Cite as: Ghahramani, E.: Improvement of the VOF-LPT Solver for Bubbles. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H.,
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Improvement of the VOF-LPT Solver for Bubbles

Developed for OpenFOAM-3.0.0

Author:
Ebrahim Ghahramani
Chalmers University of Technology
ebrahim.ghahramani@chalmers.se

Peer reviewed by:
Alessandro Tassone
Håkan Nilsson

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

January 20, 2017

Learning outcomes

The reader will learn:

- The theoretical background of Volume of Fluid and Lagrangian Particle Tracking approaches
- How to use the LPT-VOF solver
- How to add new member data and Rayleigh-Plesset equation parameters to the solver
- How to solve Rayleigh-Plesset equation numerically in OpenFOAM

Prerequisites

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

- Fundamentals of Volume of Fluid approach
- Fundamentals of Lagrangian Particle Tracking
- How to Run standard document tutorials like damBreak tutorial
- It is highly recommended to have a look at the presentation file of the VOF-LPT solver development by [Vallier (2011)], since this tutorial is based Vallier's work.

Contents

Preface

In this tutorial the VOF-LPT solver is briefly described first. This solver is not included in the general OpenFOAM solver, but it has been developed and presented in detail by Aurélia Vallier in the OpenFOAM course [Vallier (2011)]. In this solver, the particles are considered to be rigid. If we need to model bubble particles in the flow field, then a model for simulating the variation of bubble size should be implemented to the solver and this is the main purpose of the current tutorial. After description, the solver is updated for OpenFOAM 3.0.0., and is improved to consider particle (bubble) size variations.

In the following section, the theories of Volume of Fluid (VOF) and Lagrangian Particle Tracking (LPT) approaches along with the well-known Rayleigh-Plesset equation are described. Then the LPT-VOF solver that was developed previously by coupling the interFoam solver and the lagrangian library of OpenFOAM is introduced. After that the updating and improving procedures of the solver are described. Finally a sample problem is solved to explain how to use the improved solver. It should be mentioned that the final solver and the sample problem case are available in the accompanying files on the course homepage.

Chapter 1

Theory

In this section the theory and governing equations of the VOF and LPT approaches are described.

1.1 Volume of Fluid

Volume of Fluid (VOF) is an Eulerian approach for modelling multiphase flows. In this approach the multiphase flow is treated as a single fluid mixture and the main governing equations are still the well-know continuity and Navier-Stokes equations, given by

$$\frac{\partial \rho_m}{\partial t} + \frac{\partial \left(\rho_m u_i\right)}{\partial x_i} = 0, \tag{1.1}$$

$$\frac{\partial \left(\rho_m u_i\right)}{\partial t} + \frac{\partial \left(\rho_m u_i u_j\right)}{\partial x_j} = \frac{\partial \tau_{ij}}{\partial x_j} + \rho_m g_i + S_p + S_{st} \tag{1.2}$$

where τ_{ij} is the stress tensor defined as

$$\tau_{ij} = -p\delta_{ij} + \mu_m \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_i}{\partial x_j} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij} \right) \tag{1.3}$$

In the above equations ρ_m and μ_m are the mixture density and mixture dynamic viscosity defined as

$$\rho_m = \alpha\rho_1 + (1-\alpha)\rho_2 \qquad \mu_m = \alpha\mu_1 + (1-\alpha)\mu_2 \tag{1.4}$$

where indices 1 and 2 refer to fluid 1 and fluid 2 respectively (assuming a 2-fluid flow) and α is the volume fraction of the fluid 1 which is obtained by solving a scalar transport equation as

$$\frac{\partial \alpha}{\partial t} + \frac{\partial \left(\alpha u_i\right)}{\partial x_i} = S_\alpha \tag{1.5}$$

in this equation S_α is the source term of the equation which is set to zero in the $interFoam$ solver. In equation 1.2, the term S_p on the right hand side denotes the particle forces. If the particles are dense enough, their exerted forces on the surrounding liquid can be considerable. The effect of particle on the fluid momentum can be expressed as

$$S_p = \frac{1}{\Delta V}\sum_{k=1}^{N} m_{p,k}\frac{du_{p,k}}{dt} \tag{1.6}$$

where N is the number of particles in the control volume, ΔV. It should be noted that the original interFoam solver does not include particles, and this source term is therefore zero. However, it is added to the momentum equation in the VOF-LPT solver. Finally the last term, S_{st}, is the surface tension force which is defined as

$$S_{st} = -\sigma_{st}\kappa\delta n, \qquad \delta = |\nabla\alpha|, \qquad n = \frac{\nabla\alpha}{|\nabla\alpha|}, \qquad \kappa = \nabla n \tag{1.7}$$

Here, σ_{st} is the surface tension coefficient of fluid, κ is the curvature of the two-phase interface which has a unit vector of n. Also, δ is a function that ensures that the surface tension force is applied only at the two-phase interface.

1.2 Lagrangian Particle Tracking

The motion of particles in fluids is described in a Lagrangian way by solving a set of ordinary differential equations along the particle trajectory. A particle p is defined by the position of its center x_p, its diameter d_p , its velocity $u_{p,i}$ and its density ρ_p .The goal is to calculate the change of particle location and the particle velocity. Various forces are exerted on a small particle during its motion; a list of the most common forces includes drag, lift, gravity, buoyancy, Brownian, pressure and volume variation forces and the effect of added mass and particle history forces. In this tutorial we just consider the buoyancy, drag and gravity forces. Therefore, the Lagrangian equations of motion for a particle are given by

$$\begin{aligned} \frac{dx_{p,i}}{dt} &= u_{p,i} \\ m_p \frac{du_{p,i}}{dt} &= m_p \frac{(u_{f,i} - u_{p,i})}{\tau_p} + (\rho_p - \rho_f) g_i \end{aligned} \tag{1.8}$$

The last term is the combination of buoyancy and gravity forces and the term before that is the drag force in which $u_{f,i}$ is the fluid velocity and τ_p is the particle relaxation time that is defined as

$$\tau_p = \frac{4}{3} \frac{\rho_p d_p^2}{\rho_f C_D |u_{f,i} - u_{p,i}|} \tag{1.9}$$

In this equation, C_D is the drag coefficient, defined as

$$C_D = \begin{cases} \frac{24}{Re_p} & if Re_p < 0.01 \\ \frac{24}{Re_p}(1 + 0.15 Re_p^{0.687}) & if Re_p > 0.01 \end{cases} \tag{1.10}$$

where, Re_p is the particle Reynolds number, given by

$$Re_p = \frac{\rho_f d_p |u_{f,i} - u_{p,i}|}{\mu_f} \tag{1.11}$$

1.3 Rayleigh-Plesset Equation

The size of non-rigid particles (e.g. bubbles) may vary due to the flow field variations. For a single bubble, the variation of diameter in incompressible liquid flows is usually expressed by the well-known Rayleigh-Plesset equation, given by

$$R(t)\ddot{R}(t) + \frac{3}{2}\dot{R}^2(t) = \frac{P_B - P_l}{\rho_l} - 4\nu_l \frac{\dot{R}(t)}{R(t)} - \frac{2\sigma_{st}}{\rho_l R(t)} \tag{1.12}$$

In this equation $R(t)$ is the bubble radius at time t, $\dot{R}$ and $\ddot{R}$ are the first and second time derivatives of radius, respectively, P_l is the liquid pressure around the bubble, ρ_l is the liquid density (denoted as ρ_m in equation 1.4), ν_l is the fluid kinematic viscosity and σ_{st} is the fluid surface tension. Also, P_B is the bubble inside pressure, defined as

$$P_B = P_v + P_{g0} \left(\frac{R_0}{R(t)} \right)^{3k} \tag{1.13}$$

where, P_v is the vapour pressure that is a function temperature (usually kept constant in incompressible flows). The last term is the pressure of dissolved gas inside the bubble, in which P_{g0} is the

equilibrium gas pressure at the equilibrium state with $R = R_0$. Finally, k is the polytropic compression constant which is equal to 1.4 for air in an adiabatic process and equal to 1 in an isothermal process.
In order to solve the Rayleigh-Plesset equation, the equilibrium radius (R_0) and dissolved gas pressure (P_{g0}) for some reference pressure (which is named P_0) should be known as well. It is therefore necessary to compute the equilibrium radius which satisfies the condition

$$ {R_0}^{3\gamma} + \frac{2\sigma_{st}}{P_0 - P_v} {R_0}^{3\gamma-1} + \frac{R^{3\gamma}}{P_0 - P_v} \left(P_v - \frac{2\sigma_{st}}{R} - P_l \right) = 0 \tag{1.14} $$

Also, the corresponding dissolved gas pressure is obtained from

$$ P_{g0} = P_0 + \frac{2\sigma_{st}}{R_0} - P_v \tag{1.15} $$

More details about the Rayleigh-Plesset equation can be found in [Franc and Michel (2004)].

Chapter 2

The VOF-LPT Solver

In this chapter the VOF-LPT solver is described briefly. This solver was developed first by Vallier for OpenFOAM 1.6-dev [Vallier (2011)]. To have a more comprehensive perception of the current tutorial, It is recommended to read the [Vallier (2011)] presentation file regarding the development of the solver. Here the solver and libraries are described based on OpenFOAM 3.0.0. Updating Vallier's code to OpenFOAM 3.0.0 through modification of individual command lines is quite tricky. Rather, it is suggested to start from scratch and follow Vallier's suggested steps in the new version. Since an updated version of the code is not available in the literature, the main files of the solver (before the improvements of this tutorial) are made available for the reader in Appendix A. The following description of the solver is based on these files.
After the description, another necessary step for updating the solver will be explained.

