\ &= \sum_f \mathbf{S}_f [\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f|\end{aligned} \tag{5.6}$$

Similarly, using the position of the interface, we have

$$\begin{aligned}\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f\right)_f^{n+1} &\approx (\mathbf{g}\cdot\mathbf{x}_\Gamma)\frac{[\rho]\alpha_N^{n+1} + \rho^- - [\rho]\alpha_C^{n+1} - \rho^-}{|\mathbf{d}_{CN}|}|\mathbf{S}_f| \\ &= -[\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f|\end{aligned}$$

yielding the following reconstruction

$$\begin{aligned}\mathcal{R}\left(\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f\right)_f^{n+1}\right) &= \sum_f \mathbf{S}_f\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f\right)_f^{n+1} \\ &= -\sum_f \mathbf{S}_f [\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1} - \alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f|\end{aligned} \tag{5.7}$$

Now, placing Eq. (5.6) and Eq. (5.7) in the condition for obtaining a zero velocity field, Eq. (5.2), we observe

$$\begin{aligned}\mathcal{R}\left(\left(\nabla p_d \cdot \mathbf{S}_f\right)_f^{n+1}\right) + \mathcal{R}\left(\left((\mathbf{g}\cdot\mathbf{x})\nabla\rho\cdot\mathbf{S}_f\right)_f^{n+1}\right) &= \sum_f \mathbf{S}_f[\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1}-\alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| \\ &\quad - \sum_f \mathbf{S}_f[\rho] g z_\Gamma \left(\frac{\alpha_N^{n+1}-\alpha_C^{n+1}}{|\mathbf{d}_{CN}|}\right)|\mathbf{S}_f| \\ &= 0\end{aligned}$$

Saying that with the true, analytic dynamic pressure and an evaluation of the inner product at the interface, we, in the hydrostatic case, satisfy the zero velocity condition in Eq. (5.2) thus obtain a zero velocity field.

The conclusion from this analysis is two fold. First, evaluating the inner product of the gravitational force at the true interface, would yield the true dynamic pressure. Second, with the analytic dynamic pressure and the position of the interface, we would satisfy the condition in Eq. (5.2) hence obtain a zero velocity as desired.

Last, let us point out that using the interface position makes sense as the gravitational force can be considered as a Dirac delta function, as pointed out in Eq. (2.6), hence it extracts the position of the interface. That is, $(\mathbf{g}\cdot\mathbf{x})$ should be evaluated at the interface - not the face center.

5.2.2 Estimation of the interface height using `isoAdvector`

Note that in the above derivation, we have assumed a constant interface height thus the result may not hold in the general case. However, the analysis tells us that the current discretization of the gravitational force is wrong. The question is now how to estimate the location of the interface in our implementation.

The following approach is inspired by the free software, *Basilisk* which is a software intended for solving flow problems, also multiphase problems, and is developed by Popinet et al. [16]. Here, the authors estimate the interface position in each cell, let us call it $\mathbf{x}_\Gamma$, and then they evaluate the inner product in each interface cell, $(\mathbf{g}\cdot\mathbf{x}_\Gamma)$. For the face values they use an average value, if the face is neighboured by two interface cells.

We will use a similar approach. That is, we will compute the interface location in each cell and then use those values to estimate the inner product, $(\mathbf{g}\cdot\mathbf{x})$. The current proposal, called `gravityRecon`, has some similarities to the creation of the RDF in the plicRDF reconstruction scheme used in the `interIsoFoam` solver. First we need a position of the interface in a cell and here we, as in the plicRDF scheme, will distinguish between interface cells and non-interface cells being point neighbours to interface cell. For the interface cells, we will compute the interface position as the point on the interface that is closest to the cell center. That is, we will compute the interface position in cell i as

$$\mathbf{x}_{\Gamma,i} = \mathbf{x}_i - \hat{\mathbf{n}}_{S,i}\big((\mathbf{x}_i - \mathbf{x}_{S,i})\cdot\hat{\mathbf{n}}_{S,i}\big) \tag{5.8}$$

where we remember $\mathbf{x}_i$ as the center of the cell, $\hat{\mathbf{n}}_{S,i}$ as the interface unit normal and $\mathbf{x}_{S,i}$ as the interface center. Note the inner product, $(\mathbf{x}_i - \mathbf{x}_{S,i})\cdot\hat{\mathbf{n}}_{S,i}$, as the RDF value from Eq. (2.15). That is, we substract the RDF value (in the interface normal direction) from the cell center in order to compute the interface position of the cell. A visualisation of the computation can be seen in Figure 5.2a. Here we have visualized the interface center, $\mathbf{x}_{S,i}$, interface unit normal, $\hat{\mathbf{n}}_{S,i}$, the cell center, the vector, $\hat{\mathbf{n}}_{S,i}\big((\mathbf{x}_i - \mathbf{x}_{S,i})\cdot\hat{\mathbf{n}}_{S,i}\big)$ and the interface position for the cell, $\mathbf{x}_{\Gamma,i}$.

As mentioned in the description of the plicRDF scheme, non-interface cells may have several point neighbours that are interface cells. In order to determine an interface position for such a cell, we will consider a weighted sum as in the plicRDF scheme. For such cells, the interface position will be calculated as

$$\mathbf{x}_{\Gamma,i} = \frac{\sum_j w_{ij}\tilde{\mathbf{x}}_{\Gamma,ij}}{\sum_j w_{ij}} \tag{5.9}$$

(a) Interface cell (b) Non-interface cell

Figure 5.2: Illustration of how to compute the interface position using cell center, interface center and interface normal

where we have a contribution from the neighbouring interface cells

$$\tilde{\mathbf{x}}_{\Gamma,ij} = \mathbf{x}_i - \hat{\mathbf{n}}_{S,j}\big((\mathbf{x}_i - \mathbf{x}_{S,j}) \cdot \hat{\mathbf{n}}_{S,j}\big) \tag{5.10}$$

and weights similar to the weights for the RDF value

$$w_{ij} = \frac{|\hat{\mathbf{n}}_{S,j} \cdot (\mathbf{x}_i - \mathbf{x}_{S,j})\,|^2}{|\mathbf{x}_i - \mathbf{x}_{S,j}|^2} \tag{5.11}$$

The interface position from the neighbouring cells is visualized in Figure 5.2b.

With the interface position set for all relevant cells, we may then evaluate the inner product, for the cells, as $(\mathbf{g} \cdot \mathbf{x}_{\Gamma,i})$. Remember, for the pressure equation, we should also evaluate this product on the faces of the cells. This will be handled by interpolation from the two adjacent cells.

5.3 Implementation

With the theory covered, let us move into the implementation detials. The current proposal is implemented as an extension to the TwoPhaseFlow library implemented in OpenFOAM-v1812. We will begin with a short introduction to the relevant aspects of the library.

The TwoPhaseFlow library can be considered as a numerical modelling framework where implementation of new methods are simplified. It offers three solvers: One for two isothermal phases, called `interFlow`, one for two compressible non-isothermal phases, called `compressibleInterFlow` and last `multiRegionPhaseChangeFlow` which handles two compressible non-isothermal phases and computes the mass transfer at the interface. We will use the `interFlow` solver hence our attention will be devoted here.

