CFD with OpenSource software

A course at Chalmers University of Technology
Taught by HÅKAN NILSSON

Project work:

interSettlingFoam

Developed for OpenFOAM-2.2.0

Author:
Pedram Ramin

Peer reviewed by:
Olivier Petit
Ulf Nilsson

December 9, 201

Content

1. Introduction

This work presents the description of two solvers *settlingFoam* and *interFoam* and their application to solve the equations describing the pressure and velocity profile in a circular pipe. The phases are water, dispersed phase as suspended solids and air. The suspended solid, carried out along the pipe, settles due to gravity. It is possible to describe this settling of the dispersed phase using the *settlingFoam* solver. This solver is based on the averaged form of the Eulerian two-phase flow. The basic concept of this model is to consider the mixture as a whole rather than two separate phases. Considering unfilled running pipe with open surface, there is air-water interaction at the interface between the phases. To predict this mixed behavior the *interFoam* solver is chosen to investigate the sharing processes of entraining of the air into the water body and ejection of the water droplets into the air.

In this report first, each solver is described shortly and their governing equations are outlined. The corresponding codes in OpenFOAM are then presented here to compare and examine the formulation and the solution procedure. For each solver, the case setup is examined by describing how the mesh is generated and what boundary and initial conditions are chosen.

Finally, a solver called *interSettlinFoam* is introduced by combining the two above mentioned solvers. The aim of this solver is to predict the flow motion at the free surface of the water phase (where this phase interacts with air) while the settling of the dispersed phase in the water phase is also considered. The main procedure is based on the assumption that the settling velocity can be decoupled from the main flow field, which is not entirely correct but it makes the solvers coupling much easier.

As for the introduced cases, the *dahl* case is used to test *settlingFoam* solver and waterChannel case tested for *interFoam* solver. The latter case also used for *interSettlingFoam* solver. Meshing method is based on the method used in *ercoftacConicalDiffuser* in *case0*.

2. settlingFoam

The main forces shaping the flow field are buoyancy, the settling velocity of the dispersed phase and the rheology of the mixture. The involved phases are isothermal, incompressible and without phase change.
To simulate the flow field of two phase flow, two continuity equations and two momentum equations are needed. Two fluid models can give very detailed and accurate predictions of a two- phase flow field. However, the approach does have some drawbacks. There are some inherent difficulties in modeling the inter-facial momentum transfer terms, which couple the two phases together.
An alternative method is to consider the mixture as a whole rather than two separate phases. This method is based on the averaged form of the Eulerian two-phase flow. In this approach kinematic constitutive equation plays the role of a relative equation of motion and one of the momentum equations is eliminated.

2.1. Mixture property

The purpose of this section is to describe the relative motion between the phases. This is however expressed due to gravitational settling of the dispersed phase. Derivation of drift velocity is based on the relative velocity of motion. The phase fraction $\alpha_i$is defined as the volume fraction of phase *i* with respect to the whole volume. The mixture density is defined as:

$$\rho_m = \alpha_1\rho_1 + \alpha_2\rho_2 \tag{1}$$

The mixture pressure can be defined in a similar way. The mixture center of mass velocity v_m is defined as:

$$v_m = \frac{\alpha_1\rho_1 v_1 + \alpha_2\rho_2 v_2}{\rho_m} \tag{2}$$

First the velocity of each phase v_{km} shall be correlated to the mixture center of mass velocity, v_m and the phase velocity v_k (here k=2). The relation of these velocities is illustrated in Figure. 1. The relative velocity v_r and the velocity of each phase with respect to the mass center of the mixture are defined as:

$$v_r = v_1 - v_2 \tag{3}$$
$$v_{km} = v_k - v_m \tag{4}$$

By inserting the diffusion velocity in equation (4) this velocity can be defined as:

$$v_{2m} = -\frac{\alpha_1\rho_1}{\alpha_2\rho_2}v_{2m} = -\frac{\alpha_1\rho_1}{\rho_m}(v_1 - v_2) \tag{5}$$

The center velocity of the volume of the mixture (the drift velocity of each phase) is important since the constitutive equations for these velocities are rather simple. The volumetric fluxes of each phase are defined as:

$$j_k = \alpha_k v_k \tag{6}$$

However, the mixture center of mass velocity v_m, is not the same as total volumetric flux ($j = \sum \alpha_k v_k$) due to different densities of the phases. The drift velocity can be defined in terms of relative velocity:

$$v_{1j} = -\alpha_1 v_r,\ v_{2j} = -\alpha_2 v_r \tag{7}$$

Finally the relation between the drift velocities v_{kj} and the diffusion velocity v_{km} can be written as:

$$v_{2m} = -\frac{\rho_1}{\rho_2} v_{2j} = -\frac{\alpha_1 \rho_1}{\alpha_2 \rho_m} v_{1j} \tag{8}$$

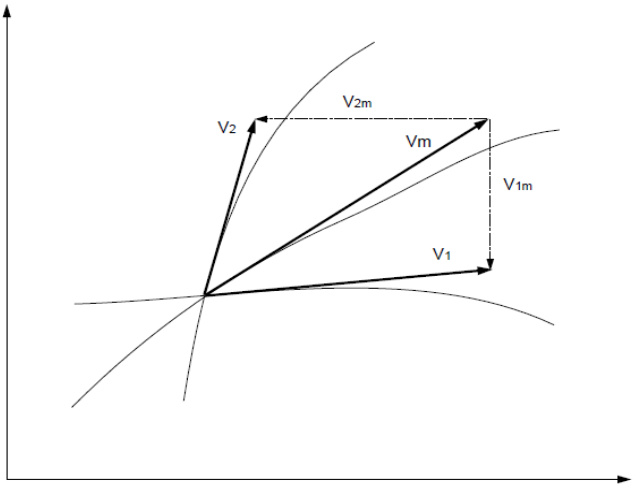

Figure 1. stream line and velocity relationship in two-phase flow (Brennan 2001)

2.2. Governing equations

Adding the two continuity equations from the two-fluids model results in a single mixture continuity equation:

$$\frac{\partial \rho_m}{\partial t} + \nabla.(\rho_m u_m) = 0 \tag{9}$$

Mixture Momentum Equation is:

$$\frac{\partial \rho_m u_m}{\partial t} + \nabla.(\rho_m u_m u_m) = -\nabla.P_m + \nabla.\left[\tau + \tau^t\right] - \nabla.(\frac{\alpha_d}{1-\alpha_d} \frac{\rho_c \rho_d}{\rho_m} v_{dj} v_{dj}) + \rho_m g + M_m \tag{10}$$