2.1 interFoam

The `interFoam` solver is a multiphase solver for 2 incompressible, isothermal immiscible fluids based on the VOF approach. Therefore, the momentum and other fluid properties are of the "mixture" and a single momentum equation is solved. Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected. The source code of the solver is located at

`$WM_PROJECT_DIR/applications/solvers/multiphase/interFoam`

where the following files/folders are found

```
alphaEqn.H
alphaEqnSubCycle.H
alphaCourantNo.H
correctPhi.H
createFields.H
interFoam.C
pEqn.H
UEqn.H
setDeltaT.H
setRDeltaT.H
Make
interMixingFoam
interDyMFoam
```

As there are many similarities between the `interFoam` solver and the general `pimpleFoam` solver, here the most important features of `interFoam` which are not present in other solvers are explained. In createFields.H file, the necessary fields are created. Among them there is an object named `mixture`, of the class `immiscibleIncompressibleTwoPhaseMixture`.

```
immiscibleIncompressibleTwoPhaseMixture mixture(U, phi);
```

The `immiscibleIncompressibleTwoPhaseMixture` class is actually an immiscible incompressible two-phase mixture transport model that calculates the transport properties of the mixture. This class is located at

```
$FOAM_SRC/transportModels/immiscibleIncompressibleTwoPhaseMixture
```

Also, in the `createFields.H` file we have:

```
const dimensionedScalar& rho1 = mixture.rho1();
const dimensionedScalar& rho2 = mixture.rho2();
```

which create two dimensioned scalars for the densities of the two fluids of the mixture. The main file of the solver is `interFoam.C`. A copy of this file can be found in appendix A.1. In the main function of the file, after setting the solution parameters, we have the pimple loop as

```
while (pimple.loop())
        {
            #include "alphaControls.H"
            #include "alphaEqnSubCycle.H"

            mixture.correct();

            #include "UEqn.H"

            // --- Pressure corrector loop
            while (pimple.correct())
            {
                #include "pEqn.H"
            }

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }
```

In this loop, the `alphaControl.H` file is included first, which calculates and outputs the mean and maximum Courant numbers for solving alpha equation. Then the `alphaEqnSubCycle.H` file is included. This file contains the `alphaEqn.H` file to solve the alpha equation. The mixture density (ρ_m) is also updated in `alphaEqnSubCycle.H`. After solving the alpha equation, the `correct()` function of the `immiscibleIncompressibleTwoPhaseMixture` class is called via `mixture.correct();`. This function updates the mixture viscosity (μ_m) and two-phase interface curvature (κ). The interface curvature is used in calculating the surface tension force in the momentum equation. Then the momentum equation is solved by including the `UEqn.H` file and the pressure is corrected by including the `pEqn.H` file (which solves the continuity equation). Finally, the turbulence quantities are calculated by calling the turbulence `correct()` function.
Vallier coupled the `interFoam` solver with the lagrangian library by constructing a solidParticleCloud object in the main function of the `interFoam.C` file [Vallier (2011)]. In the following section the lagrangian library and the coupling procedure is described.

2.2 The Lagrangian Library

To add the lagrangian library to the `interFoam` solver, it is recommended to copy the solver folder to the user directory

```
cd $WM_PROJECT_USER_DIR/applications/
mkdir myLPTVOF
cd myLPTVOF
cp -r $WM_PROJECT_DIR/applications/solvers/multiphase/interFoam/* .
```

Then one should copy the contents of the solidParticle library folder to the directory of our new solver (except the make and lnInclude folders)

```
cp $WM_PROJECT_DIR/src/lagrangian/solidParticle/* .
```

The new files that are added to the solver folder are:

```
solidParticleCloud.C
solidParticleCloud.H
solidParticleCloudI.H
solidParticle.C
solidParticle.H
solidParticleI.H
solidParticleIO.C
```

Here these files are described briefly and the reader may refer to the copies of these files in appendix (A.2-A.8) for a more comprehensive perception of the following description.
The `solidParticleCloud` class is a class that contains a cloud of solid particles and track them in the flow field. It is a subclass of `Cloud<solidParticle>` class. The class `Cloud` is a base `cloud` class which is templated on the particle type. This class is a subclass of the `cloud` class. A cloud is a collection of lagrangian particles. The `Cloud` class has some general functions for tracking a collection of particles such as creating or deleting particles, move function, etc. This class is located at

```
$FOAM_SRC/lagrangian/basic/Cloud
```

The `solidParticle` class is a subclass of the `particle` class, with specific member data (e.g. particle diameter and velocity) and functions (e.g. particle interaction with boundaries) for treating every single particle.
For coupling the `interFoam` solver with the `solidParticleCloud` class, in the VOF-LPT code the `solidParticleCloud` class is included in the `interFoam.C` file before the main function (appendix A.1)

```
#include "solidParticleCloud.H"
```

Then an object of this class is constructed before the main time loop (appendix A.1)

```
solidParticleCloud particles(mesh);
```

Also, the particles are transformed at each time step by the following command in the time loop (after the pimple loop, appendix A.1)

```
particles.move(g);
```

where `g` is the gravitational acceleration vector. Also, the particle-particle interaction is calculated by the following command in the time loop (after the `move(g)` command, appendix A.1)

```
particles.checkCo();
```

When a particle is transformed to an Eulerian structure, its velocity and volume fraction is added to the continuum fluid through the following commands in the time loop (after the `checkCo()` command, appendix A.1)

```
alpha1 += particles.AddTOAlpha();
U += particles.AddToU();
```

Finally, the effect of particle motion on the momentum of surrounding fluid is considered by adding `particles.momentumSource()` to the right hand side of the momentum predictor equation in the `UEqn.H` file as

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

The command `"particles.move(g)"` calls the `move` function of the `solidParticleCloud`. In this function, the `move` function of the `Cloud` class is called through the following command (appendix A.4, `solidParticleCloud.C`)

```
Cloud<solidParticle>::move(td, mesh_.time().deltaTValue());
```

In the move function of the `Cloud` class there is a loop over all particles of the class in which another `move` function is called. This new `move` function is defined in `solidParticle` class (`solidParticle.C`, appendix A.7) and is called in the `Cloud` class via the following command

```
p.move(td, trackTime);
```

In the `solidParticle` move function the velocity of each particle is updated by solving its equation of motion and the particle is tracked for a time step equal to `trackTime` (`solidParticle.C`, appendix A.7). So, the tracking of the cloud of particles is accomplished through the nested calling of move functions as

```
particles.move(g) -> Cloud<solidParticle>::move(td, mesh_.time().deltaTValue())
->   p.move(td, trackTime)
```

The Rayleigh-Plesset equation (which is the contribution of this tutorial) will be defined and solved in the `move` function of the `solidParticle` class and this will be explained later.
For testing the implementations, Vallier (2011) injected two solidParticle in the domain. For specifiying the properties of such particles, she added new member data to the `solidParticleCloud` class. The new data are `{UP_1, UP_2}`, `{dP_1, dP_2}` and `{PosP_1, PosP_2}` which are respectively the velocities, diameters and positions of the injected particles. Also, another member data was added to the `solidParticleCloud` class to account for the particle forces on each of the computational domain. This member data is `smom` (appendix A.2). `correctalphaW` and `correctU` are other member data for adding particle velocity and volume fraction to the continuum fluid when a particle is transformed to an Eulerian structure. For more information about the code and other governing equations one may refer to [Vallier (2011)] and [Vallier (2013)].

2.3 Adapting "inject" Function

Another function that Vallier (2011) added to `solidParticleCloud` is the `inject` function which injects new particles to the flow domain when the flow time is in a specific time interval (appendices A.2 and A.4). This time interval is defined by the user through the parameters `tInjStart` and `tInjEnd`.
As mentioned before, to develop the VOF-LPT solver in new versions of OpenFOAM, one should follow Vallier's steps, rather than modifying the original code in version 1.6-dev. However, there is still one function that needs to be adapted in the new versions, the `inject` function.
For injecting a particle to the domain at a specific position, it is needed to find the related grid cell at that position. This job is done by calling the function `findCell`. In OpenFOAM 1.6-dev, this function takes only one argument which is particle position vector. However, in new versions, such as OpenFOAM 3.0.0, it takes and return three other arguments as well. These arguments are *a cell index* (e.g. `cellI`), *a tetFace index* (e.g. `tetFaceI`) and *a tetPt index* (e.g. `tetPtI`). `tetFace` is index of the face that owns the decomposed tet that the particle is in. `tetPt` is Index of the point on the face that defines the decomposed tet that the particle is in (relative to the face base point).
Therefore, for updating the VOF-LPT solver to a new version, in addition to following Vallier's steps, it is needed to modify the definition of inject function as follows

```
void Foam::solidParticleCloud::inject(solidParticle::trackingData &td)
{
label cellI=1;
    label tetFaceI=1;
    label tetPtI=1;
    mesh_.findCellFacePt(td.cloud().posP1_, cellI, tetFaceI, tetPtI);

    solidParticle* ptr1 = new solidParticle(mesh_, td.cloud().posP1_, cellI,
    tetFaceI, tetPtI,td.cloud().dP1_, td.cloud().UP1_);
    Cloud<solidParticle>::addParticle(ptr1);

    mesh_.findCellFacePt(td.cloud().posP2_, cellI, tetFaceI, tetPtI);

    solidParticle* ptr2 = new solidParticle(mesh_, td.cloud().posP2_, cellI,
    tetFaceI, tetPtI, td.cloud().dP2_, td.cloud().UP2_);
    Cloud<solidParticle>::addParticle(ptr2);
}
```

Chapter 3

Improving the Code for Bubbles

In this chapter the required steps for implementing the Rayleigh-Plesset equation in the code is described. This equation is solved to consider the size variation of a bubble particle due to the effect of surrounding liquid such as liquid pressure, viscous effects and surface tension (equation 1.12). In the following sections, the new commands that should be added in each of the files of the solver are described.