The `interFlow` solver is similar, if not equal, to the `interIsoFoam` solver in practice. It is composed of two base models; the Volume of Fluid Model and the Surface Force Model. The Volume of Fluid Model handles the advection of the volume fractions. For advection schemes it offers the `MULES` advection scheme, known from the `interFoam` solver, and the `isoAdvector` algorithm, known from the `interIsoFoam` solver. Besides from advection of the volume fractions, the `isoAdvector` algorithm offers a reconstruction of the interface through an interface center and an interface normal defined in each interface cell. Current reconstruction schemes of the `TwoPhaseFlow` library are the `gradAlpha`, `isoAlpha`, `isoSurface` and `plicRDF`. The Surface Force Model focuses on the forces arising at the interface and is divided into three sub models: *The Acceleration Model*, which computes the acceleration due to gravity, *the Delta Function Model*, which computes the Dirac Delta function and last *the Surface Tension Force Model*, which handles computation of the curvature and the surface tension coefficient. As we have neglected the surface tension, we will skip these details. The Acceleration Model is interesting for us as it handles the computation of the gravitational force

$$(\mathbf{g} \cdot \mathbf{x})\nabla\rho$$

Curently, the TwoPhaseFlow library offers one way of handling this force, which is similar to the one found in the interIsoFoam, see Eq. (2.23).

In the following we will introduce the implementation details of the interFlow solver and the new Acceleration Model, gravityRecon, which ensures a hydrostatic balance between the dynamic pressure and the gravitational force.

5.3.1 The interFlow solver

Initially, we will go through some details of the interFlow solver but we will remark that the solver is, more or less, identical to the interIsoFoam solver, which is already, to some extend, covered in this report.

Note here that the code snippets, shown in the following, contain titles that direct the reader to the location of the files in the TwoPhaseFlow library.

As in the interIsoFoam solver, we utilize the PIMPLE loop, where we, in each PIMPLE iteration, advect the interface (the volume fractions) and then do the pressure-velocity coupling through the two files, UEqn.H and pEqn.H. Before moving into the pressure-velocity coupling, we will highlight the first difference between the interIsoFoam solver and the interFlow solver. In interFlow.C we read

TwoPhaseFlow/solver/interFlow/interFlow.C

```
130             #include "alphaControls.H"
131             #include "alphaEqnSubCycle.H"
132
133             mixture.correct();
134
135             surfForces.correct();
```

That is, after the advection step (alphaEqnSubCycle.H) and the call to mixture.correct(), we use the member function, correct() of the object, surfForces. Let us spend some time to understand this object. We initialize surfForces in the createFields.H file of the interFlow solver

TwoPhaseFlow/solver/interFlow/createFields.H

```
94 surfaceForces surfForces(alpha1,phi,U,transportProperties);
```

i.e. it is an object of the class surfaceForces and constructed using the volume fraction, the face fluxes, the velocity field and the transportProperties file. The surfaceForces class is found to be the base class for handling forces on the interface. The acceleration force (due to gravity) and the surface tension force will then be implemented as sub classes of this class. We will look into the acceleration force and leave out the surface tension force. First, let us look into the surfaceForces class. It contains some important member data. In surfaceForces.H we read

TwoPhaseFlow/src/surfaceForces/surfaceForces.H

```
64         dictionary surfaceForcesCoeffs_;
65
66         //- reference to volume fraction field
67         const volScalarField& alpha1_;
68
69         const fvMesh& mesh_;
70
71
72         autoPtr<surfaceTensionForceModel> surfTenForceModel_;
73
74         autoPtr<accelerationModel> accModel_;
```

where we will mention the dictionary, surfaceForcesCoeffs_ (line 64), which is used for handling run time modifications, and the (smart) pointer accModel_ (line 74) with the template parameter accelerationModel. These data are initialized in the constructor of the class, which is defined in surfaceForces.C and reads

TwoPhaseFlow/src/surfaceForces/surfaceForces.C

```
94 Foam::surfaceForces::surfaceForces
95 (
96     const volScalarField& alpha1,
97     const surfaceScalarField& phi,
98     const volVectorField& U,
99     const dictionary& dict
100 )
101 :
102     surfaceForcesCoeffs_(dict.subDict("surfaceForces")),
103     alpha1_(alpha1),
104     mesh_(alpha1.mesh()),
105     surfTenForceModel_(nullptr),
106     accModel_(nullptr)
107 {
108     surfTenForceModel_ = surfaceTensionForceModel::New(surfaceForcesCoeffs_,alpha1,phi,U);
109     accModel_ = accelerationModel::New(surfaceForcesCoeffs_,alpha1.mesh());
110 }
```

First, let us elaborate on the initialization of the `surfaceForcesCoeffs_` in line 102. We initialize this object as the subdictionary of the object `dict` (the `transportProperties` file) called `surfaceForces`, i.e. it will point to the `surfaceForces` dictionary found in the `transportProperties` file. Second, we mention the initialization of the `accModel_` object using the selector, `New` of the `accelerationModel` class. This selector takes the dictionary `surfaceForcesCoeffs_` and the computational mesh as arguments. Before moving into the selector, let us recall that the `accelerationModel` class is found to be a subclass of the `surfaceForces` class and contains the following important member data

TwoPhaseFlow/src/surfaceForces/accelerationForce/accelerationModel.H

```
57         surfaceScalarField accf_; // ghf
58         volScalarField acc_; // gh
```

which will hold the value of the inner product, $(\mathbf{g} \cdot \mathbf{x})$, for the faces and for the cells, respectively.

Now, let us return to the selector of the `accelerationModel` class. In `accelerationModels.C` we read

TwoPhaseFlow/src/surfaceForces/accelerationForce/accelerationModels.C

```
33     word accelerationModelTypeName
34     (
35         dict.lookup("accelerationModel")
36     );
```

Here `dict` will refer to `surfaceForcesCoeffs_` (the `transportProperties` file) thus we use the key, `accelerationModel`, specified in the `surfaceForces` subdictionary found here. This key will specify which sub class of the `accelerationModel` class we use hence which constructor, the selector should return. This is where the new implementation, `gravityRecon`, will be placed; i.e. as a sub class of the `accelerationModel` class. If we specified `accelerationModel` to be `gravityRecon` in the `surfaceForces` dictionary, the `New` selector of the `accelerationModel` class will simply return the constructor of the `gravityRecon` class. This gives us an idea of how to use the `gravityRecon` model but also that the `surfaceForces` dictionary is important to the `interFlow` solver.

With some details of the `surfForces` object, let us return to the member function, `correct()`. We remember that we found `surfForces` to be an object of the type `surfaceForces`. In `surfaceForces.H` we declare and define the `correct()` function as

TwoPhaseFlow/src/surfaceForces/surfaceForces.H

```
122         void correct()
123         {
124             surfTenForceModel_->correct();
125             accModel_->correct();
```

```
126         }
```

Thus it calls the `correct()` functions of the (smart) pointers `surfTenForceModel_` and `accModel_`. With the assumption that `accModel_` was constructed using the `gravityRecon` constructor, we should move into the `gravityRecon` class to find the implementation of the `correct()` function.

