# OpenFOAM tutorial: Free surface tutorial using interFoam and rasInterFoam

Hassan Hemida
Division of Fluid Dynamics, Department of Applied Mechanics
Chalmers University of Technology, SE-412 96 Göteborg, Sweden
e-mail: hemida@chalmers.se, hassan.hemida@epsilon.nu

April 14, 2008

# 1 Introduction

The aim of this tutorial is to explain to the new users of OpenFOAM how to use an existing tutorial in the release of OpenFOAM and modify it to suit a user case. The focus here is to use the existing tutorials for the iterFoam and rasInterFoam to solve laminar and turbulent free surface cases.

This tutorial demonstrates how to do the following:

- Set up and solve a transient problem using the VOF model.
- Modify the damBreak tutorial to solve for any VOF case.
- Change the geometry in OpenFOAM.
- Set up the properties of the two fluids.
- Initialize the flow.
- Choose the turbulent model.
- Brief explanation of the interFoam code.

# 2 Description of the problem

The problem considers the transient filling of water in a bottle initially filled with air as shown in Figure 1. For simplicity we will consider a two dimensional case by assuming that the geometry has an infinite length in the z-direction. The domain consists of two regions: an inlet chamber and a bottle. The dimensions are shown in Figure 1.

Initially, the bottle is filled with air and the inlet chamber is filled with water. At time zero, the water will move to fill the bottle. The inlet velocity of the water at the inlet section to the inlet chamber is 0.1 m/s. The time needed for filling the bottle with water is about 10 sec.

# 3 Available solvers in OpenFOAM

The case is a free surface problem, where a sharp interface between the two fluids exists. There are three different available solvers able to handle this case in OpenFOAM: interFoam, rasInterFoam and lesInterFoam.

interFoam: solver for 2 incompressible fluids capturing the interface using a VOF method. No turbulence model is used. Solution may be viewed as a DNS if the mesh is fine enough for DES otherwise it is laminar solver.

rasInterFoam: solver for 2 incompressible fluids capturing the interface using a VOF method. Turbulence is modeled using a runtime selectable incompressible RANS model.

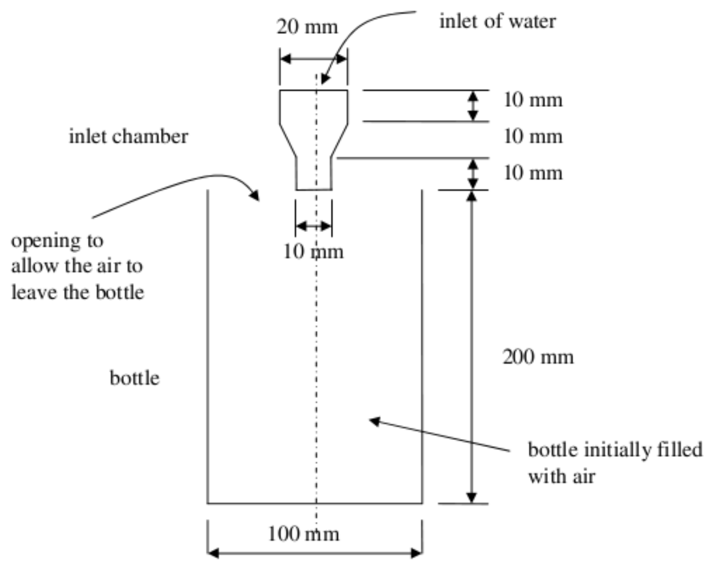


Figure 1: Schematic of the Problem

lesInterFoam: solver for 2 incompressible fluids capturing the interface using a VOF method. Turbulence is modeled using a runtime selectable incompressible LES model.

In this tutorial, we will use interFoam and rasInterFoam to solve the filling problem. Compression between the laminar solution using interFoam and the turbulent solution using rasInterFoam will be shown at the end of the tutorial.

# 4 The damBreak case

Now we will take a look at the directories and files in the original tutorial damBreak. The folder has three sub-directories, 0, constant and system.

## 4.1 The constant sub-directory

The folder contains all properties for the two fluids as will as the information required to build the mesh and the gravitational acceleration. It has three different files, transportProperties, enviromentalProperties and dynamicMeshDict, and one folder, polyMesh. The transportProperties file is a dictionary that contains information about the properties of the two phases as shown below.

```
\*------------------------transportProperties------------------------------*/
```

```
FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          transportProperties;
}

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

phase1
{
    transportModel  Newtonian;
    nu              nu [0 2 -1 0 0 0 0] 1e-06;
    rho             rho [1 -3 0 0 0 0 0] 1000;
    CrossPowerLawCoeffs
    {
        nu0             nu0 [0 2 -1 0 0 0 0] 1e-06;
        nuInf           nuInf [0 2 -1 0 0 0 0] 1e-06;
        m               m [0 0 1 0 0 0 0] 1;
        n               n [0 0 0 0 0 0 0] 0;
    }
    BirdCarreauCoeffs
    {
        nu0             nu0 [0 2 -1 0 0 0 0] 0.0142515;
        nuInf           nuInf [0 2 -1 0 0 0 0] 1e-06;
        k               k [0 0 1 0 0 0 0] 99.6;
        n               n [0 0 0 0 0 0 0] 0.1003;
    }
}

phase2
{
    transportModel  Newtonian;
    nu              nu [0 2 -1 0 0 0 0] 1.48e-05;
    rho             rho [1 -3 0 0 0 0 0] 1;
    CrossPowerLawCoeffs
    {
        nu0             nu0 [0 2 -1 0 0 0 0] 1e-06;
        nuInf           nuInf [0 2 -1 0 0 0 0] 1e-06;
        m               m [0 0 1 0 0 0 0] 1;
        n               n [0 0 0 0 0 0 0] 0;
    }
    BirdCarreauCoeffs
    {
        nu0             nu0 [0 2 -1 0 0 0 0] 0.0142515;
        nuInf           nuInf [0 2 -1 0 0 0 0] 1e-06;
        k               k [0 0 1 0 0 0 0] 99.6;
        n               n [0 0 0 0 0 0 0] 0.1003;
    }
}

sigma           sigma [1 0 -2 0 0 0 0] 0.07;


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

The dictionary sets the transport model for the two phases to be Newtonian and then gives information about the viscosity and density of the phases. It gives also some coefficients for two power laws used for the interpolation for the gamma function. At the end of the dictionary, the surface tension coefficient between the two phases is set. The dictionary environmentalProperties sets the value and direction of the gravitational acceleration as shown below.

