Solid and Fluid Mechanics
Chalmers University of Technology

CFD with OpenSource software, assignment 3

Tutorial multiphaseInterFoam

For the damBreak4phase case

Written for OpenFOAM-1.7.x

Author:
Patrik Andersson

Peer reviewed by:
Jelena Andric
Anders Rynell

November 5, 2010

Contents

1 Tutorial multiphaseInterFoam **2**
1.1 Introduction 2
1.2 Setup 3
1.2.1 Getting started 3
1.2.2 Boundary and initial conditions 3
1.2.3 Solver setup 6
1.3 The solver 9
1.3.1 multiphaseMixture 9
1.3.2 UEqn.H, pEqn.H and multiphaseInterFoam.C 16
1.4 Running the case 18
1.5 Post-processing 19

Chapter 1

Tutorial multiphaseInterFoam

1.1 Introduction

This tutorial describes how to setup, run and post-process a case involving severeal incompressible phases. It also describes the solver used in detail. The *multiphaseInterFoam* (mpIF) case is setup as a simple 2D-block (with the standard one cell depth in z-direction), but with a small obstacle located at the bottom. The geometry consists of four blocks filled with newtonian phases of: water, oil, mercury and air (see Figure 1.1). All phases are initially located behind membranes which are then removed simultaniously at `t=0` and the fluids collapse onto each other and the obstacle. This creates a complicated mixture of the four different phases where the interaction between the phases need to be interpolated and the contact angles calculated. The regular *interFoam* solver algorithm is based upon the volume of fluid method (VOF). VOF contains a specie transport equation which is used to determine the relative volume fraction of the phases or the phase fraction α_n (interFoam can only handle two phases while mpIF can handle `n` number of phases). mpIF calculates a multiphase mixture via the multiphaseMixture script, which is derived from transportModel. The solver then calculates the physical properties of the phase as weighted averages based on α_n. The phase fraction can take any value between 0 and 1. Therefore, the interface between the phases is never sharply defined. Still, the interface occupies the volume between phases.

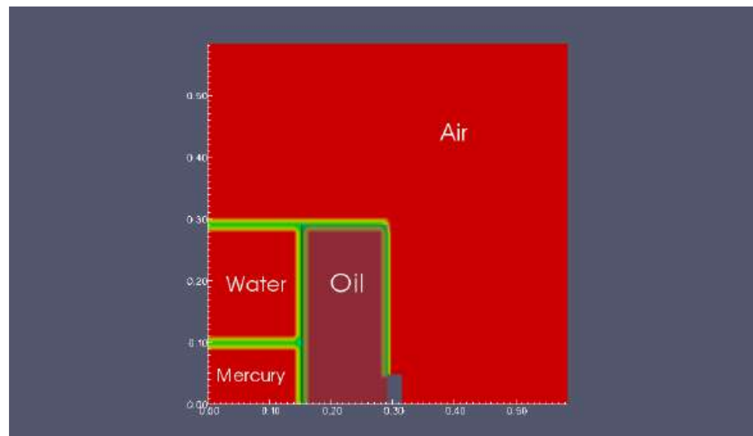

Figure 1.1: Geometry for the damBreak4phase tutorial case.

1.2 Setup

This section will cover the steps needed to be taken before running the case. We need to copy it from the tutorial folder, create the mesh, set the fields and then move on to running the mpIF-solver. Furthermore this section will cover the case files, and how values that are input are used in the solver.

1.2.1 Getting started

Copy the multiphaseInterFoam tutorial to the run directory (note that the placement of the case might differ between versions).

```
cp -r  $FOAM_TUTORIALS/multiphase/multiphaseInterFoam/laminar/\
damBreak4phase $FOAM_RUN
cd $FOAM_RUN/damBreak4phase
```

The copied folder contains:

`/0` – *alphaair alphamercury alphaoil alphas alphawater p_rgh U*

`/0.org` – *backup of original files listed above*

`/constant` – *g motionProperties transportProperties turbulenceProperties*

subfolder: `/polyMesh` – *blockMeshDict*

`/system` – *controlDict decomposeParDict fvSchemes fvSolution setFieldsDict*

1.2.2 Boundary and initial conditions

Located in `/0` are all the intial conditions for the different phases and a file called *phases*, which combines all of the phases so that they can be easier shown at the same time when post-processing in paraFoam. For example we can take a look at the reference phase air in *alphaair*.

```
leftWall
{
    type            alphaContactAngle;
    thetaProperties
    (
        ( water air ) 90 0 0 0
        ( oil air ) 90 0 0 0
        ( mercury air ) 90 0 0 0
        ( water oil ) 90 0 0 0
        ( water mercury ) 90 0 0 0
        ( oil mercury ) 90 0 0 0
    );
   value           uniform 0;
}
```