5.3.1.1 The `gravityRecon` class

The `gravityRecon` class is declared as a sub class of the `accelerationModel` class in `gravityRecon.H`. The `correct()` function is not found in the `gravityRecon` class but inherited from the `accelerationModel` class. It is defined and declared in `accelerationModel.H` as

TwoPhaseFlow/src/surfaceForces/accelerationForce/accelerationModel.H

```
122         void correct()
123         {
124             calculateAcc();
125         }
```

i.e. we should consult the function `calculateAcc()` for the outcome of `correct()`. In `gravityRecon.C` we find the definition of the `calculateAcc()` function. This function will update the data `accf_` and `acc_`, i.e. the inner product, $(\mathbf{g} \cdot \mathbf{x})$, for the cells and the faces. We will show small parts of the code that will focus on the essentials of the code. First, let us look at the following

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
103         reconstructionSchemes& surf =
104             mesh.lookupObjectRef<reconstructionSchemes>("reconstructionScheme");
105
106         surf.reconstruct(false);
107
108         const volVectorField& faceCentre = surf.centre();
109         const volVectorField& faceNormal = surf.normal();
```

That is, we create a reference to the reconstruction scheme, reconstruct the interface and then extract the interface area normal and interface center. This will be followed by

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
120         forAll(surf.interfaceCell(),celli)
121         {
122             if(mag(faceNormal[celli]) != 0)
123             {
124                 vector closeP = closestDist(mesh.C()[celli],-faceNormal[celli],faceCentre[celli]);
125                 acc_[celli] = closeP & g_.value();
126             }
```

which will introduce a `forAll` loop where we run through all cells that are point neighbours with interface cells. We then, initially, check if the cells are interface cells (it has a non-zero interface normal). If so, we compute a vector, `closeP` (line 124), using the function `closestDist`. Before moving into this function, we note that we, in line 125, form the inner product between the computed vector, `closeP` and the gravity vector and save it in the `acc_` member data.

The `closestDist` function is defined as a `private` function in `gravityRecon.C`.

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
42 Foam::vector Foam::gravityRecon::closestDist(const point p, const vector n ,const vector centre)
43 {
44     vector normal = n/mag(n);
45     return p - normal*((p - centre) & normal);
46 }
```

What we observe here is that the `closestDist` function returns the interface position of Eq. (5.8). The function uses a cell center, `p`, an interface normal, `n` and an interface center, `centre`. So, with

the use of closestDist, we see that acc_[celli] simply contains the inner product between the interface position and the gravity vector for an interface cell.

Next, we will move into the case of an non-interface cell. In the calculateAcc() function we then read

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
127             else if(nextToInterface[celli])
128             {
129                 // the to the interface
130                 vector averageSurfP = vector::zero;
131                 scalar avgWeight = 0;
132                 const point p = mesh.C()[celli];
133
134                 forAll(stencil[celli],i)
135                 {
```

Here we, initially, prepare some measures; a zero vector, averageSurfP, a zero scalar, avgWeight and the cell center of the current non-interface cell. This will be followed by a forAll loop, where we will loop through all point neighbours of the non-interface cell. The loop reads

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
136                     const label& gblIdx = stencil[celli][i];
137                     vector n = -exchangeFields.getValue(faceNormal,mapNormal,gblIdx);
138                     if (mag(n) != 0)
139                     {
140                         n /= mag(n);
141                         vector c =
142                             exchangeFields.getValue(faceCentre,mapCentres,gblIdx);
143                         vector distanceToIntSeg = (c - p);
144                         vector closeP = closestDist(p,n,c);
145                         scalar weight = 0;
146
147                         if (mag(distanceToIntSeg) != 0)
148                         {
149                             distanceToIntSeg /= mag(distanceToIntSeg);
150                             weight = sqr(mag(distanceToIntSeg & n));
151                         }
152                         else // exactly on the center
153                         {
154                             weight = 1;
155                         }
156                         averageSurfP += closeP * weight;
157                         avgWeight += weight;
158                     }
159                 }
```

where we get the global index of the point neighbour cell, gblIdx in line 136, and retrieve its interface normal in line 137. This is followed by an if statement (line 138) checking if the magnitude of the interface normal is non-zero. If so, we will normalize the normal and retrieve the interface center, called c at lines 141-142. We will then compute distanceToIntSeg in line 143, which is the distance between the cell center and the interface center. This is followed by the contribution to the interface position, from the neighbouring cell, using the closestDist function in line 144, which resembles Eq. (5.10). Next, in lines 147-155 we compute the weights, called weight, which follow Eq. (5.11). Last we sum using averageSurfP and avgWeight (lines 156-157), which will denote the numerator and denominator, respectively, of the weighted sum of Eq. (5.9).

Afterwards, we leave the loop and read

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
161                 if (avgWeight != 0)
162                 {
163                     averageSurfP /= avgWeight;
164                     acc_[celli] = averageSurfP & g_.value();
165                 }
```

saying that we compute the weighted interface position for the non-interface cell and then compute the corresponding inner product, $(\mathbf{g} \cdot \mathbf{x}_{\Gamma,i})$ in line 164. With the completion of the above loop, we have provided the relevant cells with a value for the inner product, $(\mathbf{g} \cdot \mathbf{x}_{\Gamma,i})$.

The last part of `calculateAcc()` reads

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
175         acc_.correctBoundaryConditions();
176         accf_ = fvc::interpolate(acc_);
```

where we understand that we handle the value of the inner product at the boundaries and then use interpolation from adjacent cells to compute the face values of the inner product.

Summing the above, we have computed the interface positions using the information from isoAdvector. This is used to updated the computation of the inner product for the relevant cells and the relevant faces.

5.3.1.2 Pressure-velocity coupling

We remember that everything started with the call to the `correct()` function of the `surfForces` object in the `interFlow.C` file. What follows of the `interFlow.C` file is the pressure-velocity coupling using the files `UEqn.H` and `pEqn.H`. These two files are, compared to the `interIsoFoam` solver, modified in the same way for the `interFlow` solver. Due to this, the details will be given for one of the files only, the `pEqn.H` file. The main, if not the only, difference is found when we calculate the surface forces in the object called `phig`. In `pEqn.H` we read

TwoPhaseFlow/solver/interFlow/pEqn.H

```
28      surfaceScalarField phig
29      (
30          (
31              //  mixture.surfaceTensionForce()
32              surfForces.surfaceTensionForce()
33              //- ghf*fvc::snGrad(rho)
34            + surfForces.accelerationForce()
35          )*rAUf*mesh.magSf()
36      );
```

Here we observe that the original code, still visible as comments, is replaced by two calls to member functions of the object `surfForces`. The first, `surfaceTensionForce()`, is related to surface tension forces and the second, `accelerationForce()`, to acceleration forces (due to gravity). In `surfaceForces.C` we read

TwoPhaseFlow/src/surfaceForces/surfaceForces.C

```
33 Foam::tmp<Foam::surfaceScalarField> Foam::surfaceForces::accelerationForce()
34 {
35     return accModel_->accelerationForce();
36 }
```

i.e. the `accelerationForce()` function returns the member function `accelerationForce()` of the (smart) pointer `accModel_`, which we found to be of the type `accelerationModel`. In `accelerationModel.H` we then read

TwoPhaseFlow/src/surfaceForces/accelerationForce/accelerationModel.C