Drift Equation:

$$\frac{\partial \alpha_d}{\partial t} + \nabla.(\alpha_d \ v_m) = -\nabla.(\frac{\alpha_d \rho_c}{\rho_m} v_{dj}) + \nabla.\Gamma \nabla \alpha_d \tag{11}$$

where α_d is the volume fraction of dispersion phase; P_m, ρ_m and u_m are the average flow properties; V_s is the settling velocity, and Γ is the diffusion coefficient. Γ is made equal to the molecular and turbulent viscosity, in direct analogy to the mass and momentum transport. The turbulence viscosity is determined by the turbulent kinetic energy, k, and by the rate of dissipation of turbulence kinetic energy, ε, according to:

$$\mu_t = \rho_W C_\mu \frac{k^2}{\varepsilon} \tag{12}$$

where $C_\mu = 0.09$ is a constant. The semi-empirical model transport equations for k and ε are given by the following expression:

$$\frac{\partial \rho k}{\partial t} + \nabla.(\rho v k) = \nabla.\left[\frac{\mu_t}{\sigma_t}\nabla k\right] + P_k + G_k - \rho\varepsilon \tag{13}$$

$$\frac{\partial \rho \varepsilon}{\partial t} + \nabla.(\rho v \varepsilon) = \nabla.\left[\frac{\mu_t}{\sigma_t}\nabla \varepsilon\right] + C_{1,\varepsilon}\rho\frac{\varepsilon}{k}(P_k + G_k - C_{3,\varepsilon}G_k) - C_{2,\varepsilon}\rho\frac{\varepsilon^2}{k} \tag{14}$$

where P is the generation of turbulence kinetics energy due to mean velocity gradients:

$$P = \mu_t\left(u'^2 + v'^2 + w'^2\right) \tag{15}$$

and G corresponds to the generation of turbulence kinetics energy due to buoyancy:

$$G = -g\frac{\mu_t}{\rho_w \sigma_t}\frac{\partial \rho}{\partial z} \tag{16}$$

The molecular viscosity is calculated by the rheology models that describe the non-Newtonian behavior of sludge.

2.3. Solver structure

The *settlingFoam* has the following directory structure with tree shape illustration. The viscosity files are describing the non-Newtonian behavior of the fluid as this solver is written for sludge with high concentration of solids. The same fluid is considered here as well. In this work only *alphaEqn* and *UEqn* are presented here, due to their major role in combining *settlingFoam* and *interFoam* solvers, which will be explained in section 4.

```
├── alphaEqn.H
├── calcVdj.H
├── compressibleContinuityErrs.H
├── correctViscosity.H
├── createFields.H
├── kEpsilon.H
├── Make
│   ├── files
│   └── linuxGccDPOpt
│       ├── dependencies
│       ├── dependencyFiles
│       ├── dontIncludeDeps
│       ├── files
│       ├── filesMacros
│       ├── includeDeps
│       ├── localObjectFiles
│       ├── objectFiles
│       ├── options
│       ├── settlingFoam.o
│       └── sourceFiles
│   └── options
├── pEqn.H
├── plasticViscosity.H
├── settlingFoam.C
├── settlingFoam.dep
├── UEqn.H
├── wallDissipation.H
├── wallFunctions.H
├── wallViscosity.H
└── yieldStress.H
```

2.3.1. alphaEqn.H

In order to predict the distribution of the dispersed phase within the mixture, a convection diffusion equation is derived from the continuity equation of the dispersed phase. In this *H* file the drift equation is solved according to the equation (11).

```
{
    surfaceScalarField phiAlpha
    (
        IOobject
        (
            "phiAlpha",
            runTime.timeName(),
            mesh
        ),
        phi + rhoc*(mesh.Sf() & fvc::interpolate(Vdj))
    );

    FvScalarMatrix AlphaEqn
    (
          fvm::ddt(rho, Alpha)
        + fvm::div(phiAlpha, Alpha)
        - fvm::laplacian(mut, Alpha)
    );

    Info<< "Solid phase fraction = "
        << Alpha.weightedAverage(mesh.V()).value()
        << "  Min(Alpha) = " << min(Alpha).value()
        << "  Max(Alpha) = " << max(Alpha).value()
        << endl;

    Alpha.min(1.0);
    Alpha.max(0.0);

    rho == rhoc/(scalar(1) + (rhoc/rhod - 1.0)*Alpha);
    alpha == rho*Alpha/rhod;
}
```

2.3.2. UEqn.H

This file describes the momentum equation according to equation (10).

```
  // Solve the Momentum equation

    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      + fvc::div
        (
            (Alpha/(scalar(1.001) - Alpha))*(sqr(rhoc)/rho)*Vdj*Vdj,
            "div(phiVdj,Vdj)"
        )
      - fvm::laplacian(muEff, U, "laplacian(muEff,U)")
      - (fvc::grad(U) & fvc::grad(muEff))
    //- fvc::div(muEff*dev2(T(fvc::grad(U))))
    );
```

```
    UEqn.relax();

    if (pimple.momentumPredictor())
    {
        solve
        (
            UEqn
          ==
            fvc::reconstruct
            (
                (
                  - ghf*fvc::snGrad(rho)
                  - fvc::snGrad(p_rgh)
                )*mesh.magSf()
            )
        );
    }
```

2.4 Set up the case

2.4.1. Mesh Generation

The case is set up in OpenFOAM based on the tutorial case *dahl*. The modification starts with generating the Mesh using *blockMesh*. The Mesh is based on the O grid method for the pipe, proposed by OpenFOAM Turbo machinery working group. The detail is well explained in The ERCOFTAC conical diffuser case-study. In Figure 2 the grid plane at the cross section of the pipe is illustrated. By defining the number of cells in radial direction as well as tangential direction the mesh of the cross section can be generated (see the codes below). In Figure 3. Two cross sections (as indicated by *A* and *B* in *blockMeshDict*) are defined and then the number of cells for the length of the pipe is specified. Figure 4 shows the side view of the pipe and the numbers of cells are presented in table 1.