The static contact angle is set to 90 degrees for all the combinations of mixture, at the left wall. The same goes for the other walls. The top boundary is open and set as an atmosphere and has no set contact angle since the fluids should never come in contact with this region. Setting the static contact angle to 90 degrees is a way of avoiding to use the surface tension force between the wall and the fluid. This means that, if $\hat{n}$ denotes the normal to the interface at the wall as:

$$\hat{n} = n_w \cos(\theta_{eq}) - n_t \sin(\theta_{eq}) \tag{1.1}$$

θ_{eq} is the static contact angle set to 90 degrees, n_w the unit normal vector to the wall pointing towards the wall and n_t the unit vector tangential to the wall pointing towards the liquid. Then the interface of the liquid is in fact normal to the wall. If θ_{eq} would be less than 90 degrees then this indicates that the fluid wets the wall [1, p.17].

The folder *constant* contains, as described in section 1.2.1, the subfolder *polyMesh* with the dictionary-file, *blockMeshDict*, and other files needed for blockMesh. The meshing will not be discussed in this tutorial since it does not differ from what have already been covered in the *interFoam* - damBreak-tutorial. Furthermore, *constant* contains the initial conditions for `g` (the gravitation). This is then read by the solver when creating the fields in the latter part of *createFields.H* (which can be viewed by typing: `gedit $FOAM_APP/solvers/multiphase/multiphaseInterFoam/createFields.H`):

```
#include "readGravitationalAcceleration.H"

/*
dimensionedVector g0(g);

// Read the data file and initialise the interpolation table
interpolationTable<vector> timeSeriesAcceleration
(
    runTime.path()/runTime.caseConstant()/"acceleration.dat"
);
*/

Info<< "Calculating field g.h\n" << endl;
volScalarField gh("gh", g & mesh.C());
surfaceScalarField ghf("ghf", g & mesh.Cf());
```

Prior to this, *createFields.H*, has read and set fields for `p_rgh`, `U`, `rho` (ρ), and `phi` (ϕ), which is the relative face-flux field. These fields need to be set, so that the solver solve for them, and weight them with the cell value.

Properties

Furthermore, *constant* contains the three property-files *motion*-, *transport*- and *turbulenceProperties*. *motionProperties* describes what type of motion the mesh should have. In this tutorial the mesh has no movement so staticFvMesh is used. In *transportProperties* the dynamic laminar viscosity, `mu` (μ), and the density, `rho`, are set for the different phases. Since they are all newtonian fluids, we set the transport model for them to be Newtonian.

```
phases
(
     water
     {
         transportModel Newtonian;
         nu nu [ 0 2 -1 0 0 0 0 ] 1e-06;
         rho rho [ 1 -3 0 0 0 0 0 ] 1000;
     }

     oil
     {
         transportModel Newtonian;
         nu nu [ 0 2 -1 0 0 0 0 ] 1e-06;
         rho rho [ 1 -3 0 0 0 0 0 ] 500;
```

```
    }

    mercury
    {
        transportModel Newtonian;
        nu nu [ 0 2 -1 0 0 0 0 ] 1.125e-07;
        rho rho [ 1 -3 0 0 0 0 0 ] 13529;
    }

    air
    {
        transportModel Newtonian;
        nu nu [ 0 2 -1 0 0 0 0 ] 1.48e-05;
        rho rho [ 1 -3 0 0 0 0 0 ] 1;
    }
);
```

To be able to solve the multiphase problem a reference phase is needed. Here, the air has been chosen, and is derived from the other phases such that they sum up to one. This due to the multi phase method, volume of fraction, being used (previously mentioned in Section 1.1).

```
refPhase        air;
```

transportProperties is also used for setting the surface tension for the fluids. It is set to be the same for all the phases, $\sigma = 0.07N/m$

```
sigmas
(
    (air water) 0.07
    (air oil) 0.07
    (air mercury) 0.07
    (water oil) 0.07
    (water mercury) 0.07
    (oil mercury) 0.07
);
```

In *turbulenceProperties* we set what kind of flow that is going to be solved for. In this case its sufficent to solve for laminar flow. Though, we have the option to choose between three different models (see Table 1.1 below). Both *LESModel* and *RASModel* consist of several subcategories of models. RAS

Simulation type	
LESModel	Large eddie simulation
RASModel	Reynolds average stress
laminar	Plain laminar flow

Table 1.1: Possible simulation types

is split into two categories, those for incompressible and those for compressible. LES is divided into: "Isochoric LES turbulence models" and "Anisochoric LES turbulence models" (Isochoric = constant volume). If one is to use any of these models we would need to, as previously mentioned, specify them in *turbulenceProperties*, but also create a file in *constant*. For example if we would like to use *LESModel*, then create the file *LESProperties* and in this file specify which LES model that is to be used, followed by specification for all the needed variables for that specific model.