```
33         virtual tmp<surfaceScalarField> accelerationForce() = 0;
```

thus `accelerationForce()` is a pure virtual function hence it should be defined in the sub classes of the `accelerationModel` class. The definition of the member function `accelerationForce()` in the `gravityRecon` class is found in `gravityRecon.C`. Here we read

TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon/gravityRecon.C

```
Foam::tmp<Foam::surfaceScalarField> Foam::gravityRecon::accelerationForce()
{
    const fvMesh& mesh = acc_.mesh();
    const volScalarField& rho = mesh.lookupObject<volScalarField>("rho");
    return -accf()*fvc::snGrad(rho);
}
```

That is, we initially get a reference to the computational mesh which we use to look up the density field, `rho`. Then we return (the negative of) the member function, `accf()` multiplied by `fvc::snGrad(rho)`. The member function, `accf()` is defined and declared in `accelerationModel.H` as

TwoPhaseFlow/src/surfaceForces/accelerationForce/accelerationModel.H

```
const surfaceScalarField& accf() const
        {
            return accf_;
        }
```

i.e. it returns the `accf_` member data thus the computation of the inner product, $(\mathbf{g} \cdot \mathbf{x})$, for the faces, which we updated using the `surfForces.correct()` call.

Returning back to the `pEqn.H` file, we now know that `surfForces.accelerationForce()` returns `-accf_*fvc::snGrad(rho)`, which we recognize as

$$-(\mathbf{g} \cdot \mathbf{x}_{\Gamma,f}) \frac{\langle \rho \rangle_N^{n+1} - \langle \rho \rangle_C^{n+1}}{|\mathbf{d}_{CN}|}$$

with $(\mathbf{g} \cdot \mathbf{x}_{\Gamma,f})$ being the interpolated inner product on the face.

This completes the implementation part of the `gravityRecon` model, which simply computes the inner product, $(\mathbf{g} \cdot \mathbf{x})$ using an estimate of the interface position. The motivation for this was the fact that the gradient of the density field was considered as a Dirac delta function which extracted the interface position. This should in practice ensure a balance between pressure and the gravitational force in a hydrostatic case. Before showing if this is the case, let us describe how to use and set up the `interFlow` solver of the TwoPhseFlow library.

5.4 Run a case using interFlow

We will again consider the tilted box case hence a lot will be similar to the description in Chapter 4 and therefore not repeated here. This includes the properties of the phases, the mesh creation and the initialization of the fields. One thing we should be aware of here is the `setAlphaField` utility. In order to obtain the same initialization of the volume fractions in OpenFOAM-v1812, we will use the following `setAlphaFieldDict`

system/setAlphaFieldDict

```
field       alpha.water;
type        plane;
direction   (0 0 1);
origin      (0 0 0.405);
```

where the attentive reader notices that the `plane` still needs two additional settings but the `normal` setting is called `direction` here and it is also pointing in the opposite direction. That is, in OpenFOAM-v1812, it will be the volumes in the *opposite* direction of the normal, that will be set as the reference fluid.

Regarding the `interFlow` solver, one major difference will be the dictionary called `surfaceForces` present in the `transportProperties` file of the `constant` folder. Here we should, as also noted in the previous section, specify which models we use. In `transportProperties` we have then added the following

constant/transportProperties

```
surfaceForces
{
    sigma                       0.00;
    surfaceTensionForceModel  gradAlpha;
    accelerationModel         gravityRecon;
    deltaFunctionModel        alphaCSF;
    gravity                   ( 0 0 -9.81 );
}
```

Here we specify the surface tension coefficient, `sigma`, which we set to zero as we neglect surface tension. Next, we have set the `surfaceTensionForceModel` to `gradAlpha`. This is followed by the `accelerationModel`, which specifies which model we use when handling the gravitational force. Here we have set it to `gravityRecon`. We note here that, currently, the TwoPhaseFlow library offers an acceleration model called `gravity`, which is similar to the one in `interIsoFoam`. We should also specify how to handle the Dirac delta function by setting the `deltaFunctionModel` parameter. This is set to `alphaCSF`, which reduces to the approximation

$$\mathbf{n}_\Gamma \delta_\Gamma \approx \nabla \alpha$$

Finally, we specify the gravity vector using the parameter, `gravity`. The take away message here is that using the above dictionary, we can easily change between the models used by the solver, e.g. how we discretize the gravitational force.

Last thing, we have to set, is the `interFlow` solver settings. These are set in the `alpha.water` dictionary in `fvSolution`. Here we read

constant/transportProperties

```
solvers
{
    alpha.water
    {
        isoFaceTol      1e-8;
        surfCellTol     1e-8;
        snapTol         1e-12;
        clip            true;
        reconstructionScheme plicRDF;
        nAlphaSubCycles 1;
        cAlpha          1;

