CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

LPT for erosion modeling in OpenFOAM

Differences between solidParticle and kinematicParcel, and how to add erosion modeling

---

Developed for OpenFOAM-2.2.x

Author:
Alejandro López

Peer reviewed by:
Abolfazl Asnagi
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.

February 4, 2014

Chapter 1

Theoretical Background

1.1 Introduction

The aim of this tutorial is to effectively describe the available possibilities in OpenFOAM to simulate lagrangian inert particles and the different classes used for their modelling. The tutorial also tries to give a description of how to create an incompressible multiphase solver and how to pre-process, run and post-process a case involving an incompressible flow with inert Lagrangian particles in a three-dimensional domain.

1.2 Lagrangian Particle Tracking

When dealing with the movement of a group of particles inside a fluid, there are basically two different ways to approach the problem. In the Eulerian-Eulerian models, the particles are treated as a continuous phase and conservation equations are solved for the particulate phase. This method is suitable for large particle concentrations, where two-way coupling between the fluid and the particulate phases as well as particle-particle collisions are important. On the other hand, in the Eulerian-Lagrangian approach, the Eulerian continuum equations are solved for the fluid phase, while Newton's equations for motion are solved for the particulate phase in order to determine the trajectories of the particles (or groups of particles). The trajectories are obtained once the following equation for the particles has been solved:

$$m_p \frac{d\mathbf{u}_p}{dt} = \mathbf{F}_p \tag{1.1}$$

Once the force has been calculated, the trajectories are calculated by means of integration of the particle velocity:

$$\frac{dx_p}{dt} = \mathbf{u}_p \tag{1.2}$$

There are three different possibilities when constructing the equations to solve particle motion:

- One way coupling: particle influence on the fluid phase is neglected
- Two way coupling: the force the particles exert on the fluid is no longer neglected
- Four way coupling: also particle-particle collisions are taken into acount

For a more accurate selection of the correct approach, the particle mass loading, β, and the Stokes number, S_t might also be calculated [1].
The particle mass loading is expressed as follows:

$$\beta = \frac{\text{particulate mass per unit volume of flow}}{\text{fluid mass per unit volume of flow}} = \frac{r_p\rho_p}{r\rho} \tag{1.3}$$

Where r is a volume fraction and ρ is a density. Significant two-way coupling is expected for particle mass loadings greater than 0.2 and values greater than 0.6 indicate that particle collisions are likely in, at least, some parts of the domain.

The Stokes number defines the degree to which particle motion is tied to fluid motion:

$$S_t = \frac{\rho_p d_p^2 V_s}{18\mu L_s} \tag{1.4}$$

Where d_p is the particle diameter, μ is the dynamic viscosity of the fluid and V_s and L_s are the characteristic velocity and length scales of the flow. For values of $S_t > 2.0$ the flow will be dominated by particle-wall interactions, whereas for $S_t < 0.25$ the effect of particle-wall interactions is negligible and the particles are tightly coupled to the fluid through viscous drag. At $S_t < 0.05$ the particles and the fluid are strongly coupled, while for $S_t << 0.01$ the particles are expected to respond almost instantaneously to any changes in the fluid flow.

The force balance on a spherical particle inside a viscous fluid is written:

$$\mathbf{F}_p = m_p \frac{d\mathbf{V}_p}{dt} = \mathbf{F}_D + \mathbf{F}_P + \mathbf{F}_g + \mathbf{F}_A \tag{1.5}$$

The drag force on spherical particles is then calculated as:

$$\mathbf{F}_D = m_p \frac{18\mu}{\rho_p d_p^2} \frac{C_D Re(Re)}{24} (\mathbf{u} - \mathbf{u}_p) \tag{1.6}$$

Where the drag coefficient C_D is obtained from the following equation:

$$C_D = \begin{cases} \frac{24}{Re} & \text{if } Re_p < 1 \\ \frac{24}{Re_p}(1 + 0.15Re_p^{0.687}) & \text{if } 1 \leq Re_p \leq 1000 \\ 0.44 & \text{if } Re_p > 1000 \end{cases}$$

The gravity and buoyancy force is:

$$\mathbf{F}_g = m_p \mathbf{g}(1 + \frac{\rho}{\rho_p}) \tag{1.7}$$

The pressure gradient force is:

$$\mathbf{F}_P = \frac{1}{6}\pi d_p^3 \nabla P \tag{1.8}$$

And the added mass force:

$$\mathbf{F}_A = \frac{1}{12}\pi d_p^3 \rho_p \frac{d\mathbf{V}_p}{dt} \tag{1.9}$$

1.3 Erosion

Throughout the years many authors have published a very large amount of papers and literature on erosion, having most of them developed their own equations for predicting erosive wear taking into account different approaches and factors that may influence erosion. One of the most important authors in erosion literature and responsible for one of the most commonly used equations for erosion prediction is Iain Finnie [4]. The equation developed by Finnie yields the volume of material, Q removed by a single abrasive grain of mass, m, and velocity, V.

$$Q = \frac{mV^2}{p\psi K}\left(sin2\alpha - \frac{6}{K}sin^2\alpha\right) \quad \text{if } \tan\alpha \leq \frac{K}{6} \tag{1.10}$$

$$Q = \frac{mV^2}{p\psi K}\left(\frac{Kcos^2\alpha}{6}\right) \quad \text{if } \tan\alpha > \frac{K}{6} \tag{1.11}$$

Where p is the plastic flow stress of the material being eroded, ψ is the ratio of the depth of contact to the depth of cut and K is the ratio of vertical to horizontal force components acting on the particle.

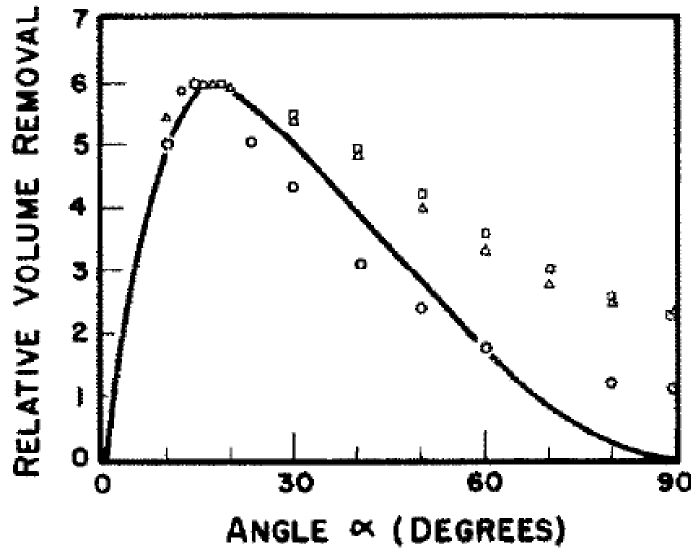

Figure 1.1: Predicted variation of volume removal with angle of impingement for a single abrasive grain. Experimental points for erosion by many grains (△ copper, □ SAE I020 steel, ∘ aluminium) are plotted so that the maximum erosion is the same in all cases.