1.2.3 Solver setup

This section will cover the needed setup conditions for our *multiphaseInterFoam*-solver. In *system* it is specified where the fields of different phases are to be set, also specify what kind of discretisation schemes that are to be used in the solver. In *setFieldsDict* we see the positioning of the phases.

```
defaultFieldValues
(
    volScalarFieldValue alphaair 1
    volScalarFieldValue alphawater 0
    volScalarFieldValue alphaoil 0
    volScalarFieldValue alphamercury 0
    volVectorFieldValue U ( 0 0 0 )
);

regions
(
    boxToCell
    {
        box ( 0 0 -1 ) ( 0.1461 0.292 1 );
        fieldValues
        (
            volScalarFieldValue alphawater 1
            volScalarFieldValue alphaoil 0
            volScalarFieldValue alphamercury 0
            volScalarFieldValue alphaair 0
        );
    }
    boxToCell
    {
        box ( 0.1461 0 -1 ) ( 0.2922 0.292 1 );
        fieldValues
        (
            volScalarFieldValue alphawater 0
            volScalarFieldValue alphaoil 1
            volScalarFieldValue alphamercury 0
            volScalarFieldValue alphaair 0
        );
    }
    boxToCell
    {
        box ( 0 0 -1 ) ( 0.1461 0.1 1 );
        fieldValues
        (
            volScalarFieldValue alphawater 0
            volScalarFieldValue alphaoil 0
            volScalarFieldValue alphamercury 1
            volScalarFieldValue alphaair 0
        );
    }
);
```

The reference phase, air, is set to occupy the whole domain, whereas the other phases occupy the lower left corner, as seen in Figure 1.1. In those boxes, air is set to zero. The new phase is then given the cell value one so that it is now the only fluid in those cells. In *fvSchemes* we set so that we use Euler scheme (first order, bounded, implicit) as default for our descretization in time.

```
ddtSchemes
{
    default         Euler;
}
```

For gradSchemes (the gradient ∇) Gauss linear scheme is set as default and to be used for `U` and `gamma`.

```
gradSchemes
{
    default         Gauss linear;
    grad(U)         Gauss linear;
    grad(gamma)     Gauss linear;
}
```

For the divergence schemes there is no default setting, instead we specify for each variable which scheme needs to be used. For the divergence of `rho*phi` and `U` a Gauss Upwind scheme (upwind differencing) is used. For the `div($\phi$,$\alpha$)` Gauss vanLeer scheme is used and for `div($\phi_{rb}$,$\alpha$)` we use Gauss interfaceCompression.

```
divSchemes
{
    div(rho*phi,U)  Gauss upwind;
    div(phi,alpha)  Gauss vanLeer;
    div(phirb,alpha) Gauss interfaceCompression;
}
```

For the laplacian (∇^2) Gauss linear corrected scheme is used, where "corrected" means that it is numerically unbounded, second order and conservative.

```
laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}
```

Fields for which the flux is needed, that is `pcorr`, `p_rgh` and `alpha` for the different phases.

```
fluxRequired
{
    default         no;
    pcorr;
    p_rgh;
    "alpha.*";
```

In *fvSolution* where the equation solvers, algorithms and tolerances are set, we have the following setup:

```
solvers
{
    pcorr
    {
        solver          PCG;
        preconditioner
        {
            preconditioner  GAMG;
            tolerance       1e-05;
            relTol          0;
            smoother        GaussSeidel;
            nPreSweeps      0;
            nPostSweeps     2;
            nBottomSweeps   2;
            cacheAgglomeration off;
            nCellsInCoarsestLevel 10;
            agglomerator    faceAreaPair;
            mergeLevels     2;
        }
        tolerance       1e-05;
        relTol          0;
        maxIter         100;
    }

    p_rgh
    {
        solver          GAMG;
        tolerance       1e-07;
        relTol          0.05;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        nFinestSweeps   2;
        cacheAgglomeration on;
        nCellsInCoarsestLevel 10;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }

    p_rghFinal
    {
        solver          PCG;
        preconditioner
        {
            preconditioner  GAMG;
            tolerance       1e-07;
            relTol          0;
            nVcycles        2;
            smoother        GaussSeidel;
            nPreSweeps      0;
            nPostSweeps     2;
            nFinestSweeps   2;
            cacheAgglomeration on;
            nCellsInCoarsestLevel 10;
```

```
            agglomerator    faceAreaPair;
            mergeLevels     1;
        }
        tolerance       1e-07;
        relTol          0;
        maxIter         20;
    }

    "(U|alpha)"
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        tolerance       1e-08;
        relTol          0;
        nSweeps         1;
    }
}

PISO
{
    nCorrectors     4;
    nNonOrthogonalCorrectors 0;
    nAlphaCorr      4;
    nAlphaSubCycles 4;
    cycleAlpha      yes;
    cAlpha          2;
}

relaxationFactors
{
    U 1;
}
```