        advectionScheme isoAdvection;
    }
```

We notice that it is similar to the settings for the `interIsoFoam` solver. One additional parameter is `advectionSchemes`, which is used to specify how we should advect the volume fractions. Here we set it to `isoAdvection`.

Chapter 6

Results

We will here devote some space to show results using the original `interIsoFoam` solver and also the `interFlow` solver of the TwoPhaseFlow library. The `interFlow` solver will use the extension, `gravityRecon`, that utilizes the interface information from the isoAdvector algorithm, in order to estimate the inner product, $(\mathbf{g} \cdot \mathbf{x})$, more accurately.

We will here return to the hydrostatic test case of Chapter 4, the tilted box. That is, we initially have the volume fractions depicted in Figure 6.1a. The velocity field will initially be set to zero and, in the ideal case, it should remain zero (to machine precision) throughout the simulation.

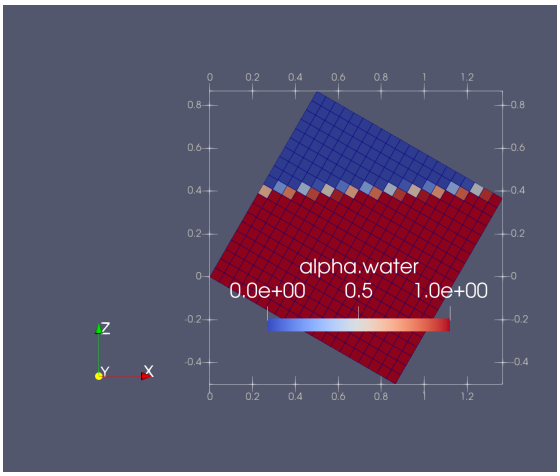
First let us consider the `interIsoFoam` solver. The case set-up is similar to the one described in Chapter 4 hence it will not be described again. See Appendix A for all the case files.

The magnitude of the velocity after one single time step is visualized in Figure 6.1b. The visualization is done in ParaView [17] using log scale coloring. Here we observe magnitudes which are significantly different from zero; the largest are of the order 10^{-1} m/s. What we can deduce from here, and the analysis in Chapter 5.2.1, is that we are not able to compute a dynamic pressure that will balance the gravitational force hence leading to the correct velocity of zero.

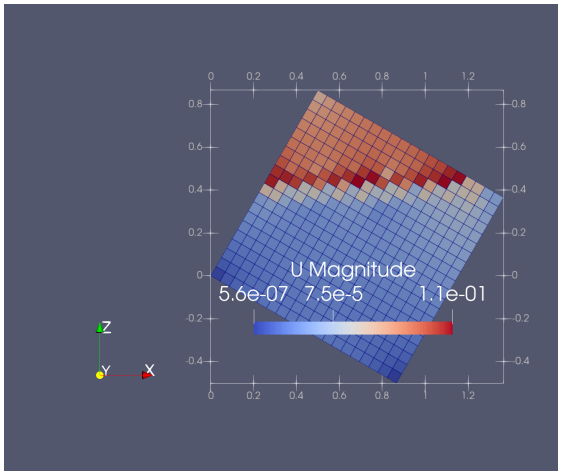
As adressed in Chapter 5.2.1 we should obtain a zero velocity field with the use of the true interface position. Therefore, let us consider the same simulation with the `interFlow` solver of the TwoPhaseFlow library with the `gravityRecon` extension. The set-up is similar to the one described in Chapter 5.4 hence omitted here. Again, see Appendix B for the full case set up.

Similar as before, we will visualize the velocity magnitudes after one time step. The magnitudes are visible in Figure 6.1c; again using the log scale coloring of ParaView. Here we observe velocities with magnitude that are significantly reduced compared to the `interIsoFoam` solver. The largest magnitudes are of the order 10^{-5} m/s. From here we deduce that introducing an estimate of the interface position will significantly reduce the numerical errors in the hydrostatic case. That is, we ensure a dynamic pressure that balances the gravitational force here.

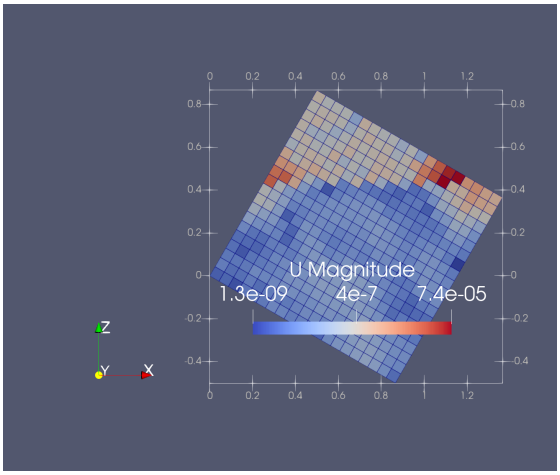
It is still a question whether or not, it reduces the errors in more general cases. Last result, we will show, is a simulation of the progression of a stream function wave. This was shown in Chapter 1 using the `interIsoFoam` solver and repeated in Figure 6.2 for convenience. The same simulation has been run using the `interFlow` solver with the `gravityRecon` extension. It is shown in Figure 6.3. If we compare the two figures, we observe that we have removed the spurious currents found at the interface using the `interIsoFoam` solver. This tells us that we, using the hydrostatic case, have created a numerical scheme which reduced the error created in the gravitational force and seems useful for more general cases as well.



(a) Initial volume fraction of the tilted box case



(b) Magnitude of velocity after 1 timestep of the size $\Delta t = 0.001$ using the `interIsoFoam` solver



(c) Magnitude of velocity after 1 timestep of the size $\Delta t = 0.001$ using the `interFlow` solver with `gravityRecon`

Figure 6.1: Results from the tilted box case

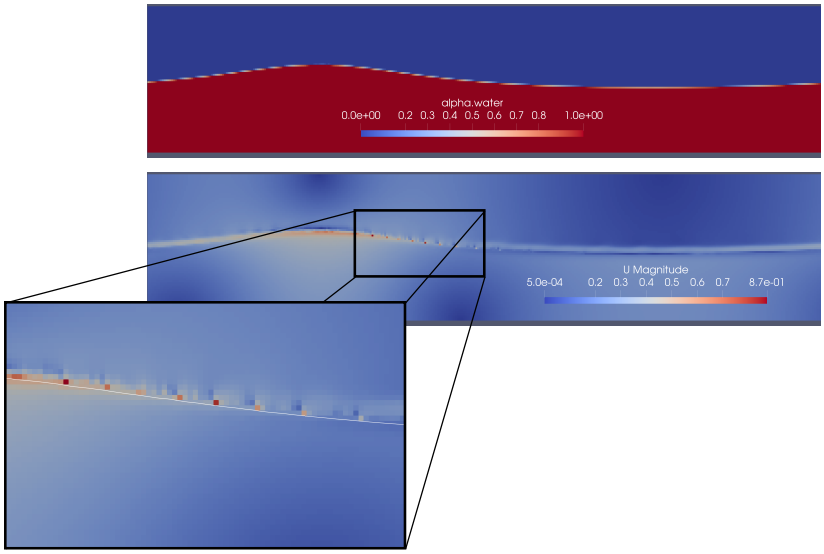


Figure 6.2: Simulation of a stream function wave using `interIsoFoam`. On top we have the volume fraction and below we have the velocity magnitude where the $\alpha = 0.5$ contour have been coloured in white. Notice how the spurious currents are present just above the interface.

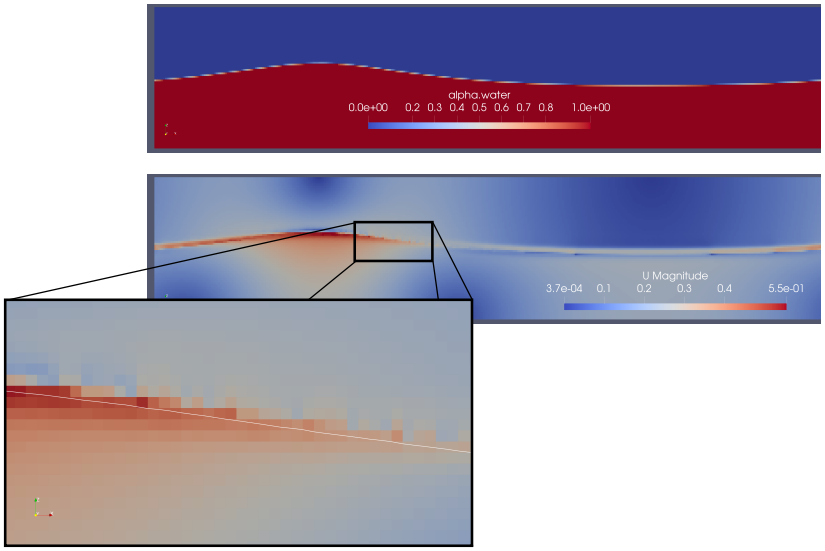


Figure 6.3: Simulation of a stream function wave using `interFlow` solver with the `gravityRecon` extension. On top we have the volume fraction and below we have the velocity magnitude where the $\alpha = 0.5$ contour have been coloured in white. Notice how velocity behaves more reasonable.

Bibliography

[1] J. Roenby, H. Bredmose, and H. Jasak, "A computational method for sharp interface advection," *Royal Society Open Science*, vol. 3, p. 160405, 2016.

[2] B. E. Larsen, D. R. Fuhrman, and J. Roenby, "Performance of interfoam on the simulation of progressive waves," *Coastal Engineering Journal*, vol. 61, no. 3, pp. 380–400, 2019.

[3] J. D. Fenton, "Numerical methods for nonlinear waves," *Advances in Coastal and Ocean Engineering*, vol. 5, pp. 241–324, 1999.

[4] H. Scheufler and J. Roenby, "Twophaseflow: An openfoam based framework for development of two phase flow solvers," 2021.

[5] G. Tryggvason, R. Scardovelli, and S. Zaleski, *Direct Numerical Simulations of Gas–Liquid Multiphase Flows*. Cambridge University Press, 2011.

[6] H. Rusche, "Computational fluid dynamics of dispersed two-phase flows at high phase fractions," 2003.

[7] S. Popinet, "Numerical models of surface tension," *Annual Review of Fluid Mechanics*, vol. 50, no. 1, pp. 49–75, 2018.

[8] F. Moukalled, L. Mangani, and M. Darwish, *The Finite Volume Method in Computational Fluid Dynamics*. Springer, Cham, 2016.

[9] J. H. Ferziger, M. Perić, and R. L. Street, *Computational Methods for Fluid Dynamics*. Springer, Cham, 2020.

[10] T. Holzmann, "Mathematics, numerics, derivations and openfoam(r)." Holzmann CFD, Release 7.0, https://Holzmann-cfd.de.

[11] J. Roenby, H. Bredmose, and H. Jasak, *IsoAdvector: Geometric VOF on General Meshes*, pp. 281–296. In: Nóbrega J., Jasak H. (eds) OpenFOAM. Springer, Cham, 2019.

[12] H. Scheufler and J. Roenby, "Accurate and efficient surface reconstruction from volume fraction data on general meshes," *Journal of Computational Physics*, vol. 383, pp. 1–23, 2019.

[13] S. S. Deshpande, L. Anumolu, and M. F. Trujillo, "Evaluating the performance of the two-phase flow solver interFoam," *Computational Science and Discovry*, vol. 5, no. 1, p. 014016, 2012.

[14] H. J. Aguerrea, C. I. Pairettib, C. M. Veniera, S. M. Damiána, and N. M. Nigroa, "An oscillation-free flow solver based on flux reconstruction," *Journal of Computational Physics*, vol. 365, pp. 135–148, 2018.

[15] T. Maric, J. Höpken, and K. G. Mooney, "The openfoam(r) technology primer." http://dx.doi.org/10.13140/2.1.2532.9600, 2021.

[16] Popinet and collaborators, "Basilisk." http://basilisk.fr, 2013-2020.

[17] U. Ayachit, "The paraview guide: A parallel visualization application," 2015.

Study questions

1. How do you use the `interIsoFoam` solver in OpenFOAM?
2. How do we solve the governing equations of interfacial flows in `interIsoFoam`?
3. How do we ensure a balance between the pressure and the gravitational force in the hydrostatic case?
4. How do we use the `interFlow` solver?
5. What is the difference between the `interIsoFoam` and the `interFlow` solver?
6. How do we add an extension to the TwoPhaseFlow library?

Appendix A

The tilted box case using the `interIsoFoam` solver

We have here collected the code for setting up the tutorial for the tilted box case using the `interIsoFoam` solver. The appendix is divided into subsections devoted to the `Allrun` and `Allclean` scripts and each of the folders found in the tutorial; namely the `constant` folder, the `system` folder and the `0.orig` folder, which hold the initial and boundary condition for the fields.

A.1 The `Allrun` and `Allclean` scripts

A.1.1 `Allrun`