```
\*--------------------environmentalProperties--------------   ---------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          environmentalProperties;
}

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

g              g [0 1 -2 0 0 0 0] (0 -9.81 0);


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

Here, the gravitational acceleration has a value of 9.81 m/s2 in the negative direction of y-axes.

The third dictionary in the constant folder is dynamicMeshDict which specify if the mesh is dynamic or static as follow.

```
\*--------------------------dynamicMeshDict----------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          motionProperties;
}

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

```
dynamicFvMesh    staticFvMesh;

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

As seen in the dynamicMeshDict, the mesh is set to be staticFvMesh.

The constant directory contains the sub-directory polyMesh where information about the mesh should be there in the dictionary blockMeshDict as shown below.

```
\*-----------------------------blockMeshDict---------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          blockMeshDict;
}

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

convertToMeters 0.146;

vertices
(
    (0 0 0)
    (2 0 0)
    (2.16438 0 0)
    (4 0 0)
    (0 0.32876 0)
    (2 0.32876 0)
    (2.16438 0.32876 0)
    (4 0.32876 0)
    (0 4 0)
    (2 4 0)
    (2.16438 4 0)
    (4 4 0)
    (0 0 0.1)
    (2 0 0.1)
    (2.16438 0 0.1)
    (4 0 0.1)
    (0 0.32876 0.1)
    (2 0.32876 0.1)
    (2.16438 0.32876 0.1)
    (4 0.32876 0.1)
    (0 4 0.1)
    (2 4 0.1)
    (2.16438 4 0.1)
    (4 4 0.1)
);

blocks
(
```

```
    hex (0 1 5 4 12 13 17 16) (23 8 1) simpleGrading (1 1 1)
    hex (2 3 7 6 14 15 19 18) (19 8 1) simpleGrading (1 1 1)
    hex (4 5 9 8 16 17 21 20) (23 42 1) simpleGrading (1 1 1)
    hex (5 6 10 9 17 18 22 21) (4 42 1) simpleGrading (1 1 1)
    hex (6 7 11 10 18 19 23 22) (19 42 1) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    wall leftWall
    (
        (0 12 16 4)
        (4 16 20 8)
    )
    wall rightWall
    (
        (7 19 15 3)
        (11 23 19 7)
    )
    wall lowerWall
    (
        (0 1 13 12)
        (1 5 17 13)
        (5 6 18 17)
        (2 14 18 6)
        (2 3 15 14)
    )
    patch atmosphere
    (
        (8 20 21 9)
        (9 21 22 10)
        (10 22 23 11)
    )
);

mergePatchPairs
(
);

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

The dictionary starts by setting a conversion to meter. If the dimensions in this dictionary are in mm then the first line should be

```
convertToMeters 0.001;
```

to convert it to meter.

The domain is divided into numbers blocks. Each block should have 8 vertices. All the vertices for all the blocks are sorted under vertices dictionary. Each vertex has x, y and z coordinate value. The vertices are numbered in ordered starting from zero for the first vertex.

The blocks are defined under the blocks directory. Each block has 8 vertices. These vertices are specified by there number according to their

appearance in the vertices dictionary. The line for each block start with the word hex meaning that the mesh will be hexahedral followed by the number of the 8 vertices connecting the block. After specifying the vertices connecting one block comes the number of grid points in the three direction x, y and z. At the end of this line a specification of the mesh grading in the three dimensions x, y and z is set. The grading in the damBreak tutorial is simpleGrading (1 1 1) meaning that the mesh is uniform mesh without any stretching. In this case, the domain has 5 hexa blocks.

Following the block dictionary is the edge dictionary where a curvature for any edge connecting two vertices can be specified. If this dictionary is empty then the edge will be a straight line in the mesh.

The patches dictionary specifies the boundary patches in the mesh. Each boundary face should be connected by four vertices. First the type of the boundary condition is specified for a patch and then the name of the patch. If a boundary patch is sharing many blocks, an entry for each peace of the patch in each block should be specified under the patch dictionary.

In our laminar fill bottle tutorial we will keep the same setting for all the dictionaries in the constant folder except the blockMeshDict. We will change this dictionary to fit or mesh in our tutorial for filling a bottle.

## 4.2 The system sub-directory

This folder contains five dictionaries, controlDict, decomposeParDict, fvSchemes and setFieldsDict. All the settings for starting, end and time step as well as the methods of saving data are in the contolDict. This dictionary takes the format shown below.