The *GAMG*-solver is an abbrevation for "Generalised geometric-algebraic multi-grid". This is a fast method since it, in rough terms, first solves for the user-specified coarse grid and then refines it in stages (more info availiable in the OpenFoam-userguide [2]). It is in this case used as solver for `p_rgh`. It is also used as a preconditioner for `p_corr` and `p_rghFinal`. `p_corr` and `p_rghFinal` both uses *PCG* as solver method (where *PCG* is a linear solver for symmetric matrices). `U|alpha`, which is the velocity field for the diferrent faces, is solved by using *smoothSolver*, where *GaussSeidel* is chosen as a smoother. For further explanation of the solver settings please read Chapter 4.5 "*Solution and algorithm control*" in the above mentioned userguide.

1.3 The solver

This section will cover a more detailed of the solver code for multiphaseInterFoam. We will now take a closer look at some of the files relevant for the actual solver: *multiphaseInterFoam.C*, *multiphaseMixture.H*, *multiphaseMixture.C*, *UEqn.H*, *PEqn.C*, *phases.C*, and *phases.H*

1.3.1 multiphaseMixture

Now it makes sense to start with the multiphaseMixture files, which is where the weighting by alpha and the contact-angle comes into play. The description in the *multiphaseMixture.H* states that: "*Incompressible multi-phase mixture with built in solution for the phase fractions with interface compression for interface-capturing. Derived from transportModel it can be used in conjunction with the*

incompressible turbulence models. Surface tension and contact-angle is handled for the interface between each phase-pair."
As mentioned in Section 1.2.2 the user is free to choose turbulence model which the description here clearly states. The surface tension and contact-angle will be discussed and explained further down in this section.

The phases are first handled in the files *phase.C* and *phase.H* located in the `multiphaseMixture` folder. Here the description states that: *Single incompressible phase derived from the phase-fraction. Used as part of the multiPhaseMixture for interface-capturing multi-phase simulations*

In other words, one phase is read at the time, and its viscosity-model for `nu` (ν the kinematic laminar viscosity) and `rho`, ρ. This was previously discussed when reviewing the case-folders. Also, the code is here define the phase its initial volVectorField for `U` and surfaceScalarField for `phi`. In *multiphaseMixture.H* the code intitially sets the selected transport model and then moves forward by defining interfacePairs for the mixture of phases. This can be seen in the Constructor and the Friend Operator part of the code, listed below.
(`gedit $FOAM_SOLVERS/multiphase/multiphaseInterFoam/multiphaseMixture/multiphaseMixture.H`)

```
// Constructors
            interfacePair()
            {}

            interfacePair(const word& alpha1Name, const word& alpha2Name)
            :
                Pair<word>(alpha1Name, alpha2Name)
            {}

            interfacePair(const phase& alpha1, const phase& alpha2)
            :
                Pair<word>(alpha1.name(), alpha2.name())
            {}

        // Friend Operators

            friend bool operator==
            (
                const interfacePair& a,
                const interfacePair& b
            )
            {
                return
                (
                    ((a.first() == b.first()) && (a.second() == b.second()))
                 || ((a.first() == b.second()) && (a.second() == b.first()))
                );
            }
            friend bool operator!=
            (
                const interfacePair& a,
                const interfacePair& b
            )
            {
                return (!(a == b));
            }
    };
```

In the "Friend Operators"-part it is seen that different combinations/scenarios of interface-pairs are written. In the first case the two pairs, a and b, are the same. Then after the or (||) statement we see that the pairing is switched, but still containing the same pair of alphas. In the last scenario they are not equal. Furthermore, the *multiphaseMixture.H* code defines member data and corresponding member functions such as those seen in Table 1.2 below.

Private data terms	
refPhase	The phase chosen as reference
rhoPhi	The volumetric flux
sigmatable, σ	The stresses for the interface pair
deltaN	Stabilisation for the normalisation of the interface normal
alphaTable, α	Phase-fraction field table for multivariate discretization from multivariateSurfaceInterpolationScheme
Member functions	**Returns the:**
phases	phases
U	velocity
phi, ϕ, *rhophi*, $\rho\phi$	volumetric flux
rho, ρ	mixture density
mu, μ	dynamic laminar viscosity
muf, μ_f	face-interpolated dynamic laminar viscosity
nu, ν	kinematic laminar viscosity
nuf, ν_f	face-interpolated kinematic laminar viscosity
nearInterface	Indicator of the proximity of the interface-field, values are 1 near and 0 away from the interface
solve	Solve for the mixture phase-fractions
correct	Correct the mixture properties
read	Read base transportProperties dictionary