3.1 solidParticle.H

To solve this equation, in addition to the flow properties which can be extracted from the flow field, we need to define new parameters (member data in class) for bubble particle. These parameters are

Table 3.1: member data added to `solidParticle` class

Description	In equations 1.12 & 1.13	In the solver
bubble radius	R	R_
temporal derivative of bubble radius	$\dot{R}$	Rdt_
equilibrium radius	R_0	R0_
dissolved gas pressure	P_g	pG_
equilibrium dissolved gas pressure	P_{g0}	pG0_
bubble inside pressure	P_B	pB_
a numerical variable in solving the equation[a]	_	F0old_
a numerical variable in solving the equation	_	F1old_

[a] will be described later

Therefore the following lines should be added to *private data* of the class, after the declaration of `U_`,

```
//- temporal derivative of the Radius
scalar Rdt_;
//- Radius
scalar R_;
//- Initial equilibirium radius
scalar R0_;
//- bubble inside pressure
scalar pB_;
//- dissolved gas pressure
scalar pG_;
scalar pG0_;
```

```
scalar F0Old_;
scalar F1Old_;
```

Inside the `solidParticle` class there is another defined class, named `trackingData`. This class is used to pass tracking data to the `trackToFace` function (used in the move function of `solidParticle`). In addition to the previous member data functions of this class (e.g. rhoInterp_ and rhoInterp()) a member data and an access function should be added to this class in order to pass the flow pressure to the particle move function. Therefore, the following line should be added in the `trackingData` class after declaration of `alphaWInterp_`

```
const interpolationCellPoint<scalar>& pInterp_;
```

Also in the the argument list of the constructor function, after the `alphaWInterp`, one should add

```
const interpolationCellPoint<scalar>& pInterp,
```

And in the declaration of the member functions the access function for the flow pressure should be declared

```
inline const interpolationCellPoint<scalar>& pInterp() const;
```

When a new bubble particle is constructed, the previously defined parameters of the `solidParticle` class are initialized by the constructor function. However, the new parameters (table 3.1) are initialized through another function, named as `initialEquilibriumRadius`. This function should be declared in the solidParticle header as well

```
bool initialEquilibriumRadius(trackingData& );
```

3.2 solidParticleI.H

Some inline functions are modified / added in the `solidParticle.H` file and the corresponding modifications need to be done here as well. In the constructor function of the `trackingData` class, add the following line to the argument list after `alphaWInterp`

```
const interpolationCellPoint<scalar>& pInterp,
```

Also, in the initialization part of the same function, after initializing `alphaWInterp_`, one should add:

```
pInterp_(pInterp),
```

Finally, the definition of the inline access function for the flow pressure (`pInterp_`) should be included as

```
inline const Foam::interpolationCellPoint<Foam::scalar>&
Foam::solidParticle::trackingData::pInterp() const
{
    return pInterp_;
}
```

3.3 solidParticleIO.C

This file needs several modifications due to the addition of new member data to the `solidParticle` class. First, in the definition of the constructor, the command `is >> U_` for the ascii format should be modified as

```
is >> U_ >> Rdt_ >> R_ >> R0_ >> pB_ >> pG_ >> pG0_ >> F0Old_ >> F1Old_;
```

Then, the command `is.read(reinterpret_cast<char*>(&d_), sizeofFields_);` should be replaced by

```
is.read(reinterpret_cast<char*>(&d_),
      sizeof(d_)
      + sizeof(U_)
      + sizeof(Rdt_)
      + sizeof(R_)
      + sizeof(R0_)
      + sizeof(pB_)
      + sizeof(pG_)
      + sizeof(pG0_)
      + sizeof(F0Old_)
      + sizeof(F1Old_));
```

After that, in the `readFields` function, after the `c.checkFieldIOobject(c, U);` command, one should add

```
IOField<scalar> Rdt(c.fieldIOobject("Rdt", IOobject::MUST_READ));
c.checkFieldIOobject(c, Rdt);

IOField<scalar> R(c.fieldIOobject("R", IOobject::MUST_READ));
c.checkFieldIOobject(c, R);

IOField<scalar> R0(c.fieldIOobject("R0", IOobject::MUST_READ));
c.checkFieldIOobject(c, R0);

IOField<scalar> pB(c.fieldIOobject("pB", IOobject::MUST_READ));
c.checkFieldIOobject(c, pB);

IOField<scalar> pG(c.fieldIOobject("pG", IOobject::MUST_READ));
c.checkFieldIOobject(c, pG);

IOField<scalar> pG0(c.fieldIOobject("pG0", IOobject::MUST_READ));
c.checkFieldIOobject(c, pG0);

IOField<scalar> F0Old(c.fieldIOobject("F0Old", IOobject::MUST_READ));
c.checkFieldIOobject(c, F0Old);

IOField<scalar> F1Old(c.fieldIOobject("F1Old", IOobject::MUST_READ));
c.checkFieldIOobject(c, F1Old);
```

Also in the `forAll` loop of this function, before the `i++` command the following commands should be added

```
 p.Rdt_ = Rdt[i];
 p.R_ = R[i];
 p.R0_ = R0[i];
 p.pB_ = pB[i];
```

```
p.pG_ = pG[i];
p.pG0_ = pG0[i];
p.F0Old_ = F0Old[i];
p.F1Old_ = F1Old[i];
```

Then in the `writeFields` function after the command

```
IOField<vector> U(c.fieldIOobject("U", IOobject::NO_READ), np);
```

the following commands should be added

```
IOField<scalar> Rdt(c.fieldIOobject("Rdt", IOobject::NO_READ), np);
IOField<scalar> R(c.fieldIOobject("R", IOobject::NO_READ), np);
IOField<scalar> R0(c.fieldIOobject("R0", IOobject::NO_READ), np);
IOField<scalar> pB(c.fieldIOobject("pB", IOobject::NO_READ), np);
IOField<scalar> pG(c.fieldIOobject("pG", IOobject::NO_READ), np);
IOField<scalar> pG0(c.fieldIOobject("pG0", IOobject::NO_READ), np);
IOField<scalar> F0Old(c.fieldIOobject("F0Old", IOobject::NO_READ), np);
IOField<scalar> F1Old(c.fieldIOobject("F1Old", IOobject::NO_READ), np);
```

And in the `forAll` loop of this function, before the `i++` command the following commands one should add:

```
Rdt[i] = p.Rdt_;
R[i] = p.R_;
R0[i] = p.R0_;
pB[i] = p.pB_;
pG[i] = p.pG_;
pG0[i] = p.pG0_;
F0Old[i] = p.F0Old_;
F1Old[i] =p.F1Old_;
```

Also, out of the loop and after the the `U.wrtie()` command, it is needed to add the following line

```
Rdt.write();
R.write();
R0.write();
pB.write();
pG.write();
pG0.write();
F0Old.write();
F1Old.write();
```

Finally for the `Ostream` operator, after the `<< token::SPACE << p.U_` command the following command should be added

```
<< token::SPACE << p.Rdt_
<< token::SPACE << p.R_
<< token::SPACE << p.R0_
<< token::SPACE << p.pB_
<< token::SPACE << p.pG_
<< token::SPACE << p.pG0_
<< token::SPACE << p.F0Old_
<< token::SPACE << p.F1Old_;
```

And after that, the command `solidParticle::sizeofFields_` should be replaced by

```
        sizeof(p.d_)
        + sizeof(p.U_)
        + sizeof(p.Rdt_)
        + sizeof(p.R_)
        + sizeof(p.R0_)
        + sizeof(p.pB_)
        + sizeof(p.pG_)
        + sizeof(p.pG0_)
        + sizeof(p.F0Old_)
        + sizeof(p.F1Old_)
```

3.4 solidParticle.C

To calculate the bubble size at each time step, the Rayleigh-Plesset equation (1.11) should be solved for each time step. For this purpose the numerical calculation of this equation is involved in the `solidParticle:move` function. But before that, the interpolated pressure at the particle location should be calculated first. So, after the interpolation of `nuc` (`td.nuInterp().interpolate(cpw);`), add the following line

```
td.nuInterp().interpolate(cpw);
```