```
\*------------------------------contolDict-------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          controlDict;
}

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

application interFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;
```

```
endTime         1;

deltaT          0.001;

writeControl    adjustableRunTime;

writeInterval   0.05;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

adjustTimeStep  yes;

maxCo           0.5;

maxDeltaT       1;


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

The solver is specified at the beginning of the dictionary to be interFoam. The start time is zero and the end time is 1 sec. The simulation will stop at the end time which is 1 sec. The time step is set to be 0.001 sec but the adjustableTimeStep option is yes meaning that the time step will change during the simulation to keep the CFL number below the max value specified in maxCo. An out put file will be written every 0.05 sec. The simulation will adjust the time step to write exactly every 0.05 sec as specified in the commands

```
writeControl    adjustableRunTime;
```

and

```
runTimeModifiable yes;
```

All the descritization schemes are set in the dictionary fvSchemes shown below. For the dambreak tutorial the VanLeer scheme is used for the discritization of the divergence terms in the U-equations and in Gamma equation. The laplacian terms are descretized using the centeral difference scheme.

```
\*-------------------------fvSchemes------------------------------*/

FoamFile
{
```

```
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          fvSchemes;
}

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

ddtSchemes
{
    default Euler;
}

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

divSchemes
{
    div(rho*phi,U)  Gauss limitedLinearV 1;
    div(phi,gamma)  Gauss vanLeer;
    div(phirb,gamma) Gauss interfaceCompression;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

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


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

The snGradSchemes sets the descritization for calculating the normal vector to the surface. In order to get a sharp interface, an interface compression scheme is used.

All the settings for the solver as the maximum residuals and number of iterations are in the dictionary fvSolution shown below.

```
\*-------------------------fvSolution------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          fvSolution;
}

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

solvers
{
    pcorr PCG
    {
        preconditioner   DIC;
        tolerance        1e-10;
        relTol           0;
    };
    pd PCG
    {
        preconditioner   DIC;
        tolerance        1e-7;
        relTol           0.05;
    };
    pdFinal PCG
    {
        preconditioner   DIC;
        tolerance        1e-7;
        relTol           0;
    };
    U PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-06;
        relTol           0;
    };
}

PISO
{
    momentumPredictor no;
    nCorrectors     3;
    nNonOrthogonalCorrectors 0;
    nGammaCorr      1;
    nGammaSubCycles 2;
    cGamma          1;
```

```
}

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

The number of correction for the pressure in the PISO algorithm is 3. Four sub-cycles corrections are required for the gamma solver. the momentumPredictor is set to no.

In the free surface problems, the initial condition for the volume fraction of the two fluids needs to be well defined. If we specify uniform internal fields, then only one fluid will occupy the entire domain. Instead, the setFieldsDict divides the domain into two parts and give different gamma value for each region. The setFieldsDict is shown below.

```
\*---------------------------setFieldsDict--------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          setFieldsDict;
}

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

defaultFieldValues
(
    volScalarFieldValue gamma 0
    volVectorFieldValue U (0 0 0)
);

regions
(
    boxToCell
    {
        box (0 0 -1) (0.1461 0.292 1);

        fieldValues
        (
            volScalarFieldValue gamma 1
        );
    }
);


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

If we do not have the regions section then the default values will be activated. In this case the default values are 0 for gamma and 0 for U. In the regions dictionary, a box is defined with the form

```
box (minX minY minZ) (maxX maxY maxZ);
```

After we defined the box, the values for gamma inside the box is defined to be 1 as

```
fieldValues
(
    volScalarFieldValue gamma 1
);
```

It is possible also to set the values of the vector U inside the box.

The last dictionary in the sub-folder is the decomposeParDict which descibe the decomposition of the geometry and data in case of running the solver in parallel.

## 4.3 The 0 subdirectory

In this subdirectory, all the boundary values are given. All the patches that have been named in the polyMeshDict should assign a value here. The gamma dictionary is shown below.

```
\*-----------------------------Gamma field Dict-----------------------------*/

FoamFile
{
    version 2.0;
    format ascii;

    root "/home/hassan/OpenFOAM/hassan-1.4.1/run/interFoam";
    case "damBreak";
    instance "0";
    local "";

    class volScalarField;
    object gamma;
}

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

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;


boundaryField
{
    leftWall
    {
        type            zeroGradient;
    }
    rightWall
    {
```

```
        type            zeroGradient;
    }
    lowerWall
    {
        type            zeroGradient;
    }
    atmosphere
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

The same structure is applicable for the other dictionary U, that gives boundary conditions for the velocity fields, and pd, that gives boundary conditions for the pressure.

# 5 Setting the case

## 5.1 interFoam solver

The case is similar to the existing tutorial supplied by the OpenFoam distributors damBreak. A copy of this tutorial should be placed in the run directory as follow:

- run
- mkdir laminarFilling
- cd laminarFilling
- cp -r $WM_PROJECT_DIR/tutorials/interFoam/damBreak ./lamFillBottle

We will keep the setting as it is for all the dictionary except polyMeshdict, setFieldsDict and the boundary dictionaries in the subfolder 0 to suit our case. The geometry is divided into six blocks as shown in Figure 2.

The new polyMeshDict for our case is shown below.

```
\*---------------------------blockMeshDict-----------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;
```