Table 1.2: multiphaseMixture.H

multiphaseMixture.C

To view type:
`gedit $FOAM_SOLVERS/multiphase/multiphaseInterFoam/multiphaseMixture/multiphaseMixture.C`
In the solve part of *multiphaseMixture.C* we have the iteration loops for ρ, μ, μ_f.There are also loops for ν, which is $\frac{\mu}{\rho}$, and one loop for the faceinterpolated ν_f. The loop for the *surfacetensionforce* (refered to as `stf` in the code) is a bit more extensive, here σ for the *interfacepair* and an interpolation is performed between the two phases. The surface tension force is defined as:

$$F_s = \sigma \left(\nabla \cdot \left(\frac{\nabla \alpha}{|\nabla \alpha|} \right) \right) (\nabla \alpha) \tag{1.2}$$

where

$$\begin{aligned} \nabla \alpha &= n\text{, the vector normal to the interface} \\ \sigma &= \text{surface tension coefficient} \end{aligned}$$

In the code this is represented by:

```
stf += dimensionedScalar("sigma", dimSigma_, sigma())
   *fvc::interpolate(K(alpha1, alpha2))*
    (
        fvc::interpolate(alpha2)*fvc::snGrad(alpha1)
```

```
        - fvc::interpolate(alpha1)*fvc::snGrad(alpha2)
    );
```

where the curvature:

$$\texttt{K(alpha1,alpha2)} = -\nabla \cdot \left(\frac{\nabla \alpha}{|\nabla \alpha|} \right) \tag{1.3}$$

and sigma:

$$\texttt{dimensionedScalar("sigma", dimSigma_, sigma())} = \sigma \tag{1.4}$$

and the last part within the paranthesis represent $\nabla \alpha$.
Moving on we have the Piso-loops for alpha in the code (see Table 1.3 below). Compared to the two-phase case for *interFoam* we now have a different setup. For example, there are four subcycles for each alpha, this gives that alpha is solved in one fourth length of the time-step.

Piso-loops	fvSolution settings	Description
nAlphaSubCycles	4	Number of subcycles for α_n for each timestep
nAlphaCorr	4	Number of correction for α, to improve quality of solution via fixed point iteration
cycleAlpha	yes	Cycling of alpha turned on
cAlpha	2	Compression of the interface, above one equals to enhanced compression of the

Table 1.3: Piso-loops in *multiphaseMixture.C*

Contact angle

A large and interesting section in *multiphaseMixture.C* is the correction for the boundary condition. Done on the unit normal `nHat` on walls, in order to produce a correct contact angle here. The dynamic contact angle is calculated by using the component of the velocity U on the direction of the interface, parallell to the wall. Before going in to this we need to take a quick look at the definitions of the angles in *alphaContactAngleFvPatchScalarField.C*.

class interfaceThetaProps		
theta0	θ_0 (θ_C)	Equilibrium contact angle
uTheta	u_θ	Dynamic contact angle velocity scale
thetaA	θ_A	Limiting advancing contact angle
thetaR	θ_R	Limiting receeding contact angle

Table 1.4: Angles

In Figure 1.2 [3] above, we see the contact angle of a droplet which will represent our phase. γ_{SL} denotes the solid liquid energy, γ_{SG} the solid vapor energy and γ_{LG} the liquid vapor energy, i.e. the surface tension. This is governed by Youngs equation (1.5) which will be satisfied at equilibrium.

$$0 = \gamma_{SG} - \gamma_{SL} - \gamma_{LG} \cos(\theta_C) \tag{1.5}$$

The angle θ_C is dependent on the highest (advancing) contact angle θ_A and the lowest (receding) contact angle θ_R, written as:

$$\theta_C = \arccos \left(\frac{r_A \cos(\theta_A) + r_R \cos(\theta_R)}{r_A + r_R} \right) \tag{1.6}$$

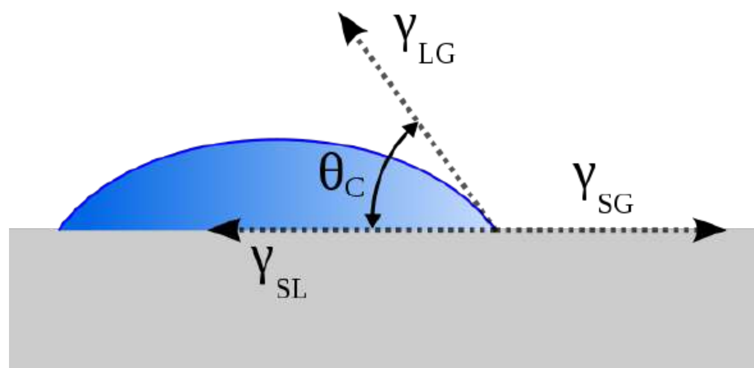

Figure 1.2: Contact angle and interface energies

where

$$r_A = \sqrt[3]{\frac{\sin^3(\theta_A)}{2 - 3\cos(\theta_A) + \cos(\theta_A)}} \tag{1.7}$$

$$r_R = \sqrt[3]{\frac{\sin^3(\theta_R)}{2 - 3\cos(\theta_R) + \cos(\theta_R)}} \tag{1.8}$$