Although this equation predicts no erosion for angles of impingement close to 90 degrees (figure 1.1), it serves its purpose as a first approach, and, implementing an additional term which takes into account particle rotation at impingement, erosion at normal angles can also be predicted [5].

Chapter 2

Implementation of LPT in OpenFOAM

2.1 Introduction

OpenFOAM provides the user with a number of possibilities in order to represent lagrangian particles, two of which are going to be commented here: The `/lagrangian/intermediate` library and the `/lagrangian/solidParticle` library. Some of the available examples and references are [13], [12] and [11], as well as the material at the course homepage [9].

2.2 SolidParticle Class

The `solidParticle` class enables the user to implement solid particles and to couple those to a given solver. Some examples on coupled solvers can be found in the references mentioned above. This library can be found by typing the following in the terminal window:

```
cd $WM_PROJECT_DIR/src/lagrangian/solidParticle
```

This folder contains the files `solidParticle.C` and `solidParticleCloud.C`, which define how the particles are implemented and their behaviour. The `solidParticle.C` starts with a definition of the `solidParticleCloud` class, which is, in fact, a templated Cloud of solid particles. The `Foam::solidParticle::move` function includes the implementation of the Reynolds number, needed for the calculation of the drag force and the new velocity. This new velocity will be affected by the parameters yielded by the operations performed on the eulerian phase. The carrier phase properties are represented inside the code by `rhoc` for the density,`Uc` for the velocity and `nuc` for the viscosity. Also some additional functions are implemented that determine what happens when a patch is hit by a particle and these are different depending on whether it is a processor patch or a wall patch for example. Regarding the file `solidParticleCloud.C`, the constructor for the cloud of solid particles is defined first. This constructor reads the properties from a dictionary called `particleProperties` and initially only the density of the particles (`rhop`) the restitution ratio (`e`) and the friction coefficient (`mu`) are required. In the examples and tutorials cited above, various modifications are implemented to this class such as the addition of an injector and modification of particle shape among others, for which additional properties need to be defined. Also a more specific definition of the class and the functions of its different members can be found in those references.

2.3 The intermediate library

The `lagrangian/intermediate` library in OpenFOAM consists of a series of models, forces and CloudFunctionObjects templated for each of the classes derived from the parcel class. The classes

available are the following ones:

- CollidingParcel
- ReactingParcel
- ReactingMultiphaseParcel
- ThermoParcel
- KinematicParcel

This report is mainly focused on the description of the KinematicParcel class and it will also try to briefly introduce the additional features of the CollidingParcel class.
Before making any changes in the code, it is highly recommended to recompile the files needed in the $WM_PROJECT_USER_DIR/src/lagrangian directory. In order to do this, we can run the following commands in the terminal window:

```
cd $WM_PROJECT_USER_DIR
mkdir -p src/lagrangian
cp -r $FOAM_SRC/lagrangian/intermediate $WM_PROJECT_USER_DIR/src/lagrangian
```

Once this is done, the necessary files will be copied into the user source directory and the next step would be to recompile them. It is highly reccommended to recompile only the necessary files into the user directory both to save disk space and to make the compilation faster. In this case, only the necessary files to compile both kinematic libraries (kinematicCloud and kinematicClollidingCLoud) are recompiled. This way, the user is able to run one, two or four-way coupled simulations of inert particles.

2.3.1 KinematicParcel Class

While in the solidParticle class the particles are tracked individually, in the KinematicParcel class, a set of particles or computational parcel is tracked. This construction is made because it is usually too expensive in computational terms to simulate all the real particles. In order to capture the behaviour of the real particles, some real case properties are defined. Thus, the main difference between the solidParticle class and the kinematicParcel class is that the solidparticle class contains no parcel treatment, but only real particles. However, both classes share some of their member functions and both are derived from the particle class and their clouds are both templates of the Cloud class. Nevertheless, the kinematicParcel class complexity lies far beyond the solidParticle class one.

2.3.2 KinematicCloudProperties dictionary

In order to set up the properties of the parcel as well as the additional submodels a dictionary called kinematicCloudProperties is needed. An example of such a file can be found on Appendix 1. Additional examples can also be found by typing in the terminal window the following commands:

```
run
find tutorials/ -name kinematicCloudProperties
```

By running this command, the search is made inside the tutorials folder within the run directory. If one wishes to find examples inside the $FOAM_TUTORIALS directory, the commands to run in the terminal window would be the following ones:

```
find $FOAM_TUTORIALS -name kinematicCloudProperties
```

and once the appropriate tutorial has been found, it can be copied to the run directory by typing:

```
cp -r desiredTutorial/ $FOAM_RUN
```

The properties of our Lagrangian particles are going to be defined within this dictionary, such as the injection model, the forces on the particles and the cloudFunctionObjects, which will enable the user to output erosion rates. The first dictionary entry (`solution`) consists of a series of switches, `sourceTerms`, `interpolationSchemes`, and `integrationSchemes` The `coupled` option can be set to true or false and the user may choose between transient or steady-state solution by switching the boolean `transient` to yes or no respectively.

```
template<class CloudType>
template<class TrackData>
void Foam::KinematicCloud<CloudType>::evolveCloud(TrackData& td)
{
    if (solution_.coupled())
    {
        td.cloud().resetSourceTerms();
    }

    if (solution_.transient())
    {
        label preInjectionSize = this->size();

        this->surfaceFilm().inject(td);

        // Update the cellOccupancy if the size of the cloud has changed
        // during the injection.
        if (preInjectionSize != this->size())
        {
            updateCellOccupancy();

            preInjectionSize = this->size();
        }

        injectors_.inject(td);


        // Assume that motion will update the cellOccupancy as necessary
        // before it is required.
        td.cloud().motion(td);
    }
    else
    {
//        this->surfaceFilm().injectSteadyState(td);

        injectors_.injectSteadyState(td, solution_.trackTime());

        td.part() = TrackData::tpLinearTrack;
        CloudType::move(td,  solution_.trackTime());
    }
}
```