```
#!/bin/sh
cd ${0%/*} || exit 1  # Run from this directory

. $WM_PROJECT_DIR/bin/tools/RunFunctions

runApplication blockMesh
restore0Dir
cp 0/alpha.org 0/alpha.water

runApplication transformPoints -yawPitchRoll '(0 30 0)'
runApplication setAlphaField

runApplication $(getApplication)
```

A.1.2 `Allclean`

```
#!/bin/sh
cd ${0%/*} || exit 1    # Run from this directory

. $WM_PROJECT_DIR/bin/tools/CleanFunctions

cleanCase0
```

A.2 The `constant` folder

A.2.1 `g`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -2 0 0 0 0];
value           ( 0 0 -9.81 );


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

A.2.2 transportProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

phases (water air);

water
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 0;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1000;
}

air
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 0;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1;

}

sigma           sigma [ 1 0 -2 0 0 0 0 ] 0.00;


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

A.2.3 turbulenceProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType  laminar;


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

A.3 The system folder

A.3.1 blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

scale 1;

vertices
(
    (0 -0.5 0)
    (1 -0.5 0)
    (1 0.5 0)
    (0 0.5 0)
    (0 -0.5 1)
    (1 -0.5 1)
    (1 0.5 1)
    (0 0.5 1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (100 1 100) simpleGrading (1 1 1)
);

edges
```

```
(
);

boundary
(
    left
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    right
    {
        type patch;
        faces
        (
            (1 2 6 5)
        );
    }
    top
    {
        type patch;
        faces
        (
            (4 5 6 7)
        );
    }
    bottom
    {
        type patch;
        faces
        (
            (0 3 2 1)
        );
    }
    front
    {
        type empty;
        faces
        (
            (0 1 5 4)
        );
    }
    back
    {
        type empty;
        faces
        (
            (2 3 7 6)
        );
    }
);

mergePatchPairs
(
);

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

A.3.2 controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
```

```
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     interIsoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         0.01;

deltaT          0.001;

writeControl    timeStep;

writeInterval   1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

adjustTimeStep  no;

maxCo           0.2;
maxAlphaCo      0.2;
maxCapillaryNum 1e8;

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

A.3.3 fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
```

```
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    div(rhoPhi,U)  Gauss limitedLinearV 1;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    pd;
    pcorr;
    alpha1;
}


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

A.3.4 fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

```

```
solvers
{
    alpha.water
    {
        isoFaceTol      1e-8;
        surfCellTol     1e-8;
        snapTol         1e-12;
        clip            true;
        reconstructionScheme plicRDF;
        nAlphaSubCycles 1;
        cAlpha          1;
    }

    pcorrFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-10;
        relTol          0;
    }

    p_rgh
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-07;
        relTol          0.05;
    }

    p_rghFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-07;
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }

    UFinal
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-07;
        relTol          0;
    }
}

PIMPLE
{
    momentumPredictor no;
    nOuterCorrectors 1;
    nCorrectors     3;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}


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

A.3.5 setAlphaFieldDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      setAlphaFieldDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

field alpha.water;
type plane;
normal (0 0 -1);
origin (0 0 0.405);

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

A.4 The 0.orig folder

A.4.1 U

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform (0 0 0);

boundaryField
{
    "(left|right|top|bottom)"
    {
        type            slip;
    }

    "(front|back)"
    {
        type            empty;
    }
}

```

```
35
36 // ************************************************************************* //
```

A.4.2 alpha.org

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       volScalarField;
13     object      alpha;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [0 0 0 0 0 0 0];
18
19 internalField   uniform 0;
20
21 boundaryField
22 {
23     "(left|right|top|bottom)"
24     {
25         type            zeroGradient;
26     }
27     "(front|back)"
28     {
29         type            empty;
30     }
31 }
32
33 // ************************************************************************* //
```

A.4.3 p_rgh

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       volScalarField;
13     object      p_rgh;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [1 -1 -2 0 0 0 0];
18
19 internalField   uniform 0;
20
21 boundaryField
22 {
23     "(left|right|top|bottom)"
```

```
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }

    "(front|back)"
    {
        type            empty;
    }
}

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

Appendix B

The tilted box case using the `interFlow` solver

We have here collected the code for setting up the tutorial for the tilted box case using the `interFlow` solver of the TwoPhaseFlow library. The appendix is divided into subsections devoted to the `Allrun` and `Allclean` scripts and each of the folders found in the tutorial; namely the `constant` folder, the `system` folder and the `0.orig` folder, which hold the initial and boundary condition for the fields.

B.1 The `Allrun` and `Allclean` scripts

B.1.1 `Allrun`