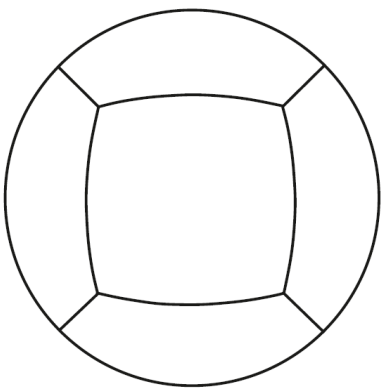

Figure 2. O grid method at the cross section

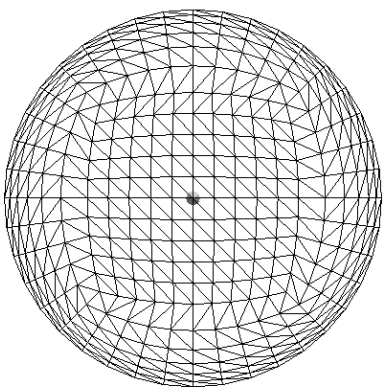

Figure3. Defining the cells in the cross section

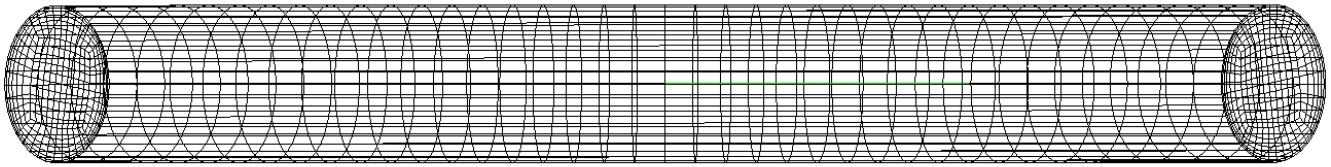

Figure 4. Side view of the pipe

Table 1. Dimensions of the pipe

Radius	0.13 m
Length	2 m
Number of cells in radial direction	6
Number of cells in tangential direction	10
Number of cells in longitudinal direction	40
Expansion ratio in radial direction	4

```
convertToMeters 1;
vertices
(
//Plane A:
        (0.0643 -0.064 -2) // Vertex A0 = 0
        (0.0643 0.0643 -2) // Vertex A1 = 1
        (-0.064 0.0643 -2) // Vertex A2 = 2
        (-0.064 -0.0643 -2) // Vertex A3 = 3
        (0.0919 -0.0919 -2) // Vertex A4 = 4
        (0.09192 0.0919 -2) // Vertex A5 = 5
        (-0.09192 0.0919 -2) // Vertex A6 = 6
        (-0.09192 -0.0919 -2) // Vertex A7 = 7
//Plane B:
        (0.0643 -0.064 0) // Vertex B0 = 8
        (0.0643 0.0643 0) // Vertex B1 = 9
        (-0.064 0.0643 0) // Vertex B2 = 10
        (-0.064 -0.0643 0) // Vertex B3 = 11
        (0.0919 -0.0919 0) // Vertex B4 = 12
        (0.09192 0.0919 0) // Vertex B5 = 13
        (-0.09192 0.0919 0) // Vertex B6 = 14
        (-0.09192 -0.0919 0) // Vertex B7 = 15
);
// Defining blocks:
blocks
(
//Blocks between plane A and plane B:
// block0 - positive x O-grid block
        hex (5 1 0 4 13 9 8 12 ) AB
        (6 10 40)
        simpleGrading (5 1 1)
// block1 - positive y O-grid block
        hex (6 2 1 5 14 10 9 13 ) AB
```

```
        (6 10 40)
        simpleGrading (5 1 1)
// block2 - negative x O-grid block
        hex (7 3 2 6 15 11 10 14 ) AB
(6 10 40)
        simpleGrading (5 1 1)
// block3 - negative y O-grid block
        hex (4 0 3 7 12 8 11 15 ) AB
(6 10 40)
        simpleGrading (5 1 1)
// block4 - central O-grid block
        hex (0 1 2 3 8 9 10 11 ) AB
(10 10 40)
        simpleGrading (1 1 1)
);
edges
(
//Plane A:
        arc 0 1 (0.0728 0 -2)
        arc 1 2 (0 0.0728 -2)
        arc 2 3 (-0.0728 0 -2)
        arc 3 0 (0 -0.0728 -2)
        arc 4 5 (0.13 0 -2)
        arc 5 6 (0 0.13 -2)
        arc 6 7 (-0.13 0 -2)
        arc 7 4 (0 -0.13 -2)
//Plane B:
        arc 8 9 (0.0728 0 0)
        arc 9 10 (0 0.0728 0)
        arc 10 11 (-0.0728 0 0)
        arc 11 8 (0 -0.0728 0)
        arc 12 13 (0.13 0 0)
        arc 13 14 (0 0.13 0)
        arc 14 15 (-0.13 0 0)
        arc 15 12 (0 -0.13 0)
);
// Defining patches:
patches
(
        patch inlet
(
        (1 5 4 0)
        (2 6 5 1)
        (3 7 6 2)
        (0 4 7 3)
        (3 2 1 0)
)
patch outlet
(
        (13 9 8 12)
        (14 10 9 13)
        (15 11 10 14)
        (12 8 11 15)
        (8 9 10 11)
```

```
)
wall wallDiffuser
        (
        (4 5 13 12)
        (5 6 14 13)
        (6 7 15 14)
        (7 4 12 15)
        )
);
```

2.4.2. Initial and boundary conditions

The boundary condition shall be specified in *0* folder for all the variables. They are defined at each patch, inlet, outlet and surrounding wall. The initial condition is also defined for each variable. As for the velocity, pressure and α, the initial and boundary conditions are defined according to table 2.

Table 2. Initial and boundary conditions

		U	P_rgh	Alpha
dimensions		[0 1 -1 0 0 0 0]	[1 -1 -2 0 0 0 0]	[0 0 0 0 0 0 0]
Internal field		Uniform (0 0 0)	Uniform (0 0 0)	Uniform 0
inlet	type	fixedValue	zeroGradient	fixedValue
	value	uniform (0 0 0.01)		uniform 0.01
outlet	type	pressureInletOutletVelocity	fixedValue	zeroGradient
	value	uniform (0 0 0)	uniform 0	
wallDiffuser	type	fixedValue	zeroGradient	zeroGradient
	value	uniform (0 0 0)		

2.4.3. Solver Setting

The solver settings in *fvSchemes*, *fvSolution* and *ControlDict* are considered to be as table 4.