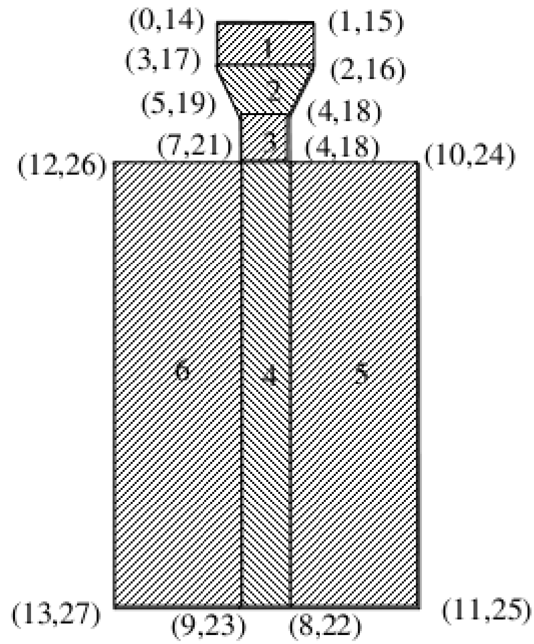


Figure 2: Different blocks in the domain

```
    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          blockMeshDict;
}

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

convertToMeters 0.001;

vertices
(
    (-10 230 0)
    (10 230 0)
    (10 220 0)
    (-10 220 0)
    (5 210 0)
    (-5 210 0)
    (5 200 0)
    (-5 200 0)
    (5 0 0)
    (-5 0 0)
    (50 200 0)
    (50 0 0)
    (-50 200 0)
    (-50 0 0)
    (-10 230 1)
```

```
    (10 230 1)
    (10 220 1)
    (-10 220 1)
    (5 210 1)
    (-5 210 1)
    (5 200 1)
    (-5 200 1)
    (5 0 1)
    (-5 0 1)
    (50 200 1)
    (50 0 1)
    (-50 200 1)
    (-50 0 1)
);

blocks
(
    hex (3 2 1 0 17 16 15 14) (10 5 1) simpleGrading (1 1 1)
    hex (5 4 2 3 19 18 16 17) (10 5 1) simpleGrading (1 1 1)
    hex (7 6 4 5 21 20 18 19) (10 5 1) simpleGrading (1 1 1)
    hex (9 8 6 7 23 22 20 21) (10 100 1) simpleGrading (1 1 1)
    hex (8 11 10 6 22 25 24 20) (40 100 1) simpleGrading (1 1 1)
    hex (13 9 7 12 27 23 21 26) (40 100 1) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    patch inlet
    (
        (0 1 15 14)
    )
    wall inletWall
    (
        (0 3 17 14)
        (3 5 19 17)
        (5 7 21 19)
        (1 2 16 15)
        (2 4 18 16)
        (4 6 20 18)
    )
    wall bottleWall
    (
        (10 11 25 24)
        (11 8 22 25)
        (8 9 23 22)
        (9 13 27 23)
        (13 12 26 27)
    )
    patch atmosphere
    (
        (6 10 24 20)
        (12 7 21 26)
    )
    empty frontAndBack
    (
        (0 1 2 3)
        (3 2 4 5)
        (4 6 7 5)
```

```
        (6 10 11 8)
        (6 8 9 7)
        (7 9 13 12)
        (14 15 16 17)
        (17 16 18 19)
        (19 18 20 21)
        (20 24 25 22)
        (20 22 23 21)
        (21 23 27 26)
    )
);

mergePatchPairs
(
);

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

Here we have four patches, inlet, bottleWall, inletWall, atmosphere amd frontAndBack specifying the inlet of the water, the wall of the bottle, the wall of the inlet pipe, the opening at the top of the bottle to the atmosphere and the front and back faces, respectively. The front and the back faces will be assigned empty type boundary condition in the boundary dictionaries to simplify the problem as 2D. Note the rotation between the different vertices to specify the corresponding block. It should be consistent for all the blocks. In this tutorial, the rotation is a right handed rotation around the positive z-axes.

The second dictionary to change is the setFieldsDict in the system subfolder. Here we need the inlet chamber to be filled with water at the beggining of the simulation while the bottle is filled with air. In this case, the setFieldsDict is modified to have a box around the inlet chamber as shown below.

```
\*-----------------------------setFieldsDict--------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          setFieldsDict;
}

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

defaultFieldValues
(
    volScalarFieldValue gamma 0
```

```
    volVectorFieldValue U (0 0 0)
);

regions
(
    boxToCell
    {
        box (-0.01 0.2 -1) (0.01 0.23 1);

        fieldValues
        (
            volScalarFieldValue gamma 1
        );
    }
);


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

Note that the bounding box might be large and extended outside the domain. When we set the fields using the setFields utility, the cells inside the box will be only assigned the value of gamma equal to 1.

We need also to change the boundary patches in the boundary dictionaries in the sub-folder 0. The gamma dictionary is shown below.

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

FoamFile
{
    version 2.0;
    format ascii;

    root "/home/hassan/OpenFOAM/hassan-1.4.1/run/interFoam";
    case "fillBottle";
    instance "0";
    local "";

    class volScalarField;
    object gamma;
}

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

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 1;
    }
    inletWall
    {
        type            zeroGradient;
    }
```

```
    bottleWall
    {
        type            zeroGradient;
    }
    atmosphere
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    frontAndBack
    {
        type            empty;
    }
}


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

We should also change by the same way the dictionaries U and pd.

### 5.1.1 Generating the mesh

The mesh will be generated by running the utility blockMesh. Go the directory laminarFilling and run the utility for the case lamFillBottle as

```
blockMesh . lamFillBottle
```

After running the utility we see some other files in the directory constant/polyMesh. These files contains information about the coordinates of the mesh points and cell boundaries and so on.

### 5.1.2 Setting the initial and boundary conditions of $\gamma$

If we start the run with the existing sitting of $\gamma$ the initial field will be filled with only air. To initialize the solution with water in the inlet pipe and air in the bottle we need to run the setFields utility on our case as follow:

```
setFilds . lamFillBottle
```

Take a look at the gamma dictionary in the 0 sub-folder to see the difference.

### 5.1.3 Start the simulation

The simulation is ready now to start. The user can start the simulation by typing the name of the solver followed by the root of the case and then the case name as shown below.

```
interFoam . lamFillBottle > out.log&
```

The output information in each time step will be in the file out.log.