As it can be seen in the code above, depending on whether we choose the solution to be coupled, uncoupled, transient or steady-state, the `evolveCloud` function will perform different operations. For instance, in case the solution is transient, the function

```
Foam::InjectionModel<CloudType>::inject(TrackData& td)
```

will inject the parcels. However, if it is a steady-state solution, the injection will be performed by the function

```
Foam::InjectionModel<CloudType>::injectSteadyState
```

In the file `InjectionModel.C`, which can be found in

```
$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/submodels/Kinematic/InjectionModel/InjectionModel
```

both functions are implemented and, as an example, one of the main differences between them has to do with the mass that the injector is going to introduce inside our computational domain. In a steady state case, the total mass to be injected is equal to the mass flow rate that the user specifies inside the `kinematicCloudProperties` dictionary, while in a transient case, the time-step, the duration of the injection, the mass flow rate and other properties defined are taken into account to calculate the number of parcels the injector is going to introduce in the system per time-step in order to fulfill the user's requirements.
In the particular case of a coupled simulation, the `sourceTerms` dictionary entry allows the user to specify what kind of scheme to use, which can be set to `explicit` or `semiImplicit`, as well as the relaxation factors, which have to be preceded by the name of the field they are going to be applied on (default value is 1). All this data introduced by the user will be processed by `cloudSolution.C`, located inside the following directory:

```
$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/clouds/Templates/KinematicCloud
/cloudSolution
```

It is also inside this file where it can be found what the switch `active` does. In fact, if it is set to true, `cloudSolution` will call the function `read()` in order to set up how is the solution going to be obtained according to the rest of parameters specified in the dictionary.
The `cellValueSourceCorrection` switch, when set to on, activates the correction of the momentum transferred from the lagrangian phase to the carrier phase in case the simulation is coupled. This is done by manipulating the updated momentum for the lagrangian phase and dividing its cell value by the mass of the cell. The function is implemented in the following file:

```
$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/parcels/Templates
/KinematicParcel/KinematicParcel.C
```

The `interpolationSchemes` entry is used for the definition of how the fields relative to the lagrangian phase have to be interpolated. The last entry inside `solution` states which schemes going to be used when integrating the lagrangian fields. Options for the schemes are `Euler` or `analytical`.
In what concerns the `constantProperties` dictionary entry, at least `rho0`, `youngsModulus` and `poissonsRatio` have to be specified.
Other parameters such as `parcelTypeId`, `rhoMin` and `minParticleMass` will be set to their default values in case they are not found (1e-15 is the default value for both density and mass). While the meaning of the last two parameters is fairly clear, the first one might not be so obvious. The `parcelTypeId` is just a form of identification of the particles belonging to this particular cloud. This might be useful if two or more different clouds with different properties are being post-processed. In this case, setting different `parcelTypeId` numbers, the different clouds will be flagged with different numbers (default is 1).

2.3.3 Submodels

The `submodels` directory contains the different templated models that can be added to the lagrangian particle cloud classes. The structure of the `submodels` directory in what concerns the `KinematicCloud` class is as follows:

```
submodels/
---- CloudFunctionObjects
    ---- CloudFunctionObject
    ---- CloudFunctionObjectList
    ---- FacePostProcessing
    ---- ParticleCollector
    ---- ParticleErosion
    ---- ParticleTracks
    ---- ParticleTrap
    ---- PatchPostProcessing
    ---- VoidFraction
---- ForceTypes
    ---- ParticleForceList
---- Kinematic
    ---- CollisionModel
    ---- DispersionModel
    ---- InjectionModel
    ---- ParticleForces
    ---- PatchInteractionModel
    ---- SurfaceFilmModel
---- SubModelBase.C
---- SubModelBase.H
```

This structure is what should be seen if one wants to recompile the intermediate library only taking the `kinematicCloud` class into account. This is useful in case this class is the only one being used since the compilation will be much faster.

Injection Model

The different injection models available are the following ones:

- `cellZoneInjection`
- `coneInjection`
- `coneNozzleInjection`
- `fieldActivatedInjection`
- `inflationInjection`
- `injectionModel`
- `kinematicLookupTableInjection`
- `manualInjection`
- `noInjection`
- `patchInjection`
- `patchFlowRateInjection`

A quick description of the injection along with the entries needed in the dictionary can be found at each `.H` file inside the correspondent injection model folder, which can be reached by typing the following line in the terminal

```
cd $WM_PROJECT_USER_DIR/src/lagrangian/intermediate/submodels/Kinematic/InjectionModel
```

Particle Forces

OpenFOAM allows the user to choose the forces to be included in the model. The forces inside the `ParticleForces` directory are:

- `Drag` Two possible drag models are available: `SphereDrag` and `NonSphereDrag`. The first one is a drag model assuming spherical particles while the latter is used for non-spherical particles and it is based on a coefficient obtained by dividing the area of a sphere with the same volume by the particle area. The Drag force for both cases is basically calculated as in equation 1.6.
- `Lift` Both Saffman-Mei for spherical particles and Tomiyama for deformable bubbles lift models are implemented. The lift coefficient is calculated with the chosen model and then the force is calculated inside `LiftForce.C`, after which it is stored in `forceSuSp` This class is a helper container for both explicit and implicit terms where `Su` is the explicit contribution (directly calculated as a force), and `Sp` is the implicit contribution (calculated as $\frac{force}{velocity}$) so that the total contribution is calculated as $F = S_p(u - u_p) + S_u$.
- `Gravity` The Gravity force is calculated with equation 1.7 and in the standard `KinematicParcel` class, a file named `g` with both units and value of the gravitational acceleration is needed inside the constant directory. The constructor of the cloud will ask for this file, which will be used for the calculation of the gravity force.
- `Paramagnetic` This model calculates the particle paramagnetic (magnetic field) force. Both the field and the magnetic susceptibility of the material are needed for the calculations.
- `PressureGradient` Function that calculates the pressure grandient force on the particles. An additional interpolation scheme has to be included for the `DUcDt` field, used for the calculations.
- `VirtualMass` Calculates the virtual mass force in coupled simulations in conjunction with the pressure gradient force. A dictionary with the virtual mass coefficient (`Cvm`) must be specified, along with the interpolation scheme for `DUcDt`. A typical value for the virtual mass coefficient is 0.5.
- `NonInertialFrame` Used to calculate the non inertial frame of reference force. The solver will look up inside the `kinematicCloudProperties` dictionary for the linear acceleration, angular velocity and angiular acceleration. If the user does not specify a name, default values will be chosen. If no value for accelerations and velocity is specified, zero values will be used as default, so the reference frame will remain static.
- `SRF` Allows calculation of the SRF reference frame force. The class contains a pointer to the SRF model being used so that the values needed will be extracted from the SRF properties