θ_A is the contact angle when increasing the volume of, for example, the droplet. θ_R is the contact angle when decreasing the volume. In other words, when there is a relative motion of the droplet over a solid surface or another phase-interface, a different angle than the equilibrium contact angle will appear. It depends on the direction of the previous motion, that is, if it was the advancing or receeding motion of the surface/interface (see Figure 1.3 for explanation) [4, p.342]. The data

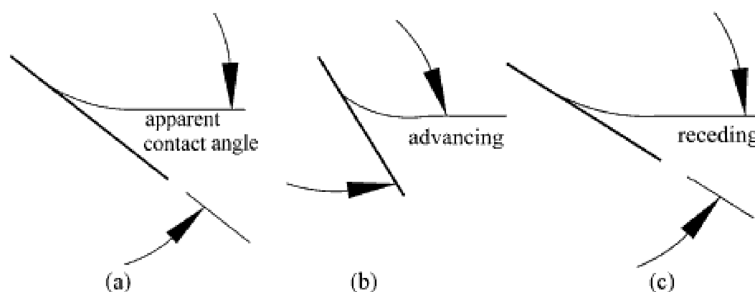

Figure 1.3: Schematic of equilibrium contact angle: θ (a) stationary liquid, (b) liquid flows upward, (c) liquid flows downward.

is grouped into the member function `thetaProps` for the interface-pairs. If we now focus on the multiphaseMixture.C code again, it calculates the dynamic contact angle if required, i.e if u_θ is not small (which is a number set by OpenFOAM somewhere in the order of 10 to the power of minus six:

```
if (uTheta > SMALL)
{
    scalar thetaA = convertToRad*tp().thetaA(matched);
    scalar thetaR = convertToRad*tp().thetaR(matched);

    // Calculated the component of the velocity parallel to the wall
    vectorField Uwall =
        U_.boundaryField()[patchi].patchInternalField()
```

```
              - U_.boundaryField()[patchi];
            Uwall -= (AfHatPatch & Uwall)*AfHatPatch;

            // Find the direction of the interface parallel to the wall
            vectorField nWall =
                nHatPatch - (AfHatPatch & nHatPatch)*AfHatPatch;

            // Normalise nWall
            nWall /= (mag(nWall) + SMALL);

            // Calculate Uwall resolved normal to the interface parallel to
            // the interface
            scalarField uwall = nWall & Uwall;

            theta += (thetaA - thetaR)*tanh(uwall/uTheta);
        }
```

The first section converts limiting angles to radians. Then proceeds with calculating velocity parallel to the wall such that: U_{wall} equals the boundary patch field with the internal patch field excluded. Then subtracts the unitvector on each cellface from the wall velocity: $U_{wall} = U_{wall}$ - (AfHatPatch $\cdot$ U_{wall})*AfHatPatch. The code then finds the direction of the interface parallel to the wall (n_{wall}), normalises it and calculates u_{wall}:

$$u_{wall} = n_{wall} \cdot U_{wall} \tag{1.9}$$

Then finally, corrects the angle θ by

$$\theta = (\theta_A - \theta_R) * tanh\left(\frac{u_{wall}}{u_\theta}\right) \tag{1.10}$$

This new angle is then used to reset `nHatPatch` (the direction of the contact interface) so that it corresponds to the contact angle. Finally, we are ready for the alpha-equation which will be used when calculating the new volumetric flux (ϕ). The equation reads:

$$\frac{d\alpha}{dt} + mvconvection- > fvmdiv(\phi, \alpha) \tag{1.11}$$

Where `fvmdiv` is the divergence of the flux and the α-field. The flux calculated from the alpha-equation is later used for `rhophiaa_`. Now the last stages of the code for this will be gone through.

```
surfaceScalarField phic = mag(phi_/mesh_.magSf());
     phic = min(cAlpha*phic, max(phic));
```

Here `phic` is defined, which is the minimum of ϕ for the current cell times the interface-compression `cAlpha` and the minimum of the maximum value of the flux, ϕ.

```
    for (int gCorr=0; gCorr<nAlphaCorr; gCorr++)
    {
        phase* refPhasePtr = &refPhase_;

        if (cycleAlpha)
        {
            PtrDictionary<phase>::iterator refPhaseIter = phases_.begin();
            for(label i=0; i<nSolves%phases_.size(); i++)
            {
                ++refPhaseIter;
            }
```

```
            refPhasePtr = &refPhaseIter();
        }

        phase& refPhase = *refPhasePtr;

        volScalarField refPhaseNew = refPhase;
        refPhaseNew == 1.0;

        rhoPhi_ = phi_*refPhase.rho();

        forAllIter(PtrDictionary<phase>, phases_, iter)
        {
            phase& alpha = iter();

            if (&alpha == &refPhase) continue;

            fvScalarMatrix alphaEqn
            (
                fvm::ddt(alpha)
              + mvConvection->fvmDiv(phi_, alpha)
            );
```