```
#!/bin/sh
cd ${0%/*} || exit 1    # Run from this directory

. $WM_PROJECT_DIR/bin/tools/RunFunctions
application=$(sed -ne "s/^application\s*\(.*\);/\1/p" system/controlDict)

runApplication blockMesh
restore0Dir
cp 0/alpha.org 0/alpha.water

runApplication transformPoints -yawPitchRoll '(0 30 0)'
runApplication setAlphaField

runApplication $(getApplication)
```

B.1.2 `Allclean`

```
#!/bin/sh
cd ${0%/*} || exit 1    # Run from this directory

. $WM_PROJECT_DIR/bin/tools/CleanFunctions

cleanCase0
```

B.2 The `constant` folder

B.2.1 `g`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -2 0 0 0 0];
value           ( 0 0 -9.81 );


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

B.2.2 transportProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

phases (water air);

water
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 0;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1000;
}

air
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 0;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1;

}

sigma           sigma [ 1 0 -2 0 0 0 0 ] 0.00;

surfaceForces
{
  sigma                     0.00;
  surfaceTensionForceModel  gradAlpha;
```

```
  accelerationModel         gravityRecon;
  deltaFunctionModel        alphaCSF;
  gravity                   ( 0 0 -9.81 );
}


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

B.2.3 turbulenceProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType  laminar;


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

B.3 The system folder

B.3.1 blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 1;

vertices
(
    (0 -0.5 0)
    (1 -0.5 0)
    (1 0.5 0)
    (0 0.5 0)
    (0 -0.5 1)
    (1 -0.5 1)
```

```
    (1 0.5 1)
    (0 0.5 1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 1 20) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    left
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    right
    {
        type patch;
        faces
        (
            (1 2 6 5)
        );
    }
    top
    {
        type patch;
        faces
        (
            (4 5 6 7)
        );
    }
    bottom
    {
        type patch;
        faces
        (
            (0 3 2 1)
        );
    }
    front
    {
        type empty;
        faces
        (
            (0 1 5 4)
        );
    }
    back
    {
        type empty;
        faces
        (
            (2 3 7 6)
        );
    }
);

mergePatchPairs
(
);
```

```
95
96 // ************************************************************************* //
```

B.3.2 controlDict

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     location    "system";
14     object      controlDict;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 application     interFlow;
19
20 startFrom       startTime;
21
22 startTime       0;
23
24 stopAt          endTime;
25
26 endTime         0.01;
27
28 deltaT          0.001;
29
30 writeControl    timeStep;
31
32 writeInterval   1;
33
34 purgeWrite      0;
35
36 writeFormat     ascii;
37
38 writePrecision  6;
39
40 writeCompression off;
41
42 timeFormat      general;
43
44 timePrecision   6;
45
46 runTimeModifiable yes;
47
48 adjustTimeStep  no;
49
50 maxCo           0.2;
51 maxAlphaCo      0.2;
52 maxCapillaryNum 1e8;
53
54 maxDeltaT       1;
55
56 // ************************************************************************* //
```

B.3.3 fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    div(rhoPhi,U)  Gauss limitedLinearV 1;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    pd;
    pcorr;
    alpha1;
}


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

B.3.4 fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
```

```
|   \\  /    A nd           | Web:      www.OpenFOAM.org                    |
|    \\/     M anipulation  |                                               |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    alpha.water
    {
        isoFaceTol      1e-8;
        surfCellTol     1e-8;
        snapTol         1e-12;
        clip            true;
        reconstructionScheme plicRDF;
        nAlphaSubCycles 1;
        cAlpha          1;

        advectionScheme isoAdvection;
    }

    pcorrFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-10;
        relTol          0;
    }

    p_rgh
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-07;
        relTol          0.05;
    }

    p_rghFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-07;
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }

    UFinal
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-07;
        relTol          0;
    }
}
```

```
73
74 PIMPLE
75 {
76     momentumPredictor no;
77     nOuterCorrectors 1;
78     nCorrectors     3;
79     nNonOrthogonalCorrectors 0;
80     pRefCell        0;
81     pRefValue       0;
82 }
83
84
85 // ************************************************************************* //
```

B.3.5 setAlphaFieldDict

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     location    "system";
14     object      setAlphaFieldDict;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 field       alpha.water;
19 type        plane;
20 direction   (0 0 1);
21 origin      (0 0 0.405);
22
23 // ************************************************************************* //
```

B.4 The 0.orig folder

B.4.1 U

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       volVectorField;
13     location    "0";
14     object      U;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 dimensions      [0 1 -1 0 0 0 0];
```

```
19
20 internalField   uniform (0 0 0);
21
22 boundaryField
23 {
24     "(left|right|top|bottom)"
25     {
26         type            slip;
27     }
28 /*
29     atmosphere
30     {
31         type            pressureInletOutletVelocity;
32         value           uniform (0 0 0);
33     }
34 */
35     "(front|back)"
36     {
37         type            empty;
38     }
39 }
40
41
42 // ************************************************************************* //
```

B.4.2 alpha.org

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  2.2.0                                 |
5  |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       volScalarField;
13     object      alpha;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [0 0 0 0 0 0 0];
18
19 internalField   uniform 0;
20
21 boundaryField
22 {
23     "(left|right|top|bottom)"
24     {
25         type            zeroGradient;
26     }
27     "(front|back)"
28     {
29         type            empty;
30     }
31 }
32
33 // ************************************************************************* //
```

B.4.3 p_rgh

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p_rgh;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    "(left|right|top|bottom)"
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
/*
    atmosphere
    {
        type            totalPressure;
        p0              uniform 0;
        U               U;
        phi             phi;
        rho             rho;
        psi             none;
        gamma           1;
        value           uniform 0;
    }
*/
    "(front|back)"
    {
        type            empty;
    }
}

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

Appendix C

The gravityRecon acceleration model

We have here collected the code for the acceleration model `gravityRecon` added to the TwoPhaseFlow library. It consists of two files; the `gravityRecon.H` file and the `gravityRecon.C` file. We note here that the files should be collected in a common folder, called `gravityRecon`, and added to the acceleration models of the TeoPhaseFlow library. That is, place the `gravityRecon` folder at

`TwoPhaseFlow/src/surfaceForces/accelerationForce/gravityRecon`

C.1 gravityRecon.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

Class
    Foam::gravityRecon

Description

SourceFiles
    gravityRecon.C
    newPhaseChangeModel.C

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