Distribution Models

In what concerns the injection of the lagrangian parcels, OpenFOAM contains a library of runtime-selectable distribution models.A distribution model is a function that, for a particular property defines quantitatively how the values of that property are distributed among the particles in the entire population. The current distribution models include:

- `exponential`
- `fixedValue`
- `general`
- `multiNormal`
- `normal`
- `RosinRammler`

- `uniform`

The equation used by each one of these distributioin models is specified inside the `.H` file inside OpenFOAM's `distributionModels` directory, which can be reached by running in the terminal window:

```
cd $FOAM_SRC/lagrangian/distributionModels
```

As an example, the Normal and the Rosin-Rammler distristributions are plotted below:

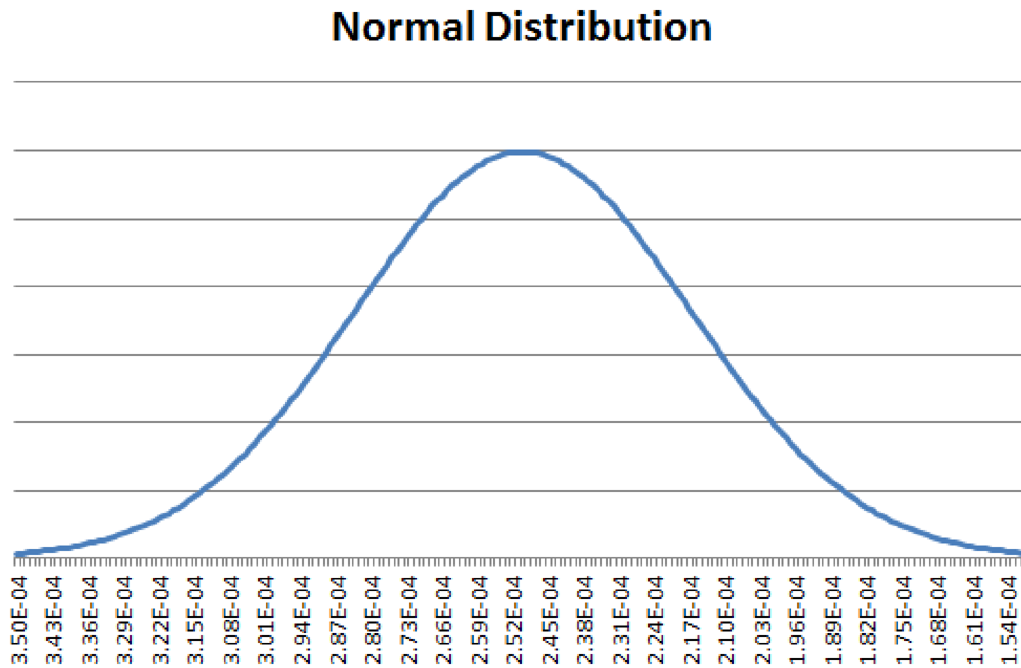

Figure 2.1: Normal Distribution for Particle diameters between $150\mu m$ and $350\mu m$

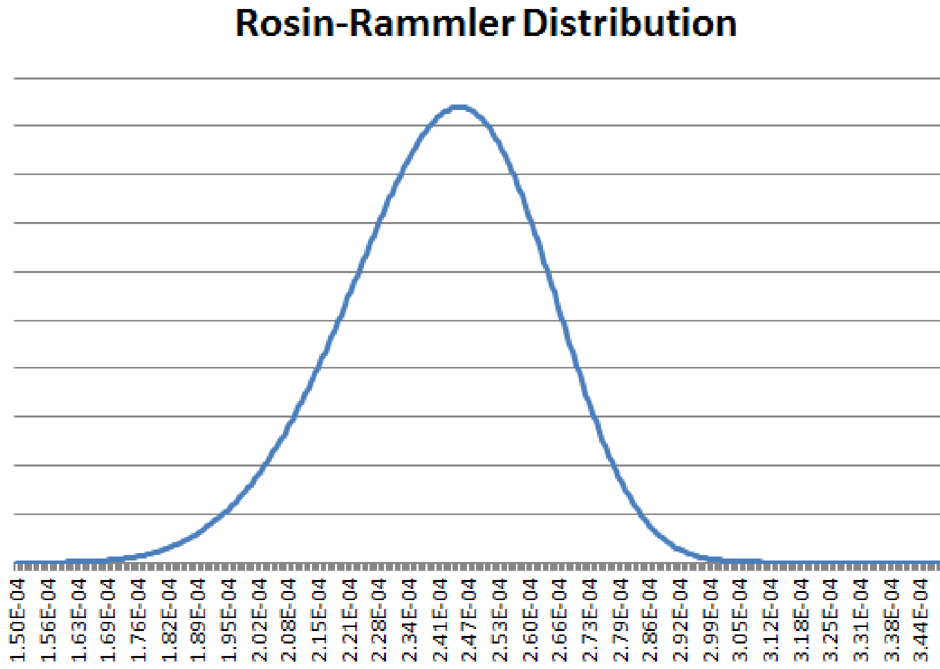

Figure 2.2: Rosin-Rammlerl Distribution for Particle diameters between $150\mu m$ and $350\mu m$

CloudFunctionObjects

One of many useful classes OpenFOAM provides the users with is the `CloudFunctionObjects`. These are library functions that provide additional capabilities to the cloud-based solvers. The available ones in OpenFOAM 2.2.x are the following ones:

- `facePostProcesing` It records particle face quantities on user-specified face zone. It supports accumulated mass and average mass flux calculations.
- `particleCollector` Function that collects the parcel-mass and mass flow rate over a set of polygons, defined as a list of points.
- `particleTracks` It records all particle variables one each call to postFace.
- `particleTrap` Traps the particles within a given phase fraction for multiphase cases.
- `patchPostProcessing` Standard post-processing. It outputs the desired information at the user-specified patches.
- `voidFraction` Creates the particle void fraction on the carrier phase.
- `particleErosion` This function creates the particle erosion field on the user-specified patches. It outputs a `volScalarField` which, at each face, it will be the sum of the volume eroded by all the particle hits.

A set up example of the `particleErosion` `CloudFunctionObject` can be found in the `kinematicCloudProperties` dictionary in Appendix 1. For the rest of the functions, the set up is basically the same. Only some of the variables have to be changed because each function requires different input information. However, the information needed can be found inside the code, by entering the respective `.H` file inside each `CloudFunctionObject` folder. This folders can be reached by typing in the terminal:

```
cd $WM_PROJECT_USER_DIR/src/
```

And then,

```
cd lagrangian/intermediate/submodels/CloudFunctionObjects
```

In case the intermediate folder has been recompiled in the user directory.