Table 4. The solver settings

terms		Numerical schemes
ddt Schemes		euler
grad Schemes		Gauss upwind
div Schemes	div(phi,U)	Gauss upwind grad(U)
	div(phi,k)	Gauss upwind
	div(phi,epsilon)	Gauss upwind
	div(phiAlpha,Alpha)	Gauss limitedLinear01 1;
	div(phiVdj,Vdj)	Gauss linear;
laplacian Schemes	laplacian(muEff,U)	Gauss linear corrected
	laplacian((rho*(1|A(U))),p_rgh)	Gauss linear corrected
	laplacian(DkEff,k)	Gauss linear corrected;
	laplacian(DepsilonEff,epsilon)	Gauss linear corrected;
	laplacian(mut,Alpha)	Gauss linear corrected;
Interpolation Schemes		linear

	U	P_rgh	alpha	k	epsilon
Solver	PBiCG	PCG	PBiCG	PBiCG	PBiCG
Tolerance	1e-07	1e-07	1e-07	1e-07	1e-07
Relaxation factor	0.3	-	0.8	0.5	0.5

Control variable	value
Maximum Corrant number	0.5
Delta T	0.1
Maximum delta T	1
endTime	1000

2.4.4. Running the Case

Figure 5 shows the settling of the dispersed phase along the pipe. *alpha* is defined as the fraction of the dispersed phase. Figure 6 shows the flow field along the pipe.

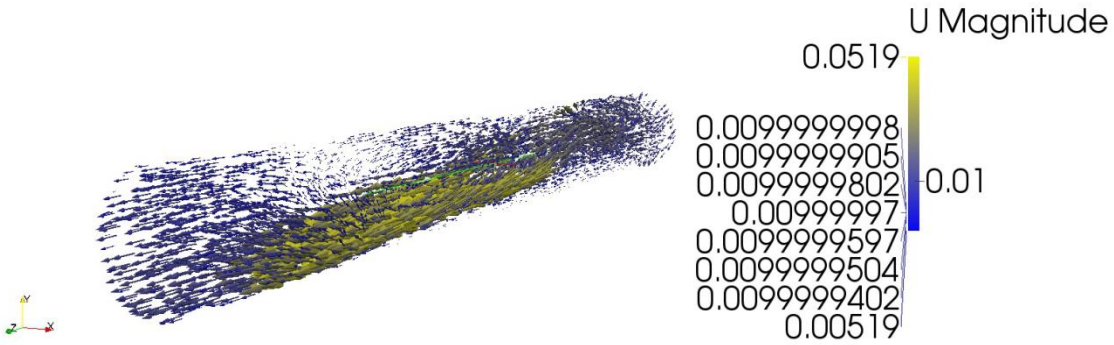

Figure 5. Settling of the dispersed phase along the pipe, indicated by dispersed fraction value

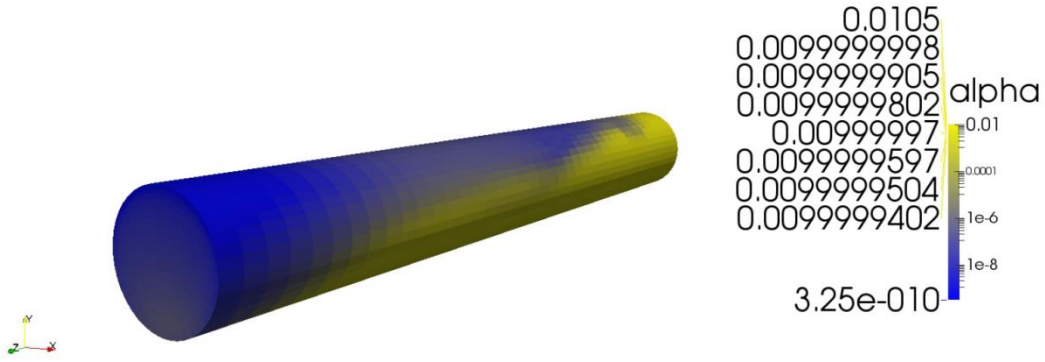

Figure 6. the flow field across the pipe

3. InterFoam

interFoam is a two-phase incompressible flow solver in OpenFOAM. This solver applies finite volume discretization on collocated grids to solve the two-phase equations of flow. Although the flow variables are cell centered, in the solution procedure their face interpolated values are also employed.

3.1. Phase fraction

Consider liquid-phase region R_1 and gas-phase region R_2 in a physical domain Ω. An indicator function II(x,t) can be defined as a function of position and time. The function has a singular nature which can be either 0 or 1 at a certain time and position. *interFoam* uses the Volume of Fluid (VOF) method which uses the volume fraction α as an indicator to specify which portion of the corresponding cell is occupied by the fluid phase. By integration of the indicator function over computational cell Ω_i the volume fraction can be defined as:

$$\alpha(x_i, t) = \frac{1}{|\Omega_i|}\int_{\Omega_i} \mathrm{II(x,t)dV} \tag{17}$$

$$\alpha(x_i, t) = \begin{cases} 1 & \text{Occupied by the fluid 1} \\ 0 < \alpha < 1 & \text{In the interface} \\ 0 & \text{Occupied by the fluid 1} \end{cases}$$

3.2. Conservation of mass and momentum

Considering the conservation of mass, the continuity equation can be written in terms of liquid fraction field. The phase fraction α for the cells which are completely liquid or completely gas is 1 and 0 respectively.

$$\frac{\partial \alpha_1}{\partial t} + \boldsymbol{\nabla}.(U_1\alpha_1) = 0 \tag{18}$$

To solve the above equation the discontinuous nature of the liquid fraction field must be preserved. The numerical scheme in *interFoam* keeps this constraint met by modifying the advection term. By integrating the above equation over the computational cell Ω_i and discrete the counterpart using forward Euler scheme, this equation can be written as:

$$\frac{\alpha_i^{n+1} - \alpha_i^n}{\Delta t} = -\frac{1}{|\,\Omega_i|}\sum_{f\epsilon\partial\,\Omega_i}(F_u + \alpha_M F_c)^n \tag{19}$$

In which

$$F_u = \theta_f \alpha_{f,upwind} \tag{20}$$

$$F_c = \theta_f\alpha_f + \theta_{rf}\alpha_{rf}\left(1-\alpha_{rf}\right) - F_u \tag{21}$$

$$\theta_f = U_f.S_f \tag{22}$$

At the interface advection term appears as the summation over the cell faces f of Ω. The delimiter α_M (also Zalesak limiter) defined to be 1 at the interface and zero elsewhere. This implies that away from the interface where $\alpha_M = 0$, the advection is treated with the straightforward upwind scheme and everywhere else the right hand side of equation (21) combines the compression flux $\theta_{rf}\alpha_{rf}\left(1-\alpha_{rf}\right)$ with a higher order scheme for $\theta_f\alpha_f$.