## 5.2 rasInterFoam solver

This solver solve for two phase flow separated by sharp interface. The difference between resInterFoam and interFoam is that RANS turbulence models are implemented in rasInterFoam solver. The settings for the rasInterFoam solver are similar to that for the interFoam solver. All the dictionaries are the same except the turbulenceProperties dictionary which is located in the sub-folder constant. This dictionary is shown below.

```
// * * * * * * * * * * * *turbulenceProperties* * * * * * * * * * * * * * * //

turbulenceModel kEpsilon;

turbulence      on;

laminarCoeffs
{
}

kEpsilonCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
}

RNGkEpsilonCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.0845;
    C1              C1 [0 0 0 0 0 0 0] 1.42;
    C2              C2 [0 0 0 0 0 0 0] 1.68;
    alphak          alphaK [0 0 0 0 0 0 0] 1.39;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 1.39;
    eta0            eta0 [0 0 0 0 0 0 0] 4.38;
    beta            beta [0 0 0 0 0 0 0] 0.012;
}

NonlinearKEShihCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphak          alphak [0 0 0 0 0 0 0] 1;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76932;
    A1              A1 [0 0 0 0 0 0 0] 1.25;
    A2              A2 [0 0 0 0 0 0 0] 1000;
    Ctau1           Ctau1 [0 0 0 0 0 0 0] -4;
    Ctau2           Ctau2 [0 0 0 0 0 0 0] 13;
    Ctau3           Ctau3 [0 0 0 0 0 0 0] -2;
    alphaKsi        alphaKsi [0 0 0 0 0 0 0] 0.9;
}

LienCubicKECoeffs
{
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphak          alphak [0 0 0 0 0 0 0] 1;
```

```
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
    A1              A1 [0 0 0 0 0 0 0] 1.25;
    A2              A2 [0 0 0 0 0 0 0] 1000;
    Ctau1           Ctau1 [0 0 0 0 0 0 0] -4;
    Ctau2           Ctau2 [0 0 0 0 0 0 0] 13;
    Ctau3           Ctau3 [0 0 0 0 0 0 0] -2;
    alphaKsi        alphaKsi [0 0 0 0 0 0 0] 0.9;
}

QZetaCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphaZeta       alphaZeta [0 0 0 0 0 0 0] 0.76923;
    anisotropic     no;
}

LaunderSharmaKECoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
}

LamBremhorstKECoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
}

LienCubicKELowReCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphak          alphak [0 0 0 0 0 0 0] 1;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
    A1              A1 [0 0 0 0 0 0 0] 1.25;
    A2              A2 [0 0 0 0 0 0 0] 1000;
    Ctau1           Ctau1 [0 0 0 0 0 0 0] -4;
    Ctau2           Ctau2 [0 0 0 0 0 0 0] 13;
    Ctau3           Ctau3 [0 0 0 0 0 0 0] -2;
    alphaKsi        alphaKsi [0 0 0 0 0 0 0] 0.9;
    Am              Am [0 0 0 0 0 0 0] 0.016;
    Aepsilon        Aepsilon [0 0 0 0 0 0 0] 0.263;
    Amu             Amu [0 0 0 0 0 0 0] 0.00222;
}

LienLeschzinerLowReCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    alphak          alphak [0 0 0 0 0 0 0] 1;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
    Am              Am [0 0 0 0 0 0 0] 0.016;
    Aepsilon        Aepsilon [0 0 0 0 0 0 0] 0.263;
    Amu             Amu [0 0 0 0 0 0 0] 0.00222;
```

```
}

LRRCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    Clrr1           Clrr1 [0 0 0 0 0 0 0] 1.8;
    Clrr2           Clrr2 [0 0 0 0 0 0 0] 0.6;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    Cs              Cs [0 0 0 0 0 0 0] 0.25;
    Ceps            Ceps [0 0 0 0 0 0 0] 0.15;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
}

LaunderGibsonRSTMCoeffs
{
    Cmu             Cmu [0 0 0 0 0 0 0] 0.09;
    Clg1            Clg1 [0 0 0 0 0 0 0] 1.8;
    Clg2            Clg2 [0 0 0 0 0 0 0] 0.6;
    C1              C1 [0 0 0 0 0 0 0] 1.44;
    C2              C2 [0 0 0 0 0 0 0] 1.92;
    C1Ref           C1Ref [0 0 0 0 0 0 0] 0.5;
    C2Ref           C2Ref [0 0 0 0 0 0 0] 0.3;
    Cs              Cs [0 0 0 0 0 0 0] 0.25;
    Ceps            Ceps [0 0 0 0 0 0 0] 0.15;
    alphaEps        alphaEps [0 0 0 0 0 0 0] 0.76923;
    alphaR          alphaR [0 0 0 0 0 0 0] 1.22;
}

SpalartAllmarasCoeffs
{
    alphaNut        alphaNut [0 0 0 0 0 0 0] 1.5;
    Cb1             Cb1 [0 0 0 0 0 0 0] 0.1355;
    Cb2             Cb2 [0 0 0 0 0 0 0] 0.622;
    Cw2             Cw2 [0 0 0 0 0 0 0] 0.3;
    Cw3             Cw3 [0 0 0 0 0 0 0] 2;
    Cv1             Cv1 [0 0 0 0 0 0 0] 7.1;
    Cv2             Cv2 [0 0 0 0 0 0 0] 5.0;
}

wallFunctionCoeffs
{
    kappa           kappa [0 0 0 0 0 0 0] 0.4187;
    E               E [0 0 0 0 0 0 0] 9;
}


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

At the begining of the solver, the turbulence model is set. in this case the standard k-epsilon model is used. Following this line is the activation of using a turbulence model. if we choose