The time-step adaptive second-order Rosenbrock method [Shampine and Reichelt (1997)] is implemented to solve the Rayleigh-Plesset equation numerically. The commands are presented first, and followed by a description based on the line numbers. In the particle move function, after the command `td.cloud().smom()[cellI] += newMom-oldMom;`, one should add the following lines

```
1  /* solving the Rayleigh-Plesset with the Rosenbrock method */
2   /* ======================================================== */
3  scalar hrk = 1.0E-10;
4  scalar h   = 0.5*hrk;
5  scalar dD  = 1.0/(2.0+sqrt(2.0));
6  scalar time_rk = 0;
7  scalar sigma = td.cloud().sigma();
8  scalar pV = td.cloud().pSat();
9
10 rhoc = 998.85;
11 nuc  = 6.93305e-04;
12
13 scalar substeps = 0;
14
15 while (time_rk < trackTime)
16   {
17
18      if (time_rk+h > trackTime)
19        {
20 h = trackTime-time_rk;
21        }
22
23 // f
24 scalar F0 = Rdt_;
25 scalar F1 = -1.5*(Rdt_*Rdt_/R_)+((pB_-
26 pc)/(rhoc*R_))-4.0*nuc*(Rdt_/(R_*R_))-((2.0*sigma)/(rhoc*R_*R_));
27
28 // f0
29 scalar F00 = F0;
```

```
30 scalar F01 = F1;
31
32 // the jacobian dF/dX
33 scalar J00 = 0.0;
34 scalar J01 = 1.0;
35 scalar J10 = 1.5*((Rdt_*Rdt_)/(R_*R_))-((pB_-pc)/(rhoc*R_*R_))+
36   ((8.0*nuc*Rdt_)/(R_*R_*R_))+((4.0*sigma)/(rhoc*R_*R_*R_));
37 scalar J11 = -3.0*Rdt_/R_-4.0*(nuc/(R_*R_));
38
39 // W = I - h dD J
40 scalar W00 = 1.0 - h*dD*J00;
41 scalar W01 = 0.0 - h*dD*J01;
42 scalar W10 = 0.0 - h*dD*J10;
43 scalar W11 = 1.0 - h*dD*J11;
44
45 // inv W
46 scalar ff = 1.0/(W00*W11-W01*W10);
47 scalar invW00 =  ff*W11;
48 scalar invW01 = -ff*W01;
49 scalar invW10 = -ff*W10;
50 scalar invW11 =  ff*W00;
51
52 // the time derivative of function, dF/dt
53 // estimated numerically
54 scalar T0 = (F0-F0Old_)/h;
55 scalar T1 = (F1-F1Old_)/h;
56
57 // k1
58 scalar tmp0 = F00+h*dD*T0;
59 scalar tmp1 = F01+h*dD*T1;
60 scalar k10 = invW00*tmp0+invW01*tmp1;
61 scalar k11 = invW10*tmp0+invW11*tmp1;
62
63 // y_n + 0.5 h k1
64 scalar R1   = R_+0.5*h*k10;
65 scalar Rdt1 = Rdt_+0.5*h*k11;
66
67 // f1
68 scalar F10 = Rdt1;
69 scalar F11 = -1.5*(Rdt1*Rdt1/R1)+((pB_ -
70 pc)/(rhoc*R1))-4.0*nuc*(Rdt1/(R1*R1))-((2.0*sigma)/(rhoc*R1*R1));
71 // k2
72 tmp0       = F10-k10;
73 tmp1       = F11-k11;
74 scalar k20 = invW00*tmp0+invW01*tmp1+k10;
75 scalar k21 = invW10*tmp0+invW11*tmp1+k11;
76
77 // y_n+1
78 scalar R2   = R_+h*k20;
79 scalar Rdt2 = Rdt_+h*k21;
80
81 // f2
82 scalar F20 = Rdt2;
83 scalar F21 = -1.5*(Rdt2*Rdt2/R2)+((pB_ -
```

```
84  pc)/(rhoc*R2))-4.0*nuc*(Rdt2/(R2*R2))-((2.0*sigma)/(rhoc*R2*R2));
85
86  // k3
87  scalar b   = 6.0 + sqrt(2.0);
88  tmp0       = F20-b*(k20-F10)-2.0*(k10-F00)+h*dD*T0;
89  tmp1       = F21-b*(k21-F11)-2.0*(k11-F01)+h*dD*T1;
90  scalar k30 = invW00*tmp0+invW01*tmp1;
91  scalar k31 = invW10*tmp0+invW11*tmp1;
92  // error
93  scalar err0 = fabs((h/6.0)*(k10-2.0*k20+k30));
94  scalar err1 = fabs((h/6.0)*(k11-2.0*k21+k31));
95  scalar err  = max(err0,err1);
96
97  time_rk  += h;
98          substeps += 1;
99
100 scalar absTol = 1.0E-5;
101
102
103 if (err > 1.0E-08)
104   {
105     h = min(0.5*hrk, 0.5*h);
106   }
107 else
108   {
109     h = max(1.5*hrk, 1.5*h);
110   }
111 // update bubble state
112 F0Old_ = F0;
113 F1Old_ = F1;
114 R_   = R2;
115 d_ = 2.0*R_;
116 Rdt_ = Rdt2;
117
118
119 if (R_ < R0_)
120   {
121     pG_ = pG0_*pow(R0_/R_,3.0);
122   }
123 else
124   {
125     pG_ = pG0_*pow(R0_/R_,3.0*1.4);
126   }
127 pB_ = pG_ + pV;
128 }
129
130  /* ========================================================= */
131  /* End of solving the Rayleigh-Plesset */
```

In lines 3 and 4, `h` and `hrk` are defined for the step size of numerical solution of the derivative equation. In line 5, `dD` is a constant number to be used later. `time_rk` is the evolved time of the solution. Since we use a multi-step method to solve the equation, after each step, the evolved time should be checked with the particle Tracktime. `time_rk` is increased by `h` after each step. The parameters `sigma` and `pV` are surface tension coefficient and vapour pressure which are obtained

from the `solidParticleCloud` class. `rhoc` and `nuc` are the fluid density and kinematic viscosity. At line 15, the `time_rk` is checked with the trackTime. While it is smaller than the trackTime, another step of the equation solution is performed. At line 18, if the `time_rk` becomes larger than the trackTime at the end of the new step, then the step size (`h`) is adapted to avoid that. At line 23, the function vector of the Rosenbrock method is constructed. The general solution for the function vector (y_n) using the Rosenbrock method is given by

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i \tag{3.1}$$

where s denotes the number of stages, h is the step size and k_i are increment vectors obtained for $i = 1, 2, ..., s$ by solving

$$W_i k_i = F\left(y_n + h\sum_{j=1}^{i-1} a_{ij} k_j\right) + hJ\sum_{j=1}^{i-1} d_{ij} k_j \tag{3.2}$$

In the above equation, J is the Jacobian of function vector. More details about the method and other terms may be found in [Shampine and Reichelt (1997)].
Since we have a 2nd order equation, then we need to solve two linear equation. Therefore the system of equations consists of two linear equation to find two functions which are $\dot{R}$ (denoted by `F0`) and $\ddot{R}$ (denoted by `F01`). Let's call this function vector as `f`. The first solution (previous time step values) of the function is stored in vector `f0` (which has the elements `F00` and `F01` in lines 27-29). Then the Jacobian of the function vector is calculated and stored in `J` matrix in lines 31-36. In lines 38-42 the `W` matrix is calculated based on the formula in line 38. In lines 44-49, the inverse of matrix `W` is calculated and stored in matrix `invW`. In lines 51-54, the time derivation of the function vector is calculated. Then the first `k` vector is obtained in lines 56-60 (`k1`), which is used to find the first intermediate solution of the 3-step calculation process. In lines 63-64 the intermediate radius and its time derivative are calculated and based on these values, the intermediate function vector is obtained and stored in vector `f1`. In lines 70-74 the second `k` vector is obtained followed by intermediate bubble radius and its time derivative (76-78). The second intermediate function vector (`f2`) is now obtained in lines 81-83 similar to `f1`. In lines 85-90 the 3rd (last) `k` vector is obtained which will be used to find the main solution of the equation. Before the solution, the error is estimated in lines 91-94 and the time evolution (`time_rk`) is increased by `h`. A new value for `h` (for the next stage of solution) is estimated in lines 102-109 and finally the new bubble diameter, radius and its time derivative is obtained in lines 111-115. Here, it can be seen that the new solution is stored into particle member data (`F0Old` and `F1Old`) to be used for the next time solution. Based on the new value of bubble radius, the dissolved gas pressure (P_g) is updated in lines 118-125. If the bubble radius is smaller than the equilibrium radius (R_0) the new pressure is obtained using the isothermal relation (line 120). Otherwise, it will be obtained based on the adiabatic assumption (line 124). Finally, the bubble pressure is updated at line 126.

`initialEquilibriumRadius` is the last function to be added to this file. As stated before, this function is implemented to initialize / set the new added member data of the `solidParticle` class. In this function, `R0_` and `pG0_` are set based on the equlibrium equations (equations 1.14 and 1.15), while other member data of table 3.1 are initialized. The definition of the function `initialEquilibriumRadius` is included in the `solidParticle.C` file (e.g. after the `move` function) according to the following lines

```
1 /* Rayleigh-Plesset: initial equilibrium radius */
2
3 bool Foam::solidParticle::initialEquilibriumRadius
4 (
5     trackingData& td
6 )
```

```
{
    td.switchProcessor = false;
    td.keepParticle = true;

    // remember which cell the parcel is in
    // since this will change if a face is hit
    label cellI = cell();

    cellPointWeight cpw(mesh_, position(), cellI, face());
    scalar pc  = td.pInterp().interpolate(cpw);


    /// initial equlibirium radius
    scalar p0 = td.cloud().p0();
    scalar pV = td.cloud().pSat();
    scalar sigma = td.cloud().sigma();
    scalar gamma = td.cloud().gamma();
    scalar R  = 0.5*d_;
    R_        = 0.5*d_;
    Rdt_      = 0.0;
    F0Old_ = 0.0;
    F1Old_ = 0.0;


    // initial guess R0 equal to R
    scalar R0 = R;
    scalar R0new = R;
    scalar R0bar = R;
    scalar R0Old = R0;
    scalar error = 1;


    scalar a = (2.0*sigma)/(p0-pV);
    int maxIter = 2000;
    int iter    = 0;
    while ((error > 1.0E-06) && (iter < maxIter))
      {
if (R < R0)
  {
    gamma = 1.4;
  }
else
  {
    gamma = 1.0;
  }
scalar  b = (pow(R,3.0*gamma)/(p0-pV))*(pV-((2.0*sigma)/R)-pc);
scalar  f = pow(R0,3.0*gamma) + a*pow(R0,3.0*gamma-1.0) + b;
scalar df = 3.0*gamma*pow(R0,3*gamma-1.0) +
  a*(3.0*gamma-1.0)*pow(R0,3.0*gamma-2.0);
R0bar  = R0 - (f/df);
if (R < R0bar)
  {
    gamma = 1.4;
  }
```

```
61 else
62   {
63     gamma = 1.0;
64   }
65 b = (pow(R,3.0*gamma)/(p0-pV))*(pV-((2.0*sigma)/R)-pc);
66 scalar fbar =  pow(R0bar,3.0*gamma) + a*pow(R0bar,3.0*gamma-1.0) + b;
67 R0new  = R0 - (f*f)/(df*(f-fbar));
68 R0new  = 0.9*R0 + 0.1*R0new;
69 error = fabs((R0new-R0)/R0);
70 R0     = R0new;
71 iter++;
72
73       }
74
75     R0_  = R0;
76     pG0_ = p0 + 2.0*sigma/R0_ - pV;
77     if (R < R0)
78       {
79 gamma = 1.4;
80       }
81     else
82       {
83 gamma = 1.0;
84       }
85     pG_ = pG0_*pow(R0_/R_,3.0);
86     pB_  = pG_ + pV;
87
88     return true;
89 }
90
```

In this function, line 8 relates to parallel simulations to avoid switching the processor zone in which the particle is located. The next line sets a boolean variable to keep the bubble particle in the simulation. Line 13 stores the current cell id in which the particle center is located. Lines 15 and 16 return the flow pressure arround the bubble particle. lines 20-23 set the scalars `p0, pV, sigma` and `gamma` from the `solidParticleCloud`. These scalars, as well as `sclar R` will be used later in the function. Lines 25-28 initialize `R_ , Rdt_, F0Old` and `F1Old`. Through lines 31-73, the equilibrium radius of the bubble is calculated based on the equation 1.14 for 2000 iterations and with minimum error of 1.0E-06. In the last lines, the obtained equilibrium radius is stored in `R0_` and `pG0_` is calculated using equation 1.15, followed by initialization of `pG_` and `pB_` member data.