Above the alpha-equation (1.11) is seen.

```
            forAllIter(PtrDictionary<phase>, phases_, iter2)
            {
                phase& alpha2 = iter2();

                if (&alpha2 == &alpha) continue;

                surfaceScalarField phir = phic*nHatf(alpha, alpha2);
                surfaceScalarField phirb12 =
                    -fvc::flux(-phir, alpha2, alphacScheme);

                alphaEqn += fvm::div(phirb12, alpha, alphacScheme);
            }

            alphaEqn.solve(mesh_.solver("alpha"));

            rhoPhi_ += alphaEqn.flux()*(alpha.rho() - refPhase.rho());
```

Here a couple of terms are defined. First, `phir` which equals our previous minimum value, times the cell face unit interface normal flux for `alpha` and `alpha2`.
`surfaceScalarField phir = phic*nHatf(alpha, alpha2);`
Secondly `phirb12` which takes the flux of `phic` and `alpha2`.
`surfaceScalarField phirb12 = -fvc::flux(-phir, alpha2, alphacScheme);`
Then the divergence of `phirb12`, and alpha is added to the the alphaEqn. Now the `alphaEqn.flux` can finally be multiplied with our calculated ρ for our given `alpha` and subtracted by ρ_{ref}. ρ_{ref} denotes the density for the reference phase.

```
            Info<< alpha.name() << " volume fraction, min, max = "
                << alpha.weightedAverage(mesh_.V()).value()
                << ' ' << min(alpha).value()
                << ' ' << max(alpha).value()
```

```
                << endl;

            refPhaseNew == refPhaseNew - alpha;
        }

        refPhase == refPhaseNew;

        Info<< refPhase.name() << " volume fraction, min, max = "
            << refPhase.weightedAverage(mesh_.V()).value()
            << ' ' << min(refPhase).value()
            << ' ' << max(refPhase).value()
            << endl;
    }

    calcAlphas();
}
```

The reference phase is then set to its new alpha-value (fraction value) so that it is now updated and can be used for further calculations.

1.3.2 UEqn.H, pEqn.H and multiphaseInterFoam.C

The *multiphaseInterFoam.C* code is relatively short. Since the major solving is done in previously discussed multiphaseMixture section. The code solves for *UEqn.H* and *pEqn.H* by using the calculated mixture density ρ_{mix}. Below the code for *multiphaseInterFoam.C* is presented.

```
    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readPISOControls.H"
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "alphaCourantNo.H"
        #include "setDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        mixture.solve();
        rho = mixture.rho();

        #include "UEqn.H"

        // --- PISO loop
        for (int corr=0; corr<nCorr; corr++)
        {
            #include "pEqn.H"
        }

        turbulence->correct();

        runTime.write();
```

```
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}
```

As seen, the mixture.solve() calls the previously described code and ρ is set to the calculated mixture rho, ρ_{mix}. Now the U- and P-equation can be solved. Firstly, the U-equation:

$$\left(\frac{d}{dt}(\rho, U) + \nabla \cdot (\rho\phi_{mix}, U) - \nabla^2(\mu_{Eff}, U) - \nabla U \cdot \nabla \mu_{Eff}\right)_{implicit} - (\nabla \cdot (\mu_{Eff}))_{explicit} \quad (1.12)$$

where:

$$\begin{aligned} \mu_{Eff} &= \mu_f + \text{interpolate}(\rho^* \nu_t) \\ \mu_f &= mixture.muf() \\ \nu_t &= \text{turbulent viscosity} \end{aligned}$$

Furthermore, *UEqn.H* contains a boolean statement for momumentum predictor, which is often useful for the convergence, and the performance of the code. If true, it solves for equation (1.12) which is the momentum equation. See below.

$$UEqn = (F_s - g * \nabla \rho_{explicit} - \nabla P_{rgh,explicit}) * cellfacevectors \quad (1.13)$$

where:

$$\begin{aligned} F_s &= \text{the surface-tension-force derived from the mixture (see Equation 1.4)} \\ \nabla &= \text{facenormal-gradient} \end{aligned}$$