The S_f corresponds to the surface of the cell P. U_f is the face velocity obtained by the point-to-point interpolation between cell P and N. θ_f is the volume flux. In *interFoam* linear interpolation is used (i.e.($U_f = U_P + U_N)/2$). The phase fraction value at the interface α_r is obtained using mixture of central and upwind scheme. Concerning the compressive flux, the θ_{rf} is given by:

$$\varphi_{rf} = min\left(C_Y \frac{|\varphi_f|}{|S_f|}, max\left[\frac{|\varphi_f|}{|S_f|}\right]\right)(n_f, S_f) \tag{23}$$

In two phase flow simulation the flow is governed by this momentum equation:

$$\frac{\partial \rho u}{\partial t} + \nabla.(\rho u u) = -\nabla P^* + \nabla.\tau + \rho g + F \tag{24}$$

In which τ is the viscosity stress tensor and F is the source of the momentum concerning the surface tension:

$$F = \int_{\Gamma \cap \Omega_i} \sigma \kappa n d\Gamma(x_s) \tag{25}$$

$\Gamma \cap \Omega_i$ is the part of the interface located in Ω_i. σ is the surface tension coefficient, κ represents the curvature which is obtained from volume of the fraction (i.e. κ=-∇.n). n is the interface normal vector. The surface tension force is acting on the interface between the two phases. Since the exact location and form of this force is unknown the F term can be converting to volume force function of the surface tension by applying Continuum Surface Force (CSF). In this model the surface curvature is formulated in terms of face normal at the interface. Finally the volumetric surface tension force can be formulated in terms of surface tension and jump pressure across the interface:

$$F = \sigma \kappa \frac{\rho}{0.5(\rho_1 + \rho_2)} \nabla \alpha \approx \sigma \kappa \nabla \alpha \tag{26}$$

Viscosity stress is obtained as:

$$\nabla.\tau = \nabla.\mu[\nabla u + (\nabla u)^T] = \nabla.(\mu \nabla u) + (\nabla u).\nabla.\mu \tag{27}$$

In *interFoam* a modified pressure P^* is adopted by excluding the hydrostatic pressure (ρg.x) from the pressure P. The gradient of this modified pressure is written as:

$$\nabla P^* = \nabla P - \nabla(\rho g.x) = \nabla P - \rho g - g.x \nabla P \tag{28}$$

The final form of the momentum equation can be obtained by taking to account the volumetric form of surface tension, viscous stress and modified pressure equations:

$$\frac{\partial \rho u}{\partial t} + \nabla.(\rho u u) - \nabla.(\mu \nabla u) = -\nabla P + (\nabla u).\nabla.\mu - g.x \nabla P + \sigma \kappa \nabla \alpha \tag{29}$$

3.3. Pressure-Velocity solution algorithm

A semi-discretized form of the momentum equation can be written as:

$$a_P U_P = H(u) - \nabla P^* - g.x\nabla\rho + \sigma\kappa\nabla\alpha \tag{30}$$

$H(u)$ operator has two parts: the first is the transport part which contains the matrix of coefficient for the neighboring cells multiplied by correspondent velocity. The second part is the source part, which includes the source terms except from the buoyancy and surface tension terms. The velocity U_P can be expressed at the cell center:

$$U_P = [a_P]^{-1}H(u) - \nabla P^* - g.x\nabla\rho + \sigma\kappa\nabla\alpha \tag{31}$$

By replacing the velocity in the continuity equation, the *Pressure Poisson Equation* in an undiscretized form can be written as:

$$\nabla.\left[[a_P]^{-1}\nabla P_f\right] = \nabla.\left[[a_P]^{-1}[H(u) - g.x\nabla\rho + \sigma\kappa\nabla\alpha]_f\right] \tag{32}$$

The solution of PIMPLE algorithm in the *interFoam* can be summarized as following steps:

- **Momentum Predictor**: first the momentum equation (30) is solved. However, since the exact pressure gradient is not known the pressure filed from the previous step is used.
- **Pressure Solution**: By using the velocities obtained in previous step, the solution of discretized form of the equation (32) will return the first estimation of the new pressure fields.
- **Velocity Correction:** The velocity can be corrected explicitly using equation (31) by applying the new pressure fields

3.4. Solver structure

The *interFoam* solver has the following directory structure. Again, here only relevant and governing files are presented which are *alphaEqn*, *UEqn* and *PEqn*.

```
├── Allwclean
├── Allwmake
├── alphaCourantNo.H
├── alphaEqn.H
├── alphaEqnSubCycle.H
├── correctPhi.H
├── createFields.H
├── interDyMFoam
├── interFoam.C
├── interFoam.dep
├── interMixingFoam
├── LTSInterFoam
├── Make
├── MRFInterFoam
├── pEqn.H
├── porousInterFoam
├── setDelta
```

3.4.1. alphaEqn.H

This *H* file computes the Phase fraction (*alpha*) which is obtained from continuity equation described earlier. In *alphaEqn.H* first the necessary fields are created which are required for the continuity equation. Creating flux due to interface compression *phic*.

```
surfaceScalarField phic = mag(phi/mesh.magSf());
phic = min(interface.cAlpha()*phic, max(phic));
```

C_Y serves as a user defined value to restrict interface smearing. However in this tutorial this value is set to 1.

Since the solver only relies on mixed property, a relative flux has to set up. Hence, compression flux term is multiplied by normalized normal vector of the interface.

```
surfaceScalarField phir("phir", phic*interface.nHatf());
```

Interfoam solves the continuity equation in this formulation:

$$\frac{\partial \alpha_1}{\partial t} + \boldsymbol{\nabla}.(\alpha_1 U) - \nabla.[\alpha_1(1-\alpha_1)U_{ra}] = 0 \tag{33}$$

In which the two first terms are due to the water phase and the third term is due to the gas phase. U_{ra} is the relative velocity. The delimiter α_M for the liquid fraction advection, explained earlier, is implemented in MULES solver (Multidimensional Universal Limiter with Explicit Solution) which is a Flux Corrected Transport explicit solver for hyperbolic equations. The solver considers upper and lower limits, suitable for α equation.

```
for (int aCorr=0; aCorr<nAlphaCorr; aCorr++)
    {
        surfaceScalarField phiAlpha
        (
            fvc::flux
            (
                phi,
                alpha1,
                alphaScheme
            )
          + fvc::flux
            (
                -fvc::flux(-phir, scalar(1) - alpha1, alpharScheme),
                alpha1,
                alpharScheme
            )
        );

        MULES::explicitSolve(alpha1, phi, phiAlpha, 1, 0);

        rhoPhi = phiAlpha*(rho1 - rho2) + phi*rho2;
```

d. In the last line the mass flux is corrected for the found α value. The "schemes" determine how to discretize the field

```
    word alphaScheme("div(phi,alpha)");
    word alpharScheme("div(phirb,alpha)");
```

3.4.2. UEqn.H

The equation for velocity distribution is set up in this *H* folder. Since the velocity and the pressure equation are coupled together, the velocity is not solved independently here. This *H* file is basically solves the equation (31) in two parts. First the temporal derivative, advection term and viscous stress term are solved and then the rest of the terms.