3.5 solidParticleCloud.H

As mentioned before, some of the flow properties or constant parameters related to the new member functions / member data of `solidParticle` are obtained / defined in the `solidParticleCloud` class. Therefore 4 new private member data need to be added in the `solidParticleCloud` class; these data are declared first in `solidParticleCloud.H` after declaration of `tInjEnd_` as

```
scalar p0_;
scalar pSat_;
scalar gamma_;
scalar sigma_;
```

`p0_` is the surrounding pressure in the equilibrium condition, `pSat_` is the liquid saturation pressure, `gamma_` is the used as the polytropic compression constant (equation 1.13) and `sigma` is the liquid

surface tension coefficient. Also, some access functions for these parameters should be declared in the public member functions as

```
inline scalar p0() const;
inline scalar pSat() const;
inline scalar gamma() const;
inline scalar sigma() const;
```

3.6 solidParticleCloud.C

The new member data of `solidParticleCloud` should be set in the class constructor after the initialization of `tInjEnd_` as

```
p0_(dimensionedScalar(particleProperties_.lookup("p0")).value()),
pSat_(dimensionedScalar(particleProperties_.lookup("pSat")).value()),
gamma_(dimensionedScalar(particleProperties_.lookup("gamma")).value()),
sigma_(dimensionedScalar(particleProperties_.lookup("sigma")).value())
```

As seen, all of the parameters need to be set in the `particleProperties` dictionary in the `constant` folder.
As stated before, when a new bubble is injected, some of the bubble parameters are initialized by the `solidParticle` constructor and some of them are initialized using the `initialEquilibriumRadius` function. Therefore, in the `inject` function, this function should be called after adding of each particle. So, after defining `ptr1` and before the command `Cloud<solidParticle>::addParticle(ptr1);` the following command should be added

```
ptr1->initialEquilibriumRadius(td);
```

Also, after defining `ptr2`, and before the command `Cloud<solidParticle>::addParticle(ptr2);`, the following command should be added

```
ptr2->initialEquilibriumRadius(td);
```

In the move function of `solidParticleCloud`, after defining `const volScalarField& alphaW`, the following definition should be added

```
const volScalarField& p = mesh_.lookupObject<const volScalarField>("p");
```

Then, after the command `interpolationCell<scalar> alphaWInterp(alphaW);`), flow pressure interpolation, `p`, should be defined as

```
interpolationCellPoint<scalar> pInterp(p);
```

Also, in the creation of `solidParticle::trackingData` object, add this parameter in the argument list

```
solidParticle::trackingData
        td(*this, rhoInterp, UInterp, nuInterp, alphaWInterp, pInterp, g.value());
```

3.7 solidParticleCloudI.H

The last step is to define the new inline access function of `solidParticleCloud` as

```
inline Foam::scalar Foam::solidParticleCloud::p0() const
{
    return p0_;
}
```

```
inline Foam::scalar Foam::solidParticleCloud::pSat() const
{
    return pSat_;
}

inline Foam::scalar Foam::solidParticleCloud::gamma() const
{
    return gamma_;
}

inline Foam::scalar Foam::solidParticleCloud::sigma() const
{
    return sigma_;
}
```

Finally, in the `Make/files` file, change the name of the solver to `myLPTVOF_RP`

Chapter 4

Sample Problem

In this chapter the improved solver is utilized to simulate the diameter variation of a single bubble in a 2D pressure driven channel flow.

4.1 Test Case Description

The boundaries of the flow field and the initial bubble state are shown in figure 4.1.

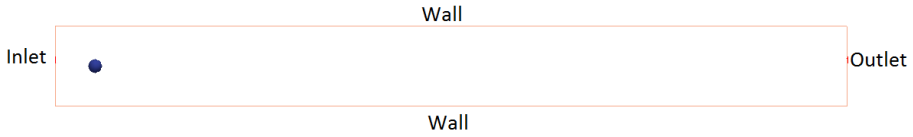

Figure 4.1: Bubble in 2D flow.

The channel has 10m*1m dimensions. The structure of the case directory is

```
channelRP
   |-- 0
   |   |-- alpha.water
   |   |-- p_rgh
   |   |-- U
   |-- constant
   |   |-- g
   |   |-- particleProperties
   |   |-- polyMesh
   |   |-- transportProperties
   |   |-- turbulenceProperties
   |-- system
       |-- blockMeshDict
       |-- controlDict
       |-- fvSchemes
       |-- fvSolution
```

The inlet and outlet pressures are 270 kPa and 70 kPa respectively and zero-gradient condition is set for pressure at walls. The velocity has zero-gradient condition on both inlet and outlet and it is set to zero on the walls. Also, the boundary condition for volume fraction (*alpha.water*) is zero-gradient everywhere. The bubble has an initial diameter of 0.4 mm. The `particleProperties` dictionary of the simulation case is specified as

```
/*--------------------------------*- C++ -*----------------------------------*\
```

```
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.5.x                                 |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      particleProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

rhop rhop [ 1 -3  0  0  0  0  0] 0.01389;
e    e    [ 0  0  0  0  0  0  0] 0.2;
mu   mu   [ 0  0  0  0  0  0  0] 0.05;
posP1 posP1 [ 0  1  0  0  0  0  0] (0.5 0.5 0.0);
dP1 dP1 [ 0  1  0  0  0  0  0] 0.0004;
UP1 UP1 [ 0  1 -1   0  0  0  0] (0 0 0);
tInjStart tInjStart [ 0 0   1  0  0  0  0] 0.025;
tInjEnd tInjEnd [ 0 0   1  0  0  0  0] 0.035;
pSat             pSat       [1 -1 -2 0 0]    2000;
sigma           sigma       [1 0 -2 0 0 0 0] 0.07;
gamma           gamma       [0 0 0 0 0 0 0] 1.4;
p0             p0       [1 -1 -2 0 0]    100000;

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

The initial volume fraction is set equal to 1 throughout the main. Therefore, the fluid is water with a density of 1000 kg/m^3 and a kinematic viscosity of 1E-06. The timestep size was set equal to 0.01 sec and the flow field was solved for 0.65 sec (after that the bubble will exit the domain). The remaining settings are similar to general cases of `interFoam` solver which can be found in the tutorial directory of OpenFOAM (e.g. dambreak problem).

4.2 Adapting the solver for the test case

Since in this case we only simulate 1 bubble particle, the properties of 1 particle are included in the `particleProperties` dictionary. Therefore, in the main solver, the related commands for the injection of the second particle should be omitted in the `inject` before compiling the solver. Also, since the bubble is completely surrounded by water, the `LPTtoVOF.H` file should not be included in the solver; otherwise the bubble will be transferred to an Eulerian structure when it is injected. Therefore the command `#include "LPTtoVOF.H"` should be omitted at the end of the `solidParticle::move` function before compiling the solver.

To start the simulation one may add the accompanying case file (which can be downloaded form the course homepage) to the user run directory. Then in the case directory, type the following command in the terminal:

```
myLPTVOF_RP > log&
```

4.3 Result

To visualize both the Lagrangian bubble and the Eulerian fields at the same time, you need to open two cases in the paraview. The bubble is injected at time `t= 0.03`. Since there is no Lagrangian particle data in the previous time folders, one may get some warning or errors in paraview. Therefore, it is better to rename the time folders before this time so that they are not seen when the case is opened in paraview. To postprocess the results, one may follow the guidelines below

```
- Terminal window: touch channelRP.foam
- Paraview window: File -> Open -> channelRP.foam
```

Perform the usual steps to visualize the pressure field in the channel