```
turbulence      on;
```

then the turbulence model is activated. If we set

```
turbulence      off;
```

then a laminar solution will be the default.

The rest of the dictionary is the coefficients for the available turbulence models that can be used in this case.

The case is similar to the existing tutorial supplied by the OpenFOAM distributors damBreak in the rasInterFoam tutorial. A copy of this tutorial should be placed in the run directory as follow:

- run
- mkdir turbelentFilling
- cd turbelentFilling
- cp -r $WM_PROJECT_DIR/tutorials/rasInterFoam/damBreak ./turbFillBottle

The user should modify the case similar to that of the lamFillBottle tutorial. the solution can now be started by typing

```
resInterFoam . turbFillBottle > out.log&.
```

The progress of the solution can be watched during the simulation by typing

```
tail -f out.log
```

on the shell.

# 6 Postprocessing

The paraFoam will be used to visualize the transient solution. This can be done by typing the following command line in the shell

```
paraFoam . lamFillBottle
```

to open the software and upload the case.

Figure 3.a shows the mesh shape generated using blockMesh utility. The same mesh is used for both interFoam and rasInterFoam solvers. The user can modify the grading of the mesh to resolve the boundary layers close to the walls.

Figure 3.b shows a plane at the middle of the domain colored by the volume fraction (red color correspond to gamma=1, which is water and blue color corresponds to gamma = 0, which is air). It can be seen from the figure that initially the inlet pipe is filled with water while the bottle is filled with air.

Figure 4 shows the field gamma after 0.2 sec from the start of the filling process. Figure 4 shows that the water stream breaks up in interFoam while it is continuous in rasInterFoam.

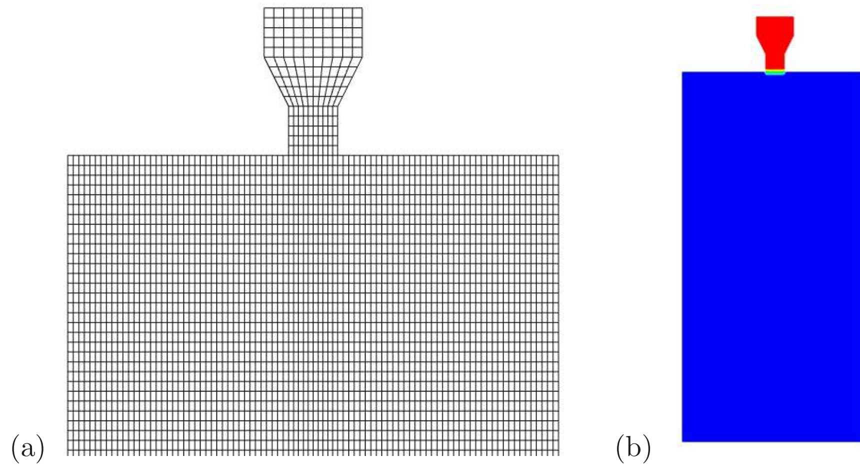


Figure 3: (a)the mesh and (b) initial field of gamma.

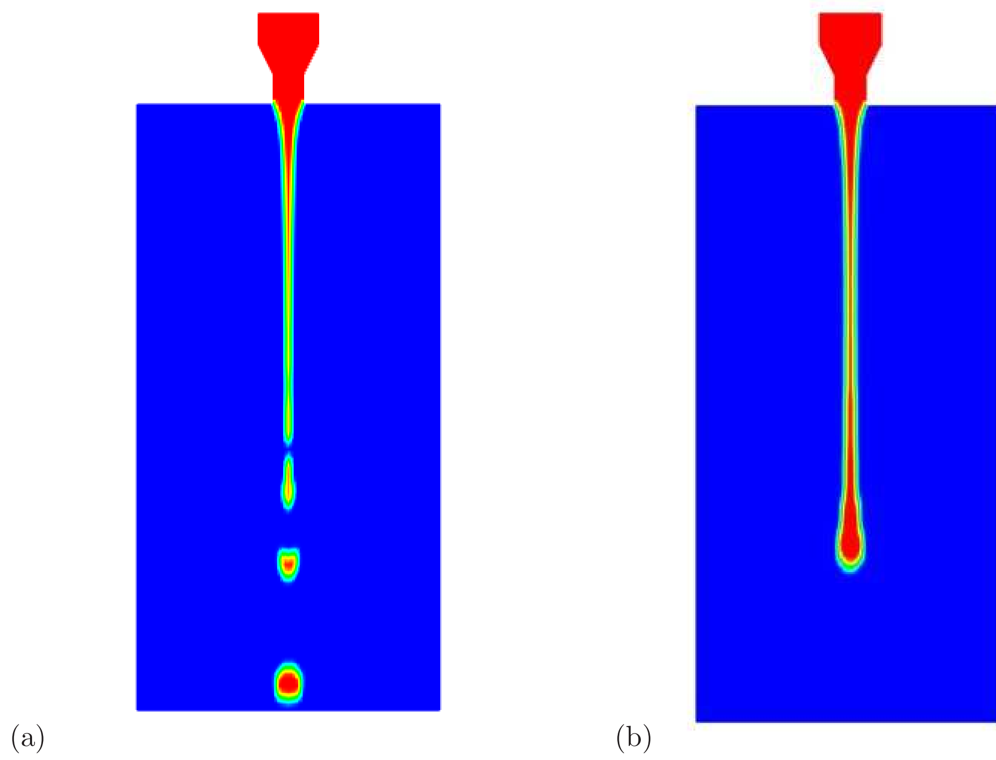


Figure 4: Field gamma after 0.2 sec from the start. (a) interFoam and (b) rasInterFoam.

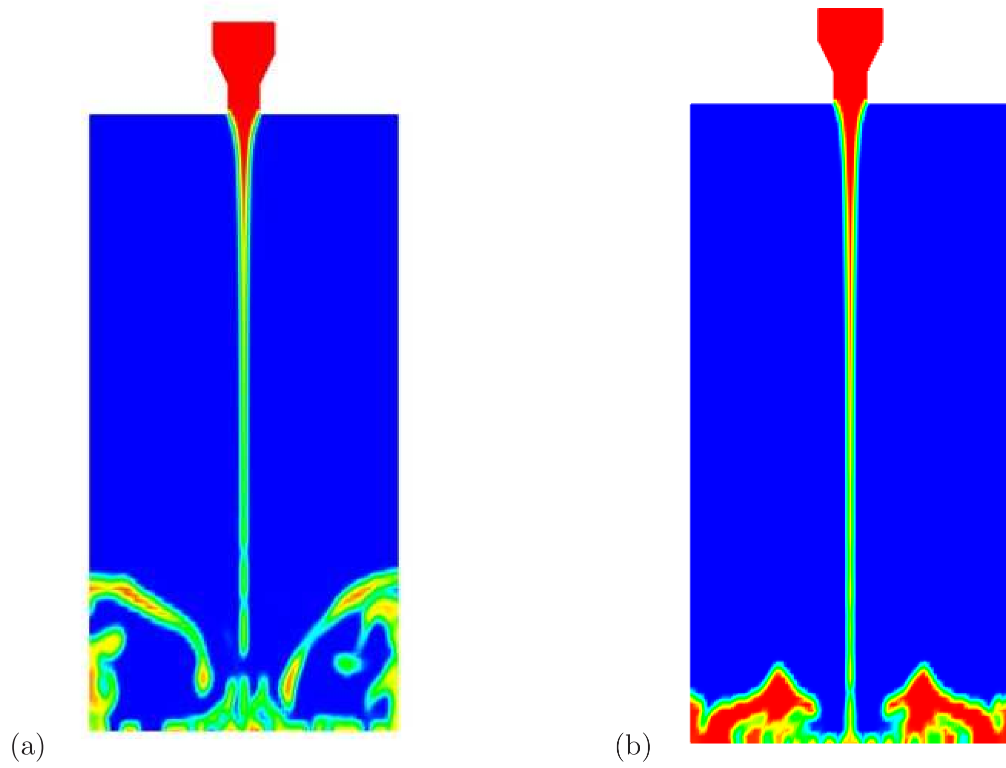


Figure 5: Field gamma after 0.4 sec from the start. (a) interFoam and (b) rasInterFoam.

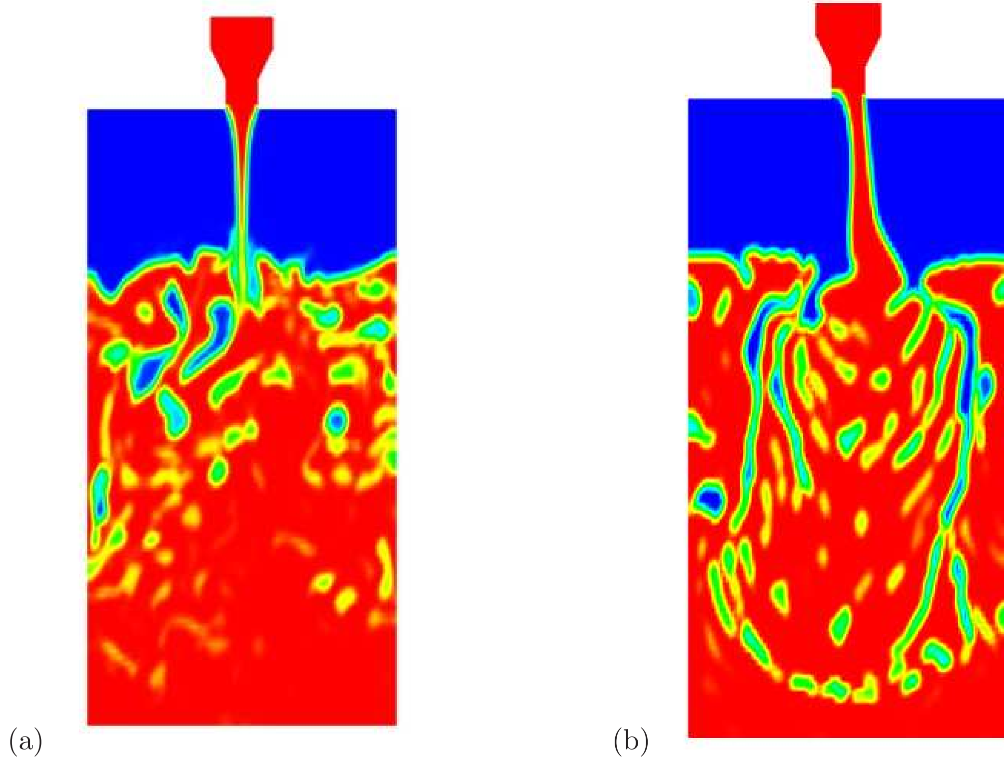


Figure 6: Field gamma after 8 sec from the start. (a) interFoam and (b) rasInterFoam.

Figure 5 shows the field gamma after 0.4 sec from the start of the filling process. Figure 5 shows that the fields obtained from interFoam and from rasInterFoam are different.