```

```
35 #ifndef gravityRecon_H
36 #define gravityRecon_H
37
38 #include "typeInfo.H"
39 #include "volFields.H"
40 #include "dimensionedScalar.H"
41 #include "autoPtr.H"
42 #include "zoneDistribute.H"
43 #include "accelerationModel.H"
44 #include "markInterfaceRegion.H"
45
46 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
47
48 namespace Foam
49 {
50
51 /*---------------------------------------------------------------------------*\
52                Class gravityRecon Declaration
53 \*---------------------------------------------------------------------------*/
54
55 class gravityRecon
56     : public accelerationModel
57 {
58
59 private:
60
61     //-
62         //- Stabilisation for normalisation of the interface normal
63         const dictionary& gravityDict_;
64         dimensionedVector g_;
65
66     // Private Member Functions
67
68         //- Disallow copy construct
69         gravityRecon(const gravityRecon&);
70
71         //- Disallow default bitwise assignment
72         void operator=(const gravityRecon&);
73
74         vector closestDist(const point p, const vector n ,const vector center);
75
76
77         protected:
78
79         //- Re-calculate the acceleration
80         virtual void calculateAcc();
81
82 public:
83
84     //- Runtime type information
85     TypeName("gravityRecon");
86
87
88     // Constructors
89
90         //- Construct from components
91         gravityRecon
92         (
93             const dictionary& dict,
94             const fvMesh& mesh
95         );
96
97
98     //- Destructor
99     virtual ~gravityRecon()
100    {}
101
102    virtual tmp<surfaceScalarField> accelerationForce();
```

```
103
104
105 };
106
107
108 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
109
110 } // End namespace Foam
111
112 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
113
114 #endif
115
116 // ************************************************************************* //
```

C.2 gravityRecon.C

```
1  /*---------------------------------------------------------------------------*\
2    =========                 |
3    \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
4     \\    /   O peration     |
5      \\  /    A nd           | Copyright (C) 2011 OpenFOAM Foundation
6       \\/     M anipulation  |
7  -------------------------------------------------------------------------------
8  License
9      This file is part of OpenFOAM.
10
11     OpenFOAM is free software: you can redistribute it and/or modify it
12     under the terms of the GNU General Public License as published by
13     the Free Software Foundation, either version 3 of the License, or
14     (at your option) any later version.
15
16     OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
17     ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
18     FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
19     for more details.
20
21     You should have received a copy of the GNU General Public License
22     along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.
23
24 \*---------------------------------------------------------------------------*/
25
26 #include "gravityRecon.H"
27 #include "addToRunTimeSelectionTable.H"
28
29 #include "reconstructionSchemes.H"
30
31 #include "plane.H"
32 #include "fvc.H"
33
34 // * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //
35
36 namespace Foam
37 {
38     defineTypeNameAndDebug(gravityRecon, 0);
39     addToRunTimeSelectionTable(accelerationModel,gravityRecon, components);
40 }
41
42 Foam::vector Foam::gravityRecon::closestDist(const point p, const vector n ,const vector centre)
43 {
44     vector normal = n/mag(n);
45     return p - normal*((p - centre) & normal);
46 }
47
48
```

```
49 // * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //
50
51 Foam::gravityRecon::gravityRecon
52 (
53     const dictionary& dict,
54     const fvMesh& mesh
55 )
56 :
57     accelerationModel
58     (
59         typeName,
60         dict,
61         mesh
62     ),
63     gravityDict_(dict),
64     g_
65     (
66         "gravity",
67         dimAcceleration,
68         vector(0,0,0)
69     )
70
71
72 {
73     // calculateAcc();
74 }
75
76 // * * * * * * * * * * * * * * * * Public Access Member Functions  * * * * * * * * * * * * * * * * * //
77
78 void Foam::gravityRecon::calculateAcc()
79 {
80     const fvMesh& mesh = acc_.mesh();
81     zoneDistribute& exchangeFields = zoneDistribute::New(mesh);
82     g_.value() = gravityDict_.lookup("gravity");
83     dimensionedScalar hRef("hRef",dimLength, gravityDict_.lookupOrDefault("hRef",0));
84
85
86     dimensionedScalar ghRef
87     (
88         mag(g_.value()) > SMALL
89       ? g_ & (cmptMag(g_.value())/mag(g_.value()))*hRef
90       : dimensionedScalar("ghRef", g_.dimensions()*dimLength, 0)
91     );
92
93
94     acc_ = (g_ & mesh.C()) - ghRef;
95     accf_ = ((g_ & mesh.Cf()) - ghRef);
96
97     if(mesh.foundObject<reconstructionSchemes>("reconstructionScheme"))
98     {
99         reconstructionSchemes& surf = mesh.lookupObjectRef<reconstructionSchemes>("
     reconstructionScheme");
100
101         surf.reconstruct(false);
102
103         const volVectorField& faceCentre = surf.centre();
104         const volVectorField& faceNormal = surf.normal();
105
106         boolList nextToInterface(mesh.nCells(),false);
107         markInterfaceRegion markIF(mesh);
108
109         markIF.markCellsNearSurf(surf.interfaceCell(),1,nextToInterface);
110
111         exchangeFields.setUpCommforZone(nextToInterface,true);
112
113         Map<vector > mapCentres =
114             exchangeFields.getDatafromOtherProc(nextToInterface,faceCentre);
115         Map<vector > mapNormal =
```

```
116             exchangeFields.getDatafromOtherProc(nextToInterface,faceNormal);
117
118         const labelListList& stencil = exchangeFields.getStencil();
119
120         forAll(surf.interfaceCell(),celli)
121         {
122             if(mag(faceNormal[celli]) != 0)
123             {
124                 vector closeP = closestDist(mesh.C()[celli],-faceNormal[celli],faceCentre[celli]);
125                 acc_[celli] = closeP & g_.value();
126             }
127             else if(nextToInterface[celli])
128             {
129                 // the to the interface
130                 vector averageSurfP = vector::zero;
131                 scalar avgWeight = 0;
132                 const point p = mesh.C()[celli];
133
134                 forAll(stencil[celli],i)
135                 {
136                     const label& gblIdx = stencil[celli][i];
137                     vector n = -exchangeFields.getValue(faceNormal,mapNormal,gblIdx);
138                     if (mag(n) != 0)
139                     {
140                         n /= mag(n);
141                         vector c =
142                             exchangeFields.getValue(faceCentre,mapCentres,gblIdx);
143                         vector distanceToIntSeg = (c - p);
144                         vector closeP = closestDist(p,n,c);
145                         scalar weight = 0;
146
147                         if (mag(distanceToIntSeg) != 0)
148                         {
149                             distanceToIntSeg /= mag(distanceToIntSeg);
150                             weight = sqr(mag(distanceToIntSeg & n));
151                         }
152                         else // exactly on the center
153                         {
154                             weight = 1;
155                         }
156                         averageSurfP += closeP * weight;
157                         avgWeight += weight;
158                     }
159                 }
160
161                 if (avgWeight != 0)
162                 {
163                     averageSurfP /= avgWeight;
164                     acc_[celli] = averageSurfP & g_.value();
165                 }
166
167             }
168             else
169             {
170                 // do nothing
171             }
172
173         }
174
175         acc_.correctBoundaryConditions();
176         accf_ = fvc::interpolate(acc_);
177
178     }
179     else
180     {
181         Info << "notFound" << endl;
182     }
183
```

```
184 }
185
186 Foam::tmp<Foam::surfaceScalarField> Foam::gravityRecon::accelerationForce()
187 {
188     const fvMesh& mesh = acc_.mesh();
189     const volScalarField& rho = mesh.lookupObject<volScalarField>("rho");
190     return -accf()*fvc::snGrad(rho);
191 }
192
193
194
195
196 // ************************************************************************* //
```

Index