The momentum equations are firstly solved using the pressure field, from previous timestep. The solution of the momentum equation gives a new velocity field which does not satisfy the continuity conditions.Moving on in *multiphaseInterFoam*-code, to the piso-loop for the solving and correcting of the P-equation. When looking into the code for the p-equation, there are some steps that are unclear and we will for the time being leave them be. For example the purpose of the ddtphicorr operation at the very beginning of the code would be interesting to explain further since the information available this parameter are limited. What is know, is that the pressure corrector is looped a set number of times with ”ncorr” (see [5]), where ”ncorr” is set in the *FvSolution*-file to equal 4. In *FvSolution* we also set, nNonOrthogonalCorrectors to equal zero. This means that the solution has been judged stable enough to be able to solve without any non-orthogonal correction, which is fairly obvious since our mesh is purely orthogonal. The pressure corrector loop works as follows. The calculated velocities from *Ueqn.H* are used to assemble H(u) a velocity field without the pressure gradient [5]. This is then used to calculate the pressure-equation giving an new pressure field. Also calculated, is a new volumetric flux field consistent with the new pressure field. The new pressure field is used in an explicit velocity correction for the velocity field such that it now also matches the pressure field.

Summarizing the solution steps

Onno Ubbink summarizes the solution steps in a short and elegant way in his Phd work with the title: "Numerical prediction of two fluid systems with sharp interfaces" (see [1, p. 27]). The solution sequence is as follows:
1. Initialise all the variables.
2. Calculate the Courant number and adjust the time step if necessary.
3. Solve the equation by using the old time level's volumetric fluxes.
4. Use the new values together with the constitutive relations to obtain an estimate for the new viscosity, density and the face densities.
5. Use the above values to do a momentum prediction and continue with the PISO algorithm.
6. If the final time has not yet been reached advance to the next time level and return to step 2.

1.4 Running the case

We will now run the case without any modifications so that the results can be viewed and evaluated. Go to the case-folder in the run directory and execute blockMesh. When done, type `setFields`. This utility now sets the specified α-values for the different phases in their respective boxes as previously described in section 1.2.3 (see also figure 1.1). Now it is time to run the case, type:
`multiphaseInterFoam | tee log`

1.5 Post-processing

Now the results can finally be viewed by typing `paraFoam`. The best way to visualize it is by simply choosing the alphas parameter to visualize the 4 different phases.

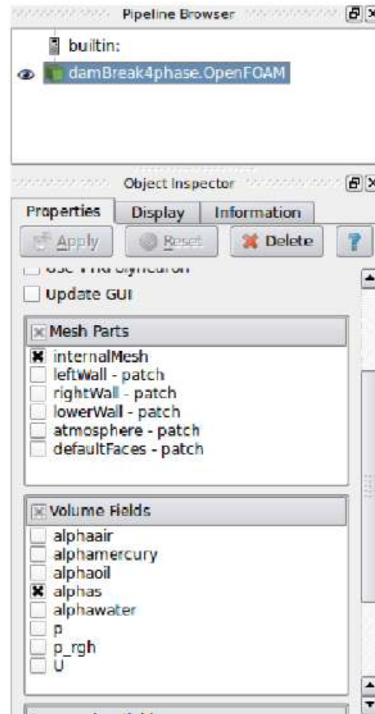

Figure 1.4: Setup in paraFoam.

Click apply, and then press the play button to see how the three liquids fall over the obstacle and finally settle down in layers corresponding to their densitys. Mercury on the bottom, oil in the middle, water on top and air above.

In order to investigate the solution further, the visualization of the velocity field with solid colored glyphs is a good idea. This gives the user a clearer picture of how the movement is propagating in the phases.

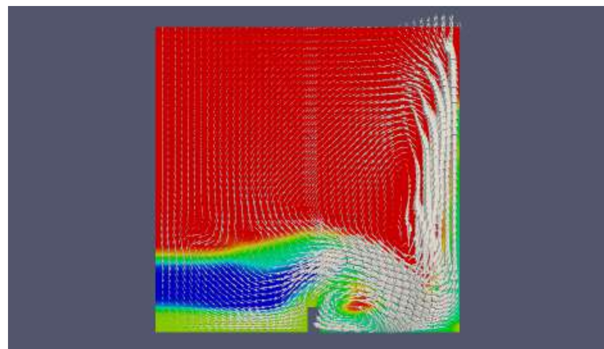

Figure 1.5: The 4 phases, demonstrated with the alphas-field and glyphs

Bibliography

[1] O. Ubbink, "Numerical prediction of two fluid systems with sharp interface," tech. rep., Department of Mechanical Engineering, London, England, 1997. http://powerlab.fsb.hr/ped/kturbo/OpenFOAM/docs/OnnoUbbinkPhD.pdf.

[2] OpenFoam documentation, "User guide, doxygen." http://www.openfoam.com/docs/user/, October 2010.

[3] Wikipedia, "Contact angle." http://en.wikipedia.org/wiki/Contact_angle, October 2010.

[4] Amir Faghri, Yuwen Zhang, *Transport phenomena in multiphase systems.* Academic Press, 2006.

[5] H. Nilsson, "implementapplication." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/implementApplication.pdf, September 2010.