Figure 6 shows the field gamma after 8 sec from the start of the filling process. Figure 6 shows that the amount of air trapped in the water obtained from interFoam and from rasInterFoam are different.

# 7 Implementation in the code

In the VOF method we solve one momentum equation and one continuity equation. These equation are the same for the two phases. The physical properties of one fluid are calculated as weighted averages based on the volume fraction of the two fluids in one cell. The momentum equation takes the form

$$\frac{\partial \rho U}{\partial t} + \nabla.(\rho U U) - \nabla.\mu\nabla U - \rho g = -\nabla p - F_s \tag{1}$$

where $F_s$ is the surface tension force which takes place only at the free surfaces.

The continuity equation takes the form

$$\nabla . U = 0. \tag{2}$$

The volume of fluid in a cell is computed as $F_{vol} = \gamma V_{cell}$, where $V_{cell}$ is the volume of a computational cell and $\gamma$ is the fluid fraction in a cell. The values of $\gamma$ in a cell should range between 1 and 0. If the cell is completely filled with fluid then $\gamma = 1$ and if it is filled with the void phase then its value should be 0. At the interface the value of $\gamma$ is between 0 and 1. The scalar function $\gamma$ can be computes from a separate transport equation that takes the form:

$$\frac{\partial \gamma}{\partial t} + \nabla . (\gamma U) = 0. \tag{3}$$

In OpenFOAM, the necessary compression of the surface is achieved by intreducing an extra artificial compression term into the VOF equation 3 as follow:

$$\frac{\partial \gamma}{\partial t} + \nabla . (\gamma U) + \nabla . (\gamma (1 - \gamma) U_r) = 0 \tag{4}$$

wher $U_r$ is a velocity field suitable to compress the interface. This artificial term is active only in the interface region due to the term $(\gamma(1 - \gamma)$.

The density at any point in the domain is calculated as a weighted averaged of the volume fraction of the two fluids, $\gamma$ as:

$$\rho = \gamma \rho + (1 - \gamma) \rho. \tag{5}$$

The surface tension $F_s$ is computed as:

$$F_s = \sigma \kappa (x) n \tag{6}$$

where $n$ is a unit vector normal to the interface that can be calculated from

$$n = \frac{\nabla \gamma}{|\nabla \gamma|} \tag{7}$$

and $\kappa$ is the curvature of the interface that can be calculated from

$$\kappa (x) = \nabla . n. \tag{8}$$

These equations are implemented in the interFoam solver as shown in the source code shown below.