2.4 Erosion modelling

Impingement information, such as impact speed and impact angle, is gathered as particles hit the wall of the geometry.

2.5 Implementation of Erosion Modelling in OpenFOAM

Taking a look inside the `ParticleErosion.C` file, the constructor is implemented and it requires, in this case, the names of the patches where it is going to be applied and the plastic flow stress of the material being eroded. Both ratios, depth of contact to length of cut and normal and tangential forces, are also read but in this case, if they are not found, the default ones are used (2 is the default vaue for both of them).

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

template<class CloudType>
void Foam::ParticleErosion<CloudType>::preEvolve()
{
```

```
    if (QPtr_.valid())
    {
        QPtr_->internalField() = 0.0;
    }
    else
    {
        const fvMesh& mesh = this->owner().mesh();

        QPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + "Q",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimVolume, 0.0)
            )
        );
    }
}
```

The `preEvolve` member function, as seen above, initializes the field. It can be seen that the name the field will be given is going to be the name of the cloud that is being tracked with a Q at the end. In case the `kinematicCloud` is being used, the erosion field will have the name of `kinematicCloudQ`. The member function in charge of gathering all the necessary information, manipulate it and store it inside the erosion field is called `postPatch`. Here is where the user can set up his own erosion model (different from the one that is already implemented) just by taking the necessary particle variables and changing the function into the desired one.

```
template<class CloudType>
void Foam::ParticleErosion<CloudType>::postPatch
(
    const parcelType& p,
    const polyPatch& pp,
    const scalar trackFraction,
    const tetIndices& tetIs,
    bool&
)
{
    const label patchI = pp.index();

    const label localPatchI = applyToPatch(patchI);

    if (localPatchI != -1)
    {
        vector nw;
        vector Up;
```

```
        // patch-normal direction
        this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // particle velocity reletive to patch
        const vector& U = p.U() - Up;

        // quick reject if particle travelling away from the patch
        if ((nw & U) < 0)
        {
            return;
        }

        //Calculate magnitude of the particle velocity at impingement
        const scalar magU = mag(U);
        //Udir is the velocity unitary vector, i.e, the direction of the particle at impingement.
        const vector Udir = U/magU;

        // determine impact angle, alpha
        const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

        const scalar coeff = p.nParticle()*p.mass()*sqr(magU)/(p_*psi_*K_);

        const label patchFaceI = pp.whichFace(p.face());
        scalar& Q = QPtr_->boundaryField()[patchI][patchFaceI];
        if (tan(alpha) < K_/6.0)
        {
            Q += coeff*(sin(2.0*alpha) - 6.0/K_*sqr(sin(alpha)));
        }
        else
        {
            Q += coeff*(K_*sqr(cos(alpha))/6.0);
        }
    }
}


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

If the user-specified patch is hit, the magnitude of the velocity of the impinging particle at that moment is calculated with the expression `mag(u)` and stored inside `magU`. In order to calculate the angle of impingement, the direction of the particle velocity is determined first ad stored in the `Udir` vector. The angle of impingement is the one between `Udir` and the patch normal direction, and it is stored inside `alpha`. the scalar `coeff` is the number of particles inside the parcel times the mass of those particles (i.e., the total mass) multiplied by the velocity and the constant coefficients: plastic flow stress and the two ratios. The field this `CloudFunctionObject` writes to the case directory is going to be zero everywhere but in the specified patches, where is is going to print a `nonuniform List<scalar>`, which will be the erosion rate at each face of the boundary patches specified. The equation used for the calculation of the erosion field is exactly equations 1.10 and 1.11.

2.6 Templating in OpenFOAM

Due to the fact that templates are very common within the `KinematicParcel` class, an introduction to templating could be of some use in order to be able to understand and customize Lagrangian simulations in OpenFOAM.
C++ requires the declaration of variables, functions, and most other kinds of entities using specific types. However, a lot of code looks the same for different types. Templates are functions or classes that are written for one or more types not yet specified. When using a template, the types are passed as arguments, explicitly or implicitly.

2.6.1 Function Templates [3] [6] [8]

A function template is used to represent a family of functions. The difference with ordinary functions is that, in the template, some of the parameters are left undetermined, i.e., parametrized. For example, in case one wants to create a function that returns the greater one of two objects:

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
return a<b?b:a;
}
```

The keyword `typename` introduces a type parameter. This is the most common one but it is not the only possible kind of parameter. The template parameters have to be announced with a syntax like:

```
template<parameters separated by commas>
```

In the example, the type parameter is `T`. This type parameter can have any identifier. It represents an arbitrary type that is going to be specified when the function is called. The condition to use a particular type is that it supports the operation that is being done in the function. For instance, in the example, the type specified has to support the operator `<`. The word `typename` may also be substituted by the word `class`. This keyword was actually the only way to introduce type parameters until the use of the keyword `typename` was enabled and it still remains valid. Normally, templates aren't compiled into single entities that can handle any type. Instead, different entities are generated from the template for every type for which the template is used. The process of replacing the template parameters by concrete types is called instantiation. In the example above the template can be instantiated:

```
inline int const& max (int const& a, int const& b)
{
// if a < b then use b else use a
return a<b?b:a;
}
```

for integers,

```
const double& max (double const&, double const&);
```

for double,

```
const std::string& max (std::string const&, std::string const&);
```

for strings and so on.
OpenFOAM contains a very large amount of templates. As an example, the following functions are used inside the `KinematicParcel` class to interpolate the density and the velocity. However,they have different types, being the density a scalar and the velocity a vector. That is why the generic type of `Foam::interpolation` is redefined for each one of the functions: scalar for the density and vector for the velocity.