```
- In the same paraview window: File -> Open -> channelRP.foam
In the Mesh regions: lagrangian/DefaultCloud
Create a Glyph from the second case: Type Sphere, Scalars d,
Mode Scalar, Edit Set Scalar Factor 1000
```

The scalar factor is set to 1000 to make the bubble easier to visualize as it is much smaller the the channel dimensions. In Figure 4.2 the motion of the bubble inside the channel is depicted for five different timesteps. The increase of the bubble size during its travel downstream due to pressure decrease can be seen in the figure. The initial diameter of the bubble is 0.4mm and its final diameter is about 0.55 mm.

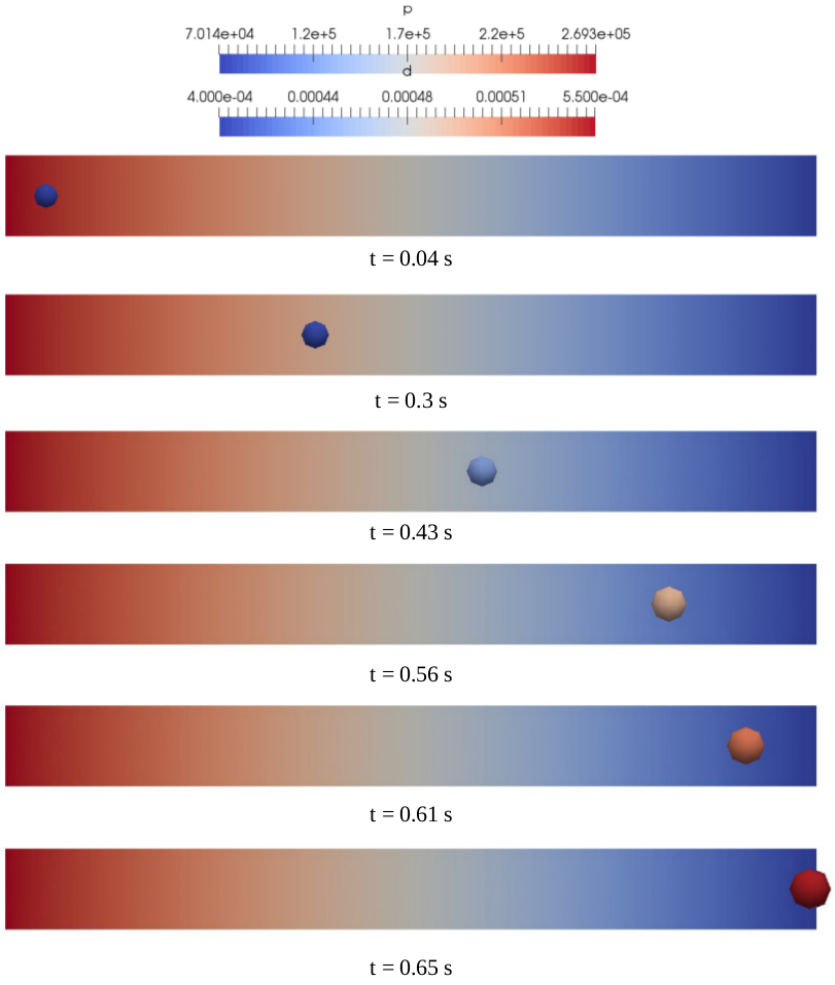

Figure 4.2: Bubble size variation with pressure change in 2D channel flow

Bibliography

[Vallier (2011)] A. Vallier (2011). *Coupling of VOF with LPT in OpenFOAM.* Webpage: `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2011/OF_kurs_LPT_120911.pdf`

[Vallier (2013)] A. Vallier (2013). *Simulations of cavitation from the large vapour structures to the small bubble dynamics* PhD Thesis, Lund University

[Shampine and Reichelt (1997)] L. Shampine and M. Reichelt (1997). *The Matlab ODE suite.* J. Sci. Comput., 18(1), 1-22.

[Franc and Michel (2004)] J.-P. France and J.-M. Michel (ed.) (2012). *Fundamentals of Cavitation.* Kluwer Academic Publishers.

Study Questions

1. What is the main purpose of implementing the Rayleigh-Plesset equation in the VOF-LPT solver?
2. What is the bubble inside pressure composed of? Describe different terms.
3. How is the interface curvature (κ) calculated in the VOF-LPT solver? And what is it used for?
4. Why do we need to adapt `inject` function for new versions of OpenFOAM?
5. What method is employed to solve Rayleigh-Plesset equation numerically?
6. In calculating the dissolved gas pressure (P_g) from the equilibrium state (P_{g0} and R_0) in equation (1.13), when do we set k = 1.4 or 1.0?
7. What is the purpose of defining the `initialEquilibriumRadius` function?
8. Why do we need to omit the `#include "LPTtoVOF.H"` command in the `solidParticle.C` file for our specific test case?

Appendix A

Original Solver Files

A.1 interFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

Application
    interFoam

Description
    Solver for 2 incompressible, isothermal immiscible fluids using a VOF
    (volume of fluid) phase-fraction based interface capturing approach.

    The momentum and other fluid properties are of the "mixture" and a single
    momentum equation is solved.

    Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected.

    For a two-fluid approach see twoPhaseEulerFoam.
```

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

#include "fvCFD.H"
#include "CMULES.H"
#include "EulerDdtScheme.H"
#include "localEulerDdtScheme.H"
#include "CrankNicolsonDdtScheme.H"
#include "subCycle.H"
#include "immiscibleIncompressibleTwoPhaseMixture.H"
#include "solidParticleCloud.H"
#include "turbulentTransportModel.H"
#include "pimpleControl.H"
#include "fvIOoptionList.H"
#include "CorrectPhi.H"
#include "fixedFluxPressureFvPatchScalarField.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"

    solidParticleCloud particles(mesh);

    pimpleControl pimple(mesh);

    #include "createTimeControls.H"
    #include "createRDeltaT.H"
    #include "initContinuityErrs.H"
    #include "createFields.H"
    #include "createMRF.H"
    #include "createFvOptions.H"
    #include "correctPhi.H"


    if (!LTS)
    {
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "setInitialDeltaT.H"
    }

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

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

    while (runTime.run())
    {
        #include "readTimeControls.H"
```

```
    if (LTS)
    {
        #include "setRDeltaT.H"
    }
    else
    {
        #include "CourantNo.H"
        #include "alphaCourantNo.H"
        #include "setDeltaT.H"
    }

    runTime++;

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



    // --- Pressure-velocity PIMPLE corrector loop
    while (pimple.loop())
    {
        #include "alphaControls.H"
        #include "alphaEqnSubCycle.H"

        mixture.correct();

        #include "UEqn.H"

        // --- Pressure corrector loop
        while (pimple.correct())
        {
            #include "pEqn.H"
        }

        if (pimple.turbCorr())
        {
            turbulence->correct();
        }
    }

    particles.move(g);
    Info<< "Cloud size= "<< particles.size() <<endl;

    particles.checkCo();  // 4Way Coupling
    alpha1 += particles.AddTOAlpha();  //LPT2VOF
    U += particles.AddToU();    //LPT2VOF

    runTime.write();

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

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

```
    return 0;
}


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

A.2 solidParticleCloud.H

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

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

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

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

Class
    Foam::solidParticleCloud

Description
    A Cloud of solid particles

SourceFiles
    solidParticleCloudI.H
    solidParticleCloud.C
    solidParticleCloudIO.C

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

#ifndef solidParticleCloud_H
#define solidParticleCloud_H

#include "Cloud.H"
#include "solidParticle.H"
#include "IOdictionary.H"

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

namespace Foam
{

// Forward declaration of classes
class fvMesh;

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

```
                        Class solidParticleCloud Declaration
\*---------------------------------------------------------------------------*/

class solidParticleCloud
:
    public Cloud<solidParticle>
{
    // Private data

        const fvMesh& mesh_;

        IOdictionary particleProperties_;

        scalar rhop_;
        scalar e_;
        scalar mu_;
        vectorField smom_;
        scalarField correctalphaW_;
        vectorField correctU_;
        vector posP1_;
        scalar dP1_;
        vector UP1_;
        vector posP2_;
        scalar dP2_;
        vector UP2_;
        scalar tInjStart_;
        scalar tInjEnd_;


    // Private Member Functions

        //- Disallow default bitwise copy construct
        solidParticleCloud(const solidParticleCloud&);

        //- Disallow default bitwise assignment
        void operator=(const solidParticleCloud&);


public:

    // Constructors

        //- Construct given mesh
        solidParticleCloud
        (
            const fvMesh&,
            const word& cloudName = "defaultCloud",
            bool readFields = true
        );


    // Member Functions

        // Access
```

```
            virtual bool hasWallImpactDistance() const;

            inline const fvMesh& mesh() const;

            inline scalar rhop() const;
            inline scalar e() const;
            inline scalar mu() const;

            inline vectorField& smom();
            inline const vectorField& smom() const;
            inline tmp<volVectorField> momentumSource() const;
            inline scalarField& correctalphaW();
            inline const scalarField& correctalphaW() const;
            inline tmp<volScalarField> AddTOAlpha() const;
            inline vectorField& correctU();
            inline const vectorField& correctU() const;
            inline tmp<volVectorField> AddToU() const;


        // Edit

            //- Move the particles under the influence of the given
            //  gravitational acceleration
            void move(const dimensionedVector& g);
            //- Check if there will be collision and update velocities
            void checkCo();  // 4Way Coupling
            //- Inject particles according to the dictionnary particleProperties
            void inject(solidParticle::trackingData &td);      // Injection
};


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

} // End namespace Foam

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

#include "solidParticleCloudI.H"

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

#endif

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

A.3 solidParticleCloudI.H

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

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

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

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

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

namespace Foam
{

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

inline const Foam::fvMesh& Foam::solidParticleCloud::mesh() const
{
    return mesh_;
}


inline Foam::scalar Foam::solidParticleCloud::rhop() const
{
    return rhop_;
}


inline Foam::scalar Foam::solidParticleCloud::e() const
{
    return e_;
}


inline Foam::scalar Foam::solidParticleCloud::mu() const
{
    return mu_;
}
```

```
// 2Way Coupling- Momentum Source:
inline tmp<volVectorField> solidParticleCloud::momentumSource() const
{
tmp<volVectorField> tsource
(
    new volVectorField
    (
       IOobject
       (
          "smom",
          mesh_.time().timeName(),
          mesh_,
          IOobject::NO_READ,
          IOobject::NO_WRITE
       ),
       mesh_,
       dimensionedVector
          (
             "zero",
             dimensionSet(1, -2, -2, 0, 0),
             vector::zero
          )
    )
   );
   tsource().internalField() = - smom_/(mesh_.time().deltaT().value()*mesh_.V());

   return tsource;
}

inline Foam::vectorField& Foam::solidParticleCloud::smom()
{
    return smom_;
}

inline const Foam::vectorField& Foam::solidParticleCloud::smom() const
{
    return smom_;
}


inline tmp<volScalarField> solidParticleCloud::AddTOAlpha() const
{
tmp<volScalarField> alphasource
(
new volScalarField
(
IOobject
(
"correctalphaW",
mesh_.time().timeName(),
mesh_,
IOobject::NO_READ,
```