```
/*------------------------interFoam code-------------------------------*/
int main(int argc, char *argv[])
{

#   include "setRootCase.H"
```

```
#   include "createTime.H"
#   include "createMesh.H"
#   include "readEnvironmentalProperties.H"
#   include "readPISOControls.H"
#   include "initContinuityErrs.H"
#   include "createFields.H"
#   include "readTimeControls.H"
#   include "correctPhi.H"
#   include "setInitialDeltaT.H"

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

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

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

        runTime++;

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

        twoPhaseProperties.correct();

#       include "gammaEqnSubCycle.H"

#       include "UEqn.H"

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

#       include "continuityErrs.H"

        runTime.write();

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

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

    return(0);
}


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

The first part of the code is to initialize the different fields and to get the libraries needed for the finite volume solution.

The time iteration loop starts at the command

```
    while (runTime.run())
    {
```

```
#       include "readPISOControls.H"
#       include "readTimeControls.H"
#       include "CourantNo.H"
#       include "setDeltaT.H"

        runTime++;

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

where the the solver calculates the correction of PISO loop around the $\gamma$ phase. It calculates also the CFL number and based on the value of CFL number it calculates the new time step. The solver increases the run time and then print out the information about the new run iteration, current time step, elapsed time, CFL and so on.

The next part of the solver is to compute the phase fraction $\gamma$ as shown below.

```
#       include "gammaEqnSubCycle.H"
```

For better accuracy of the gamma field and to make stable solution we need very small time step to solve for the gamma equation. In order to achieve stable solution without reducing the time step and vastly increasing solution time, the time step is divided to number of sub-cycle time step. The gamma equation is solved in each sub-cycle. The sub-cycle algorithm if in the header file gammaEqnSubCycle.H shown below.

```
/******************* gammaEqnSubCycle.H *****************/

label nGammaCorr
(
    readLabel(piso.lookup("nGammaCorr"))
);

label nGammaSubCycles
(
    readLabel(piso.lookup("nGammaSubCycles"))
);

if (nGammaSubCycles > 1)
{
    dimensionedScalar totalDeltaT = runTime.deltaT();
    surfaceScalarField rhoPhiSum = 0.0*rhoPhi;

    for
    (
        subCycle<volScalarField> gammaSubCycle(gamma, nGammaSubCycles);
        !(++gammaSubCycle).end();
    )
    {
#       include "gammaEqn.H"
        rhoPhiSum += (runTime.deltaT()/totalDeltaT)*rhoPhi;
    }

    rhoPhi = rhoPhiSum;
}
```

```
else
{
#       include "gammaEqn.H"
}

interface.correct();

rho == gamma*rho1 + (scalar(1) - gamma)*rho2;
```

The algorithm checks the number of the sub-cycles. If the number of sub-cycles are more than one then it loops to solve the gamma equation in each subCycle otherwise it solves the gamma equation once. At the end of the algorithm, the new fluid properties are calculated using the weighted averaged of the gamma field. The algorithm to solve for gamma equation is a special algorithm for OpenFOAM developed by Henry Weller and it uses a technique called interfaceCompression to resolve some of the fundamental problems of the traditional VOF interface compression methods.

The next part of the code is to solve the momentum equation to find the velocity field. This is done by inserting the UEqn.H as shown below.

```
#       include "gammaEqnSubCycle.H"
```

The UEqn.H has the implementations of the momentum equation.

```
/*--------------------UEqn.H-----------------------------------*/
    surfaceScalarField muf = twoPhaseProperties.muf();

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

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

The first part of the UEqn.H is the LHS of the momentum equation. The last part of the algorithm is to solve the LHS of the momentum equation which is equal to the gravity and surface tension forces.

The last section of the solver before it finishes the current time step is to calculate the pressure correction through the PISO loop and to calculate the continuity error to check for convergence before it writes out the information about the residuals and errors.