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

    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(rhoPhi, U)
      - fvm::laplacian(muEff, U)
      - (fvc::grad(U) & fvc::grad(muEff))
    //- fvc::div(muEff*(fvc::interpolate(dev(fvc::grad(U))) & mesh.Sf()))
    );
    mrfZones.addCoriolis(rho, UEqn);

    UEqn.relax();

    if (momentumPredictor)
    {
        solve
        (
            UEqn
         ==
            fvc::reconstruct
            (
                (
                    fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
                  - ghf*fvc::snGrad(rho)
                  - fvc::snGrad(p_rgh)
                ) * mesh.magSf()
            )
        );
    }
```

3.4.3. PEqn.H

After setting up the velocity equation, the solver goes to the pressure solution step in *PEqn.H*. First the velocity is predicted as it is required to set up the pressure equation. *UEqn.H* returns the central coefficient, this is inverted and retrieved. The coefficients are interpolated into faces values and finally the velocity field is predicted by multiplying the reciprocal of the central coefficients with the operation source from the same matrix.

```
volScalarField rAU("rAU", 1.0/UEqn.A());
    surfaceScalarField rAUf("Dp", fvc::interpolate(rAU));

    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn.H();
```

The right hand side terms in equation (32) is assembled in two steps. The first is the flux with only velocity dependency, created using the predicted velocity

```
   surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (fvc::interpolate(HbyA) & mesh.Sf())
      + fvc::ddtPhiCorr(rAU, rho, U, phi)
    );
```

The *ddtPhiCorr* incorporates a correction for the divergence of the face velocity field by taking out the difference between the interpolated velocity and the real flux.

Considering flux due to velocity as well as gravity and surface tension, the complete flux is constructed

```
    adjustPhi(phiHbyA, U, p_rgh);
    phi = phiHbyA;

    surfaceScalarField phig
    (
        (
            fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
          - ghf*fvc::snGrad(rho)
        )*rAUf*mesh.magSf()
    );

    phiHbyA += phig;
```

In which *sigmaK* accounts for interfacial forces.

From the step 2 of the PIMPLE algorithm described previously, the pressure equation is defined according to equation (29). A non-orthogonal correction is made using the discretized form of the equation (32) in a loop. The correction is made to the velocity field using equation (30). This correction is also includes the boundaries. At the end the total pressure is calculated and the modified pressure *P_rgh* in equation (32) is defined.

```
while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phiHbyA)
        );

        p_rghEqn.setReference(pRefCell, getRefCellValue(p_rgh, pRefCell));

        p_rghEqn.solve(mesh.solver(p_rgh.select(pimple.finalInnerIter())));
```

```
        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA - p_rghEqn.flux();

            U = HbyA + rAU*fvc::reconstruct((phig - p_rghEqn.flux())/rAUf);
            U.correctBoundaryConditions();
            fvOptions.correct(U);
        }
    }

    #include "continuityErrs.H"

    p == p_rgh + rho*gh;

    if (p_rgh.needReference())
    {
        p += dimensionedScalar
        (
            "p",
            p.dimensions(),
            pRefValue - getRefCellValue(p, pRefCell)
        );
        p_rgh = p - rho*gh;
}
```

3.5 Set up the case

The considered case is taken from the *waterChannel* tutorial in the following directory tutorials/multiphase/ interFoam/ras/waterChannel.

3.5.1. Mesh Generation

The same O grid method as for *settlingFoam* case is applied here. However unlike the previous case, different initial condition is required which is non-uniform. This can be done by using the *setField* utility which requires *SetFieldDict* dictionary located in the system directory. The *setFieldDIct* file is created with following lines

```
defaultFieldValues
(
        volScalarFieldValue alpha1 0
);

regions
(
    boxToCell
    {
           box (-0.13 0.0643 -2) (0.13 0.13 0)
        fieldValues
        (
           volScalarFieldValue alpha1 1
        );
    }
);
```

These lines define that the *alpha* value is *0* everywhere except the specified region. The box encompasses the area where the water exists. This is depicted in the figure 7. However, defined box is not well matched with the patches. The atmosphere patch, which is defined later on, is only covering part of the area which is defined for the gas phase. Alternatively, one can try to use *funkySetFiels* to improve the initial condition.

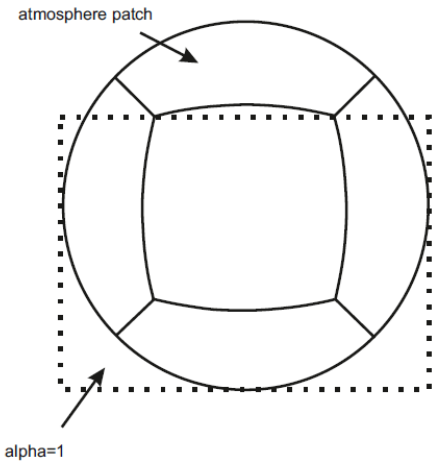

Figure 7. The box illustration on the pipe cross section for *setFieldDict* dictionary

In the *blockMeshDict* the following changes is done for the patches

```
// Defining patches
patches
(
   patch inlet
   (
        (1 5 4 0)
        (3 7 6 2)
        (0 4 7 3)
        (3 2 1 0)
   )
   patch outlet
   (
        (13 9 8 12)
        (15 11 10 14)
        (12 8 11 15)
        (8 9 10 11)
   )
   wall walls
   (
        (4 5 13 12)
        (6 7 15 14)
        (7 4 12 15)
   )
   patch atmosphere
   (
        (5 6 14 13)
        (2 6 5 1)
        (14 10 9 13)
   )
);
```

3.5.2. Initial and boundary condition

In this case there should be one more patch as for the open surface. For the pressure, *P_rgh* in *0* folder there would be:

```
dimensions [1 -1 -2 0 0 0 0];
internalField uniform 0;
boundaryField
{
inlet
{
        type zeroGradient;
}
outlet
{
        type fixedValue;
        value uniform 0;
}
walls
```