```
IOobject::NO_WRITE
),
mesh_,
dimensionedScalar
(
"zero",
dimensionSet(0,0 , 0, 0, 0),
0
)
)
);
alphasource().internalField() = correctalphaW_;
return alphasource;
}


inline tmp<volVectorField> solidParticleCloud::AddToU() const
{
tmp<volVectorField> Usource
(
new volVectorField
(
IOobject
(
"correctU",
mesh_.time().timeName(),
mesh_,
IOobject::NO_READ,
IOobject::NO_WRITE
),
mesh_,
dimensionedVector
(
"correctU",
dimensionSet(0, 1, -1, 0, 0),
vector(0,0,0)
)
)
);
Usource().internalField() = correctU_;
return Usource;
}


inline Foam::scalarField& Foam::solidParticleCloud::correctalphaW()
{
return correctalphaW_;
}

inline const Foam::scalarField& Foam::solidParticleCloud::correctalphaW() const
{
return correctalphaW_;
}
```

```
inline Foam::vectorField& Foam::solidParticleCloud::correctU()
{
return correctU_;
}

inline const Foam::vectorField& Foam::solidParticleCloud::correctU() const
{
return correctU_;
}

}   // End namespace Foam
// ************************************************************************* //
```

A.4 solidParticleCloud.C

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

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

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

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

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

#include "solidParticleCloud.H"
#include "fvMesh.H"
#include "volFields.H"
#include "interpolationCellPoint.H"

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

Foam::solidParticleCloud::solidParticleCloud
(
    const fvMesh& mesh,
    const word& cloudName,
    bool readFields
)
:
    Cloud<solidParticle>(mesh, cloudName, false),
    mesh_(mesh),
    particleProperties_
    (
        IOobject
        (
            "particleProperties",
            mesh_.time().constant(),
            mesh_,
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE
        )
    ),
```

```
    rhop_(dimensionedScalar(particleProperties_.lookup("rhop")).value()),
    e_(dimensionedScalar(particleProperties_.lookup("e")).value()),
    mu_(dimensionedScalar(particleProperties_.lookup("mu")).value()),
    smom_(mesh_.nCells(), vector::zero),
    correctalphaW_(mesh_.nCells(), 0),
    correctU_(mesh_.nCells(), vector::zero),
    posP1_(dimensionedVector(particleProperties_.lookup("posP1")).value()),
    dP1_(dimensionedScalar(particleProperties_.lookup("dP1")).value()),
    UP1_(dimensionedVector(particleProperties_.lookup("UP1")).value()),
    posP2_(dimensionedVector(particleProperties_.lookup("posP2")).value()),
    dP2_(dimensionedScalar(particleProperties_.lookup("dP2")).value()),
    UP2_(dimensionedVector(particleProperties_.lookup("UP2")).value()),
    tInjStart_(dimensionedScalar(particleProperties_.lookup("tInjStart")).value()),
    tInjEnd_(dimensionedScalar(particleProperties_.lookup("tInjEnd")).value())
{
    if (readFields)
    {
        solidParticle::readFields(*this);
    }
}


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


// Inject Function:
void Foam::solidParticleCloud::inject(solidParticle::trackingData &td)
{
label cellI=1;
    label tetFaceI=1;
    label tetPtI=1;
    mesh_.findCellFacePt(td.cloud().posP1_, cellI, tetFaceI, tetPtI);

    solidParticle* ptr1 = new solidParticle(mesh_, td.cloud().posP1_, cellI,
     tetFaceI, tetPtI,td.cloud().dP1_, td.cloud().UP1_);
    Cloud<solidParticle>::addParticle(ptr1);

    mesh_.findCellFacePt(td.cloud().posP2_, cellI, tetFaceI, tetPtI);

    solidParticle* ptr2 = new solidParticle(mesh_, td.cloud().posP2_, cellI,
     tetFaceI, tetPtI, td.cloud().dP2_, td.cloud().UP2_);
    Cloud<solidParticle>::addParticle(ptr2);
}

bool Foam::solidParticleCloud::hasWallImpactDistance() const
{
    return true;
}


void Foam::solidParticleCloud::move(const dimensionedVector& g)
{
    const volScalarField& rho = mesh_.lookupObject<const volScalarField>("rho");
    const volVectorField& U = mesh_.lookupObject<const volVectorField>("U");
```

```
    const volScalarField& nu = mesh_.lookupObject<const volScalarField>("nu");
    const volScalarField& alphaW =
    mesh_.lookupObject<const volScalarField>("alpha.water");

    interpolationCellPoint<scalar> rhoInterp(rho);
    interpolationCellPoint<vector> UInterp(U);
    interpolationCellPoint<scalar> nuInterp(nu);
    interpolationCell<scalar> alphaWInterp(alphaW);

    smom_ = vector::zero;
    correctalphaW_=0;
    correctU_=vector::zero;

    solidParticle::trackingData
        td(*this, rhoInterp, UInterp, nuInterp, alphaWInterp, g.value());

    Cloud<solidParticle>::move(td, mesh_.time().deltaTValue());

    // Inject
    if(mesh_.time().value()> td.cloud().tInjStart_ &&
     mesh_.time().value()< td.cloud().tInjEnd_)
    {
       this->inject(td);
    }
}


void Foam::solidParticleCloud::checkCo()
{
if ((*this).size() < 2)
{
return;
}

Cloud<solidParticle>::iterator secondP = (*this).begin();
++secondP;
Cloud<solidParticle>::iterator p1 = secondP;
while (p1 != (*this).end())
{
Cloud<solidParticle>::iterator p2 = (*this).begin();
while (p2 != p1)
{
bool collision=false;
#include "myCM.H"
if (collision)
{
Info<< "Velocities before collision: "<< p1().U() <<p2().U()<<endl;

vector p = p2().position() - p1().position();
vector n12=p/mag(p);
scalar v1n=p1().U() & n12;
vector v1t=p1().U() - v1n*n12 ;
scalar v2n=p2().U() & n12;
vector v2t=p2().U() - v2n*n12 ;
```

```
scalar COR=0.8;
scalar m1 = rhop()*constant::mathematical::pi / 6.0*pow(p1().d(),3);
scalar m2 = rhop()*constant::mathematical::pi / 6.0*pow(p2().d(),3);
scalar mrn = m1*v1n + m2*v2n;
scalar vnRel = v1n - v2n;
p1().U_ = v1t+ ((mrn - COR* m2*vnRel)/(m1+m2))*n12;
p2().U_ = v2t+ ((mrn + COR* m1*vnRel)/(m1+m2))*n12;

Info<< "new velocities "<< p1().U() << " and "<<p2().U()<<endl;

}
++p2;
} // end - inner loop
++p1;
} // end - outer loop
}


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

A.5 solidParticle.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

Class
    Foam::solidParticle

Description
    Simple solid spherical particle class with one-way coupling with the
    continuous phase.

SourceFiles
    solidParticleI.H
    solidParticle.C
    solidParticleIO.C

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

#ifndef solidParticle_H
#define solidParticle_H

#include "particle.H"
#include "IOstream.H"
#include "autoPtr.H"
#include "interpolationCellPoint.H"
#include "interpolationCell.H"   //LPT2VOF
#include "contiguous.H"

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

namespace Foam
{
```

```
class solidParticleCloud;

/*---------------------------------------------------------------------------*\
                          Class solidParticle Declaration
\*---------------------------------------------------------------------------*/

class solidParticle
:
    public particle
{
    // Private data

        //- Size in bytes of the fields
        static const std::size_t sizeofFields_;

        //- Diameter
        scalar d_;

        //- Velocity of parcel
        vector U_;


public:

    friend class Cloud<solidParticle>;
    friend class solidParticleCloud;

    //- Class used to pass tracking data to the trackToFace function
    class trackingData
    :
        public particle::TrackingData<solidParticleCloud>
    {
        // Interpolators for continuous phase fields

            const interpolationCellPoint<scalar>& rhoInterp_;
            const interpolationCellPoint<vector>& UInterp_;
            const interpolationCellPoint<scalar>& nuInterp_;
            const interpolationCell<scalar>& alphaWInterp_;

        //- Local gravitational or other body-force acceleration
        const vector& g_;


    public:

        // Constructors

            inline trackingData
            (
                solidParticleCloud& spc,
                const interpolationCellPoint<scalar>& rhoInterp,
                const interpolationCellPoint<vector>& UInterp,
                const interpolationCellPoint<scalar>& nuInterp,
                const interpolationCell<scalar>& alphaWInterp,
```

```
            const vector& g
        );


    // Member functions

        inline const interpolationCellPoint<scalar>& rhoInterp() const;

        inline const interpolationCellPoint<vector>& UInterp() const;

        inline const interpolationCellPoint<scalar>& nuInterp() const;

        inline const interpolationCell<scalar>& alphaWInterp() const;

        inline const vector& g() const;
};


// Constructors

    //- Construct from components
    inline solidParticle
    (
        const polyMesh& mesh,
        const vector& position,
        const label cellI,
        const label tetFaceI,
        const label tetPtI,
        const scalar d,
        const vector& U
    );

    //- Construct from Istream
    solidParticle
    (
        const polyMesh& mesh,
        Istream& is,
        bool readFields = true
    );

    //- Construct and return a clone
    virtual autoPtr<particle> clone() const
    {
        return autoPtr<particle>(new solidParticle(*this));
    }

    //- Factory class to read-construct particles used for
    //  parallel transfer
    class iNew
    {
        const polyMesh& mesh_;

    public:
```

```
            iNew(const polyMesh& mesh)
            :
                mesh_(mesh)
            {}

            autoPtr<solidParticle> operator()(Istream& is) const
            {
                return autoPtr<solidParticle>
                (
                    new solidParticle(mesh_, is, true)
                );
            }
        };


    // Member Functions

        // Access

            //- Return diameter
            inline scalar d() const;

            //- Return velocity
            inline const vector& U() const;


        // Tracking

            //- Move
            bool move(trackingData&, const scalar);


        // Patch interactions

            //- Overridable function to handle the particle hitting a patch
            //  Executed before other patch-hitting functions
            bool hitPatch
            (
                const polyPatch&,
                trackingData& td,
                const label patchI,
                const scalar trackFraction,
                const tetIndices& tetIs
            );

            //- Overridable function to handle the particle hitting a
            //  processorPatch
            void hitProcessorPatch
            (
                const processorPolyPatch&,
                trackingData& td
            );

            //- Overridable function to handle the particle hitting a wallPatch
```

```
            void hitWallPatch
            (
                const wallPolyPatch&,
                trackingData& td,
                const tetIndices&
            );

            //- Overridable function to handle the particle hitting a polyPatch
            void hitPatch
            (
                const polyPatch&,
                trackingData& td
            );

            //- Transform the physical properties of the particle
            //  according to the given transformation tensor
            virtual void transformProperties(const tensor& T);

            //- Transform the physical properties of the particle
            //  according to the given separation vector
            virtual void transformProperties(const vector& separation);

            //- The nearest distance to a wall that
            //  the particle can be in the n direction
            virtual scalar wallImpactDistance(const vector& n) const;


    // I-O

        static void readFields(Cloud<solidParticle>& c);

        static void writeFields(const Cloud<solidParticle>& c);


    // Ostream Operator

        friend Ostream& operator<<(Ostream&, const solidParticle&);
};


template<>
inline bool contiguous<solidParticle>()
{
    return true;
}


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

} // End namespace Foam

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

#include "solidParticleI.H"
```

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

#endif

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

A.6 solidParticleI.H

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

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

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

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

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

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

inline Foam::solidParticle::trackingData::trackingData
(
    solidParticleCloud& spc,
    const interpolationCellPoint<scalar>& rhoInterp,
    const interpolationCellPoint<vector>& UInterp,
    const interpolationCellPoint<scalar>& nuInterp,
    const interpolationCell<scalar>& alphaWInterp,
    const vector& g
)
:
    particle::TrackingData<solidParticleCloud>(spc),
    rhoInterp_(rhoInterp),
    UInterp_(UInterp),
    nuInterp_(nuInterp),
    alphaWInterp_(alphaWInterp),
    g_(g)
{}


inline Foam::solidParticle::solidParticle
(
    const polyMesh& mesh,
    const vector& position,
    const label cellI,
    const label tetFaceI,
```

```
    const label tetPtI,
    const scalar d,
    const vector& U
)
:
    particle(mesh, position, cellI, tetFaceI, tetPtI),
    d_(d),
    U_(U)
{}


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

inline const Foam::interpolationCellPoint<Foam::scalar>&
Foam::solidParticle::trackingData::rhoInterp() const
{
    return rhoInterp_;
}


inline const Foam::interpolationCellPoint<Foam::vector>&
Foam::solidParticle::trackingData::UInterp() const
{
    return UInterp_;
}


inline const Foam::interpolationCellPoint<Foam::scalar>&
Foam::solidParticle::trackingData::nuInterp() const
{
    return nuInterp_;
}

inline const Foam::vector& Foam::solidParticle::trackingData::g() const
{
    return g_;
}


inline Foam::scalar Foam::solidParticle::d() const
{
    return d_;
}


inline const Foam::vector& Foam::solidParticle::U() const
{
    return U_;
}

inline const Foam::interpolationCell<Foam::scalar>&
Foam::solidParticle::trackingData::alphaWInterp() const
{
return alphaWInterp_;
```

```
}

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

A.7 solidParticle.C

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

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

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

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

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

#include "solidParticleCloud.H"

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

namespace Foam
{
    defineTemplateTypeNameAndDebug(Cloud<solidParticle>, 0);
}

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

bool Foam::solidParticle::move
(
    trackingData& td,
    const scalar trackTime
)
{
    td.switchProcessor = false;
    td.keepParticle = true;

    const polyBoundaryMesh& pbMesh = mesh_.boundaryMesh();

    scalar tEnd = (1.0 - stepFraction())*trackTime;
    scalar dtMax = tEnd;

    while (td.keepParticle && !td.switchProcessor && tEnd > SMALL)
    {
```

```
if (debug)
{
    Info<< "Time = " << mesh_.time().timeName()
        << " trackTime = " << trackTime
        << " tEnd = " << tEnd
        << " steptFraction() = " << stepFraction() << endl;
}

// set the lagrangian time-step
scalar dt = min(dtMax, tEnd);

// remember which cell the parcel is in
// since this will change if a face is hit
label cellI = cell();

dt *= trackToFace(position() + dt*U_, td);

tEnd -= dt;
stepFraction() = 1.0 - tEnd/trackTime;

cellPointWeight cpw(mesh_, position(), cellI, face());
scalar rhoc = td.rhoInterp().interpolate(cpw);
vector Uc = td.UInterp().interpolate(cpw);
scalar nuc = td.nuInterp().interpolate(cpw);

scalar rhop = td.cloud().rhop();
scalar magUr = mag(Uc - U_);

scalar ReFunc = 1.0;
scalar Re = magUr*d_/nuc;

if (Re > 0.01)
{
    ReFunc += 0.15*pow(Re, 0.687);
}

scalar Dc = (24.0*nuc/d_)*ReFunc*(3.0/4.0)*(rhoc/(d_*rhop));

scalar m = rhop*(4.0/3.0)*constant::mathematical::pi*pow(d_/2.0,3);
vector oldMom = U_*m;

U_ = (U_ + dt*(Dc*Uc + (1.0 - rhoc/rhop)*td.g()))/(1.0 + dt*Dc);

vector newMom = U_*m;
td.cloud().smom()[cellI] += newMom-oldMom;

if (onBoundary() && td.keepParticle)
{
    if (isA<processorPolyPatch>(pbMesh[patch(face())]))
    {
        td.switchProcessor = true;
    }
}
#include "LPTtoVOF.H"
```

```
    }

    return td.keepParticle;
}


bool Foam::solidParticle::hitPatch
(
    const polyPatch&,
    trackingData&,
    const label,
    const scalar,
    const tetIndices&
)
{
    return false;
}


void Foam::solidParticle::hitProcessorPatch
(
    const processorPolyPatch&,
    trackingData& td
)
{
    td.switchProcessor = true;
}


void Foam::solidParticle::hitWallPatch
(
    const wallPolyPatch& wpp,
    trackingData& td,
    const tetIndices& tetIs
)
{
    vector nw = tetIs.faceTri(mesh_).normal();
    nw /= mag(nw);

    scalar Un = U_ & nw;
    vector Ut = U_ - Un*nw;

    if (Un > 0)
    {
        U_ -= (1.0 + td.cloud().e())*Un*nw;
    }

    U_ -= td.cloud().mu()*Ut;
}


void Foam::solidParticle::hitPatch
(
    const polyPatch&,
```

```
    trackingData& td
)
{
    td.keepParticle = false;
}


void Foam::solidParticle::transformProperties (const tensor& T)
{
    particle::transformProperties(T);
    U_ = transform(T, U_);
}


void Foam::solidParticle::transformProperties(const vector& separation)
{
    particle::transformProperties(separation);
}


Foam::scalar Foam::solidParticle::wallImpactDistance(const vector&) const
{
    return 0.5*d_;
}


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

A.8 solidParticleIO.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "solidParticle.H"
#include "IOstreams.H"

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

const std::size_t Foam::solidParticle::sizeofFields_
(
    sizeof(solidParticle) - sizeof(particle)
);


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

Foam::solidParticle::solidParticle
(
    const polyMesh& mesh,
    Istream& is,
    bool readFields
)
:
    particle(mesh, is, readFields)
{
    if (readFields)
    {
        if (is.format() == IOstream::ASCII)
        {
            d_ = readScalar(is);
```

```
            is >> U_;
        }
        else
        {
            is.read(reinterpret_cast<char*>(&d_), sizeofFields_);
        }
    }

    // Check state of Istream
    is.check("solidParticle::solidParticle(Istream&)");
}


void Foam::solidParticle::readFields(Cloud<solidParticle>& c)
{
    if (!c.size())
    {
        return;
    }

    particle::readFields(c);

    IOField<scalar> d(c.fieldIOobject("d", IOobject::MUST_READ));
    c.checkFieldIOobject(c, d);

    IOField<vector> U(c.fieldIOobject("U", IOobject::MUST_READ));
    c.checkFieldIOobject(c, U);

    label i = 0;
    forAllIter(Cloud<solidParticle>, c, iter)
    {
        solidParticle& p = iter();

        p.d_ = d[i];
        p.U_ = U[i];
        i++;
    }
}


void Foam::solidParticle::writeFields(const Cloud<solidParticle>& c)
{
    particle::writeFields(c);

    label np = c.size();

    IOField<scalar> d(c.fieldIOobject("d", IOobject::NO_READ), np);
    IOField<vector> U(c.fieldIOobject("U", IOobject::NO_READ), np);

    label i = 0;
    forAllConstIter(Cloud<solidParticle>, c, iter)
    {
        const solidParticle& p = iter();
```

```
        d[i] = p.d_;
        U[i] = p.U_;
        i++;
    }

    d.write();
    U.write();
}


// * * * * * * * * * * * * * * * * IOstream Operators  * * * * * * * * * * * * * //

Foam::Ostream& Foam::operator<<(Ostream& os, const solidParticle& p)
{
    if (os.format() == IOstream::ASCII)
    {
        os  << static_cast<const particle&>(p)
            << token::SPACE << p.d_
            << token::SPACE << p.U_;
    }
    else
    {
        os  << static_cast<const particle&>(p);
        os.write
        (
            reinterpret_cast<const char*>(&p.d_),
            solidParticle::sizeofFields_
        );
    }

    // Check state of Ostream
    os.check("Ostream& operator<<(Ostream&, const solidParticle&)");

    return os;
}


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