```
{
        type zeroGradient;
}
        atmosphere
{
        type totalPressure;
        p0 uniform 0;
        U U;
        phi phi;
        rho rho;
        psi none;
        gamma 1;
}
}
```

The same boundary and initial condition for U is considered as previous case. All other boundaries are kept as original and just by altering according to the patches.

3.5.3 Running the Case

The velocity is chosen to be 2 m/s. The result after running the case under above mentioned conditions is illustrated in Figure 8. It shows the velocity profile on the free surface of the fluid interacting with air.

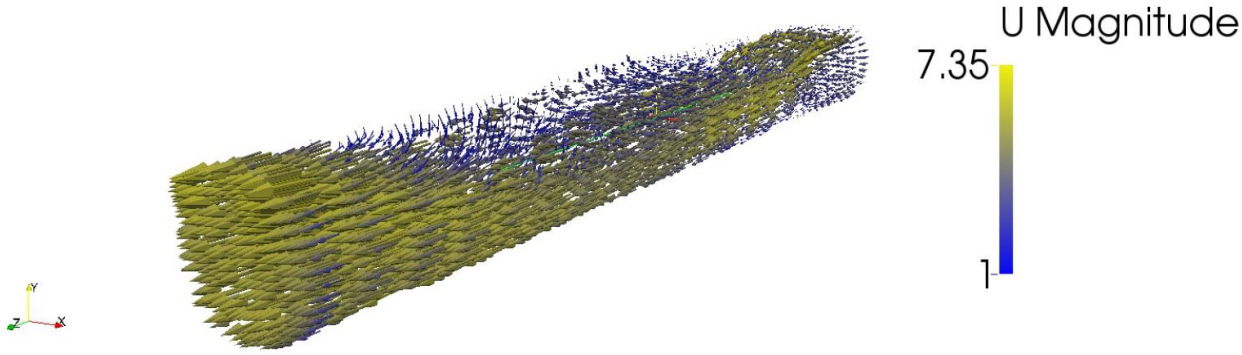

Figure 8. Velocity profile in an open surface of the fluid in a pipe

4. interSettlingFoam

In this section, two solvers, *interFoam* and *settlingFoam* are combined together with the purpose to describe the velocity distribution at the surface and the settling velocity in the pipe. The *interFoam* solver is considered as the base solver and the drift equation (*alphaEqn*) in *settlingFoam* is added to the velocity-pressure equation in PIMPLE loop. The reason is that the settling is mainly governed by the drift equation. However the assumption is that settling has no effect on the flow field.

4.1. creatField.H

Since the header file *creatField.H* is called before the solution loop, by adding new equation, the information related to what new variables will be solved needs to be modified. First of all, the *creatFileds.H* file in *interFoam* needs to be changed by adding necessary transport properties i.e *rhod*, *muc*, *muMax* from *settlingFoam*. To reflect this change, the dictionary file has to be edited as well. This is done by adding following lines to the *creatFieldH*.

```
Info<< "Reading field AlphaS\n" << endl;
   volScalarField AlphaS
   (
        IOobject
        (
                "AlphaS",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
        ),
        mesh
    );
 Info<< "Reading transportProperties\n" << endl;

   IOdictionary transportProperties
   (
        IOobject
        (
                "transportProperties",
                runTime.constant(),
                mesh,
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE
        )
    );

    dimensionedScalar rhod(transportProperties.lookup("rhod"));
    dimensionedScalar muc(transportProperties.lookup("muc"));
    dimensionedScalar muMax(transportProperties.lookup("muMax"));

        volScalarField rhoS
    (
        IOobject
        (
                "rhoS",
```

```
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
        ),
        rho1/(scalar(1) + (rho1/rhod - 1.0)*AlphaS)
    );
        volScalarField alphaS
    (
        IOobject
        (
                "alphaS",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
        ),
        rho*AlphaS/rhod
     );

Info<< "Initialising field Vdj\n" << endl;
    volVectorField Vdj
    (
        IOobject
        (
                "Vdj",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedVector("0.0", U.dimensions(), vector::zero),
        U.boundaryField().types()
    );
        Info<< "Selecting Drift-Flux model " << endl;
  const word VdjModel(transportProperties.lookup("VdjModel"));
  Info<< tab << VdjModel << " selected\n" << endl;
  const dictionary& VdjModelCoeffs
 (
        transportProperties.subDict(VdjModel + "Coeffs")
  );
  dimensionedVector V0(VdjModelCoeffs.lookup("V0"));
  dimensionedScalar a(VdjModelCoeffs.lookup("a"));
  dimensionedScalar a1(VdjModelCoeffs.lookup("a1"));
  dimensionedScalar alphaMin(VdjModelCoeffs.lookup("alphaMin"));
```

4.2. interFoam.C

The next step is to add a new equation for '*alpha*' (it is important to rename this *alpha* to e.p *alphaS*, not be confused with *alpha* already defined in *interFoam* as liquid-air phase fraction). This is done in *interFoam.C* (better to call it *interSettlingFoam.C*). As mentioned previously, the *alphaS* equation only reads the velocity and does not play an active role in PIMPLE loop.

```
        Info<< "Time = " << runTime.timeName() << nl << endl;
        twoPhaseProperties.correct();
        #include "alphaEqnSubCycle.H"
        interface.correct();

// adding new line
        #include "alphaSEqn.H"
// done adding the new line

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
                #include "UEqn.H"
                // --- Pressure corrector loop
                while (pimple.correct())
                {
                   #include "pEqn.H"
                }
                if (pimple.turbCorr())
                {
                   turbulence->correct();
                }
        }
```

The next is to add *alphaSEqn.H* to the main directory. This file is the same as *alphaEqn.H* explained earlier but with renaming the same parameters e.p *rhoC* change to *rho1*, *Alpha* to *AlphaS* etc.

4.3. transportProperties

As for the new tutorial (working on *waterChannel* as the base case), the *transportProperties* dictionary needs to be edited by adding the following lines. These are the new parameters which their values need to be specified.

```
muMax              muMax [ 1 -1 -1 0 0 0 0 ] 10.0;
muc                muc [ 1 -1 -1 0 0 0 0 ] 0.00178;
rhod               rhod [ 1 -3 0 0 0 0 0 ] 1996;
VdjModel           simple;
simpleCoeffs
{
        V0         V0 [0 1 -1 0 0 0 0 ] ( 0 -0.002198 0 );
        a          a [ 0 0 0 0 0 0 0 ] 285.84;
        a1         a1 [ 0 0 0 0 0 0 0 ] 0;
        alphaMin   alphaMin [ 0 0 0 0 0 0 0 ] 0;
}
generalCoeffs
{
        V0         V0 [ 0 1 -1 0 0 0 0 ] ( 0 -0.0018 0 );
        a          a [ 0 0 0 0 0 0 0 ] 1e-05;
        a1         a1 [ 0 0 0 0 0 0 0 ] 0.1;
        alphaMin   alphaMin [ 0 0 0 0 0 0 0 ] 2e-05;
}
```

4.4. FvSchemes and fvSolution

Choosing the type of discretization scheme to apply to the equation is decided in *fvSchemes*. New terms need to be added here with regards to what was added in the solver source code. Here the new terms are formulated in divergence formulation and following lines need to be added.

```
divSchemes
{
  …
div(phiAlphaS,AlphaS) Gauss upwind; //
div(phiVdj,Vdj) Gauss linear;//
  …
}
```

fvSolution is edited as follows. This includes the information about the *alpha* solver.

```
AlphaS
  {
        solver          BICCG;
        preconditioner  DILU;
        tolerance       1e-7;
        relTol          0;
  }
AlphaSFinal
  {
        solver          BICCG;
        preconditioner  DILU;
        tolerance       1e-7;
        relTol          0;
   }
relaxationFactors
{
        fields
        {
        }
        equations
        {
        "AlphaS.*" 0.2;
        }
}
```

4.5. Initial and boundary condition

In *0* folder where initial and boundary conditions are defined, new file for *AlphaS* should be included. This would be the same as before in *settlingFoam* section.

```
dimensions      [0 0 0 0 0 0 0];
internalField   uniform 0;
boundaryField
{
        inlet
        {
                type fixedValue;
```

```
                value uniform 0.01;
        }
        walls
        {
           type zeroGradient;
        }
        outlet
        {
           type zeroGradient;
        }
        atmosphere
        {
           type inletOutlet;
           inletValue uniform 0;
           value uniform 0;
        }
}
```

4.6 Running the case

After all the steps above are accomplished, the Mesh created for the *interFoam* case can be inserted directly in to the *blockMesh*. It has to be noted that *wmake* should run after each alteration in the solver. Following lines also needs to be edited in the "files" inside the Make directory of the solver. This allows you to call the solver *interSettlingFoa*.

```
interSettlingFoam.C
EXE = $(FOAM_USER_APPBIN)/interSettlingFoam
```

Following control variables are considered for the solver:

Table 5. control variables for *interSettlingFoam*

		U	P_rgh	AlphaS
dimensions		[0 1 -1 0 0 0 0]	[1 -1 -2 0 0 0 0]	[0 0 0 0 0 0 0]
Internal field		Uniform (0 0 0.01)	Uniform (0 0 0)	Uniform 0
inlet	type	fixedValue	zeroGradient	fixedValue
	value	uniform (0 0 0.01)		uniform 0.1
outlet	type	zeroGradient	fixedValue	zeroGradient
			uniform 0	
wall	type	fixedValue	zeroGradient	zeroGradient
	value	uniform (0 0 0)		
atmosphere	type	pressureInletOutletVelocity	TotalPressure	inletOutlet

	Value	Value uniform (0 0 0)	P0 uniform 0 U U phi phi rho rho Psi none Gamma 1	inletValue uniform 0 Value uniform 0

Control variable	**value**
Maximum Corrant number	0.5
Delta T	0.05
Maximum delta T	1
endTime	200

Figure 9. illustrates the final result. This shows the velocity profile while the settling of the dispersed phase is also considered.

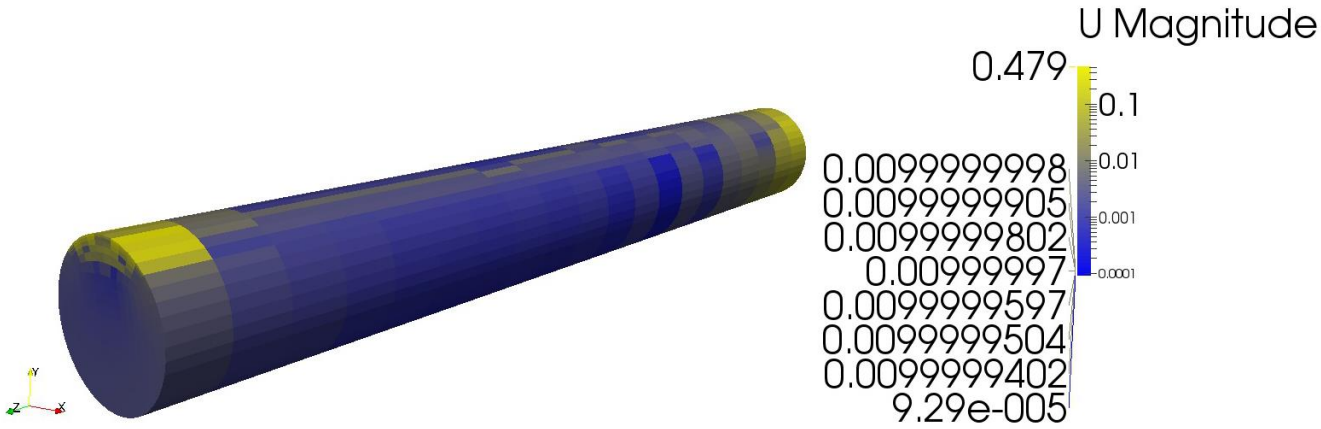

Figure 9. velocity profile result solved by *interSettlingFoam* solver

References

Henrik Rusche, 2002, Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions. PhD Thesis, Imperial College of Science, Technology & Medicine, Department of Mechanical Engineering. Roman

Thiele, 2010, Modeling of Direct Contact Condensation With OpenFOAM, Division of Nuclear Reactor Technology, Royal Institute of Technology, Stockholm, Sweden.

Pedro Miguel Borges Lopes, 2013, Free-surface flow interface and air-entrainment modelling using OpenFOAM, Thesis Project in Hydraulic, PhD thesis, Water Resources and Environment, Universidade de Coimbra .

Suraj S Deshpande, Lakshman Anumolu and Mario F Trujillo, 2012, Evaluating the performance of the two-phase flow solver interFoam, Computational Science & Discovery 5 (2012) 014016 (36pp).

Daniel Brennan 2001, The Numerical Simulation of Two-Phase Flows in Settling Tanks, PhD Thesis, University of London.