```
template<class ParcelType>
template<class CloudType>
inline const Foam::interpolation<Foam::scalar>&
Foam::KinematicParcel<ParcelType>::TrackingData<CloudType>::rhoInterp() const
{
    return rhoInterp_();
}

template<class ParcelType>
template<class CloudType>
inline const Foam::interpolation<Foam::vector>&
Foam::KinematicParcel<ParcelType>::TrackingData<CloudType>::UInterp() const
{
    return UInterp_();
}
```

An attempt to instantiate a template for a type that doesn't support all the operations used within it will result in a compile-time error. Thus, templates are compiled twice: without instantiation, the template code is checked for correct syntax and when instantiated, it is checked that the calls are valid. When function templates are called for some arguments, the template parameters are determined by the arguments passed. If integers are passed as arguments, the compiler must conclude that the parameters are integers. In templating, there are two different kinds of parameters:

- Template parameters: These are declared in angle brackets (`template<typename T>`) before the function template name.
- Call parameters: Declared in parentheses (`max (T const& a, T const& b)`)after the function template name.

Also, like ordinary functions, function templates can be overloaded. There can be different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call.

2.6.2 Class Templates [3] [6] [8]

Similar to functions, classes can also be parameterized with one or more types. Container classes, which are used to manage elements of a certain type, are a typical example of this feature. The declaration procedure is very similar to function templates. Just before the declaration, a statement declares an identifier as a type parameter. Inside the class, the type parameter can be used like any other type in oder to declare members and member functions. Because templates are compiled when required, the implementation of a template class function must be in the same file as its declaration. The class template declaration starts with the same syntax as the function templates:

```
template<class T>
class Item
```

The keyword class is used twice. The first one defines the template type specification, and the second one is the C++ class declaration. While in funciton templates, it is the compiler the one that deduces the template type arguments, in class templates the user must explicitly pass the template type (in angle brackets `<>`).
An additional feature of templates is specialization. This feature enables the user to define a different implementation for a template when a specific type is passed as template parameter. An example of specialization could be as follows:

```
template <> class myclass <char> { ... };
```

This allows the definition of a specific implementation when the argument is of type char. The fact that the angle brackets are empty (`<>`) allows the identification of this structure as a template specification.

2.7 Coupling of the kinematicCloud class and an incompressible solver

2.7.1 Uncoupled Lagrangian Particle Tracking

First of all, it is best if the necessary libraries to couple the incompressible solver are recompiled inside the `$WM_PROJECT_USER_DIR` directory. An `src/lagrangian` directory should have been created inside the user directory by typing:

```
cd $WM_PROJECT_USER_DIR
mkdir -p src/lagrangian/
```

As explained at the beginning of this chapter. The -p argument is used in this case to create the nested directories. Once this has been done, the next step is to copy the necessary files for compilation.

```
cp -r $FOAM_SRC/lagrangian/intermediate $WM_PROJECT_USER_DIR/src/lagrangian
```

Now all the files inside the intermediate library have been copied to the user directory and the next step is to recompile them. Here, there are two possibilities: Recompile them without changing anything inside the `\Make` directory or change a few lines there and then recompile the library inside the user's `src/` directory. The basic difference is the compilation time. If one is going to be constantly working with this library (or any other library) and assuming that, in the process of adding new lines to the source code and changing the functions, the library is going to be recompiled, it is more efficient to just recompile the part of the code being used, since this will save a lot of time in compilation time. In this case, as the libraries that are going to be used are the ones that have the word `kinematic` on them, the rest of the parcel types can be deleted from the `Make/files` directory, leaving only the following:

```
PARCELS=parcels
BASEPARCELS=$(PARCELS)/baseClasses
DERIVEDPARCELS=$(PARCELS)/derived

CLOUDS=clouds
BASECLOUDS=$(CLOUDS)/baseClasses
DERIVEDCLOUDS=$(CLOUDS)/derived


/* Cloud base classes */
$(BASECLOUDS)/kinematicCloud/kinematicCloud.C

/* kinematic parcel sub-models */
KINEMATICPARCEL=$(DERIVEDPARCELS)/basicKinematicParcel
$(KINEMATICPARCEL)/defineBasicKinematicParcel.C
$(KINEMATICPARCEL)/makeBasicKinematicParcelSubmodels.C

KINEMATICCOLLIDINGPARCEL=$(DERIVEDPARCELS)/basicKinematicCollidingParcel
$(KINEMATICCOLLIDINGPARCEL)/defineBasicKinematicCollidingParcel.C
$(KINEMATICCOLLIDINGPARCEL)/makeBasicKinematicCollidingParcelSubmodels.C

submodels/Kinematic/PatchInteractionModel/LocalInteraction/patchInteractionData.C
submodels/Kinematic/PatchInteractionModel/LocalInteraction/patchInteractionDataList.C

KINEMATICINJECTION=submodels/Kinematic/InjectionModel
$(KINEMATICINJECTION)/KinematicLookupTableInjection/kinematicParcelInjectionData.C
```

```
$(KINEMATICINJECTION)/KinematicLookupTableInjection/kinematicParcelInjectionDataIO.C
$(KINEMATICINJECTION)/KinematicLookupTableInjection/kinematicParcelInjectionDataIOList.C

/* integration schemes */
IntegrationScheme/makeIntegrationSchemes.C

/* phase properties */
phaseProperties/phaseProperties/phaseProperties.C
phaseProperties/phaseProperties/phasePropertiesIO.C
phaseProperties/phasePropertiesList/phasePropertiesList.C

/* Additional helper classes */
clouds/Templates/KinematicCloud/cloudSolution/cloudSolution.C

LIB = $(FOAM_USER_LIBBIN)/libkinematiclagrangianIntermediate
```

Inside the `\Make` directory, the `files` file should be similar to the one above. It is important to remember changing the location directory into the user's one. In this case the library is written inside `username-2.2.x/plattforms/linux64GccDPOpt/lib` (the folder containing the lib and bin directories may a slightly different name).

After this, the library can be recompiled by typing the following two commands in the terminal window:

```
wclean lib
wmake libso
```

Once the library is compiled, the next step is to link it to the incompressible solver that is going to be created. In this first case, an incompressible transient uncoupled solver is going to be created. For low particle concentrations, this is a sufficiently accurate assumption. The steps to create the solver are the following ones:

- Create the solver directory inside the user's `applications/solvers` directory and copy the original pimpleFoam solver into that directory by doing:

  ```
  cd $WM_PROJECT_USER_DIR/ applications/solvers/
  mkdir pimpleKinematicFoam
  ```

- Copy the pimpleFoam solver into the created directory

  ```
  cd pimpleKinematicFoam
  cp -r $WM_PROJECT_DIR/ applications/solvers/incompressible/pimpleFoam/* .
  ```

  This will copy all the files inside the pimpleFoam directory into the user directory (also pimpleDyMFoam and SRFPimpleFoam solvers will be copied, but one can get rid of them easily, if they are not going to be used by doing `rm -r pimpleDyMFoam SRFPimpleFoam` The next step is to change the name of the solver to the desired one by doing:

  ```
  mv pimpleFoam.C pimpleKinematicFoam.C
  ```

- Modify the `Make/files` and `Make/options`

  Inside files, the user should have:

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

Which tells the compiler which files to compile and to store the app created inside the user directory.

The options file should contain the following lines for the compiler to know where to look for the files:

```
LIB_USER_SRC = $(WM_PROJECT_USER_DIR)/src

EXE_INC = \
    -I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/finiteVolume/lnInclude\
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude \
    -I$(LIB_SRC)/regionModels/regionModel/lnInclude \
    -I$(LIB_SRC)/lagrangian/intermediate/lnInclude \
    -I$(LIB_USER_SRC)/lagrangian/intermediate/lnInclude \
#swap LIB_USER_SRC for LIB_SRC in case the $FOAM_SRC library is used
    -I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/liquidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/properties/solidMixtureProperties/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/SLGThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/radiationModels/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude



EXE_LIBS = \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -lmeshTools\
    -lfvOptions \
    -llagrangian\
    -llagrangianIntermediate \
    -lkinematiclagrangianintermediate \
    -lthermophysicalFunctions \
    -lsurfaceFilmModels \
    -ldistributionModels \
    -lregionModels \
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
```

```
-lliquidMixtureProperties \
-lsolidProperties \
-lsolidMixtureProperties \
-lreactionThermophysicalModels \
-lSLGThermo \
-lradiationModels \
-lLESdeltas \
-lcompressibleTurbulenceModel \
-lcompressibleRASModels \
-lcompressibleLESModels \
-lregionModels \
-lsurfaceFilmModels \
-ldynamicFvMesh \
-lsampling
```

It is essential to tell the compiler where to look for the user's library. The purpose of the first line is to let the compiler know that when it reads, `LIB_USER_SRC`, it should go to the `WM_PROJECT_USER_DIR/src` directory, and there, look for the intermediate library.

- Once this is ready, it is time to modify the `createFields.H` file:

  At the beginning of the file the following lines are added in order to be able to look for some properties defined in the `transportProperties` dictionary for further manipulation:

```
Info<< "\nReading transportProperties\n" << endl;

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


  dimensionedScalar rhoInfValue
  (
      transportProperties.lookup("rhoInf")
  );
```

  Where `rhoInf` is the density of the carrier phase defined in the mentioned dictionary.

- A couple of fields for the carrier phase density and viscosity have to be created too. It is an incompressible solver and they are not going to vary, but the `KinematicCloud` class needs those for the constructor. The first of those fields is the density:

```
volScalarField rhoInf
(
    IOobject
    (
```

```
            "rho",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
        rhoInfValue
    );
```

And the second one is the dynamic viscosity:

```
     volScalarField mu
    (
        IOobject
        (
            "mu",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        laminarTransport.nu()*rhoInfValue
    );
```

The dynamic viscosity is to be added just after the following lines:

```
autoPtr<incompressible::turbulenceModel> turbulence
(
    incompressible::turbulenceModel::New(U, phi, laminarTransport)
);
```

- The last modification inside the `createFields.H` file is to include the constructor for the `KinematicCloud` class:

```
    word kinematicCloudName("kinematicCloud");
    args.optionReadIfPresent("cloudName", kinematicCloudName);

    Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
    basicKinematicCloud kinematicCloud
    (
        kinematicCloudName,
        rhoInf,
        U,
        mu,
        g
    );
```

As seen in this last piece of code, the constructor needs the density of the carrier phase and the dynamic viscosity, which were created in the previous step. This constructor will also ask for the gravity, and that means, that the user must have a `g` file inside the `constant` directory with the value and units of the gravity force so that the `readGravitationalAcceleration.H` header, which is going to be included later, is able to read it from that location.

- The modifications to the `pimpleKinematicFoam.C` file are the following ones:

  Copy the following line after `#include "IOMRFZoneList.H"`:

```
#include "basicKinematicCloud.H"
```

  This tells the compiler which cloud is going to be used. If four way coupling was necessary, the file to paste inside the `#include` statement would be `basicCollidingCloud.H`.

  Include after `int main(int argc, char *argv[])` the following lines:

```
argList::addOption
    (
        "cloudName",
        "name",
        "specify alternative cloud name. default is 'kinematicCloud'"
    );
```

  Which adds the option for the user to specify an alternative cloud name.

  Then, after `#include "createMesh.H"`:

```
#include "readGravitationalAcceleration.H"
```

  So the solver knows how and where from to read the gravitational acceleration to use it for both calculations and the construction of the cloud.

- Finally, just before `runTime.write();` the following lines are added too:

```
Info<< "Evolving " << kinematicCloud.name() << endl;
kinematicCloud.evolve();
```

- Once all this is completed, the solver can be compiled and, hopefully, the process will not output any errors. In order to compile, inside the solver directory run, as always:

```
wclean
wmake
```

This is the procedure to create the solver. In the next chapter, the necessary files for running a case will be specified.

2.7.2 Coupled Lagrangian Particle Tracking

As it can be seen in the `calc` function displayed below, the term responsible for the coupling is `Su()`.

```
template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::calc
(
    TrackData& td,
    const scalar dt,
    const label cellI
)
{
    // Define local properties at beginning of time step
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    const scalar np0 = nParticle_;
    const scalar mass0 = mass();

    // Reynolds number
    const scalar Re = this->Re(U_, d_, rhoc_, muc_);


    // Sources
    //~~~~~~~~

    // Explicit momentum source for particle
    vector Su = vector::zero;

    // Linearised momentum source coefficient
    scalar Spu = 0.0;

    // Momentum transfer from the particle to the carrier phase
    vector dUTrans = vector::zero;


    // Motion
    // ~~~~~~

    // Calculate new particle velocity
    this->U_ = calcVelocity(td, dt, cellI, Re, muc_, mass0, Su, dUTrans, Spu);


    // Accumulate carrier phase source terms
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if (td.cloud().solution().coupled())
    {
        // Update momentum transfer
        td.cloud().UTrans()[cellI] += np0*dUTrans;

        // Update momentum transfer coefficient
        td.cloud().UCoeff()[cellI] += np0*Spu;
    }
}
```

And the function to calculate the new particle velocity (calcVelocity):

```
        //- Calculate new particle velocity
        template<class TrackData>
        const vector calcVelocity
        (
            TrackData& td,
            const scalar dt,            // timestep
            const label cellI,          // owner cell
            const scalar Re,            // Reynolds number
            const scalar mu,            // local carrier viscosity
            const scalar mass,          // mass
            const vector& Su,           // explicit particle momentum source
            vector& dUTrans,            // momentum transfer to carrier
            scalar& Spu                 // linearised drag coefficient
        ) const;
```

If the solution is properly coupled, the solver will write out two fields:

- nameoftheCloud:Ucoeff It is of type volscalarField::DimensionedInternalField It is a `nonuniform List<vector>`
- nameoftheCloud:Utrans It is of type volVectorField::DimensionedInternalField It is a `nonuniform List<scalar>`

However, if the solution is not coupled, these fields will be written to the `runTime` but instead of being non-uniform lists, they will be `uniform 0;` and `uniform (0 0 0);` respectively. First, and provided that the `KinematicClass` is being coupled to an incompressible solver, the momentum will have to be divided by the density of the carrier phase. In pimpleFoam and the rest of the incompressible solver in OpenFOAM, the equations are divided by the carrier phase density. An easy way of doing this is creating the inverse of the density and then, in the file`UEqn.H`, multiply the momentum by that inverse, so there will be no mismatched units. Inside the `createFields.H` the following line is added after the density is read from the `particleProperties` dictionary:

```
dimensionedScalar invrhoInf("invrhoInf",(1.0/rhoInfValue));
```

Then, in order to create the coupled version of the solver it is necessary to include this momentum transfer into the `UEqn.H`, so the file will look now like:

```
// Solve the Momentum equation

tmp<fvVectorMatrix> UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
 ==
     fvOptions(U)
    +invrhoInf*kinematicCloud.SU(U)
);

UEqn().relax();

fvOptions.constrain(UEqn());

volScalarField rAU(1.0/UEqn().A());

if (pimple.momentumPredictor())
{
    solve
    (
        UEqn()
         ==
        -fvc::grad(p)
    );

    fvOptions.correct(U);
}
```

Finally, once all the changes are done and the files are saved the final step is to run in the terminal (inside the solver directory):

```
wclean
wmake
```

Given that the two-way coupling is going to be switched on and off inside the `kinematicCloudProperties` dictionary, it is probably a good option to create a unique solver and decide whether the simulation is going to be coupled or uncoupled switching from on to off in the mentioned dictionary.

Chapter 3

Preprocessing

3.1 Geometry definition

The geometry chosen for this tutorial is a very simple one, consisting of a 100 mm length quadrangular pipe with a 90 degrees bend (figure 3.1).

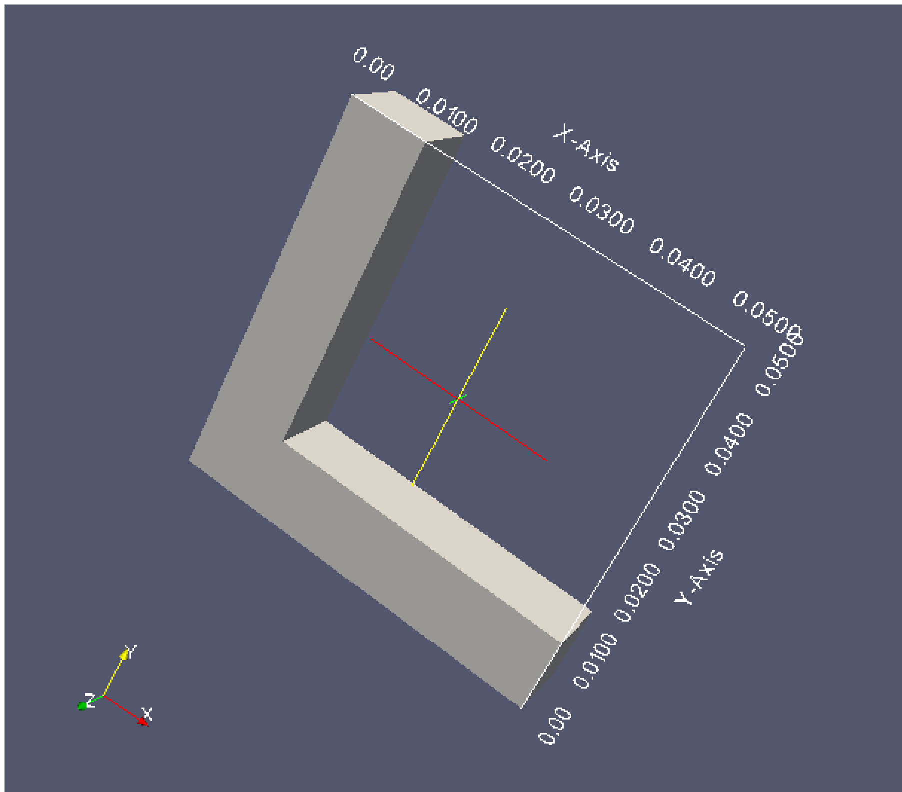

Figure 3.1: Geometry of the pipe used for the tutorial case.

The geometry is defined in the `blockMeshDict` dictionary and in order to generate the rest of the mesh files needed by OpenFOAM, the `blockMesh` command has to be run in the case directory.

3.2 The 0/ directory

In order to have a more efficient case solution it is recommended to run the case without the particles until it reaches the steady state (`simpleFoam` is the most suitable one for this simulation) and then use the command,

```
mapFields /Path-To-Steady-State-Case-Directory -consistent
```

to map the fields obtained for the fluid flow into the transient case with particles and use them as initial conditions in our 0 directory. Regarding the application settings, before running `mapFields`, the starting time in the transient case has to be the same one as the time step being mapped from the steady state solution, and the directory created by the application and containing the non-uniform scalar and vector fields has to be renamed as `0`, once `mapFields` has finished the transfer.
For the steady-state case, three directories are necessary: `0`, `constant` and `system`. Once the steady-state case has been defined, the command to run it is as follows:

```
simpleFoam >&log&
```

The calculation process can also be viewed (and stopped with ctrl + C) in the terminal typing:

```
tail -f log
```

The case can also be killed by typing,

```
pidof simpleFoam
```

Which outputs the PID of the process and then, being PID the number obtained in the terminal window, run,

```
kill PID
```

Once the steady state case fields are correctly mapped into the transient case directory, it is time to set up the necessary files for the lagrangian simulation.

3.3 The constant/ directory

Inside the `constant` directory the user must specify the properties of the lagrangian simulation in a dictionary called `kinematicCloudProperties`. The example used for this tutorial can also be found in Appendix 1. A file named g is necessary for the construction of the cloud, as explained before. This file must, at least, contain:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

Where the gravity vector is specified, depending on which reference is being used. For this case, the gravity is in the -$\hat{\jmath}$ direction.
In this particular case, a $k-\epsilon$ model has been chosen. For this, a `RASProperties` file is also required containing:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        kEpsilon;

turbulence      on;

printCoeffs     on;
```

And as explained before, the `transportProperties` file, which should look similar to:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel  Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 1e-06;
rhoInf        rhoInf  [ 1 -3 0 0 0 0 0] 1000;
```

3.4 The system/ directory

The files inside the system directory will be the usual ones, i.e., `fvSchemes`, `fvSolution` and `controlDict`. However, it is good to pay some attention in this kind of simulations to the Courant number. The Courant number is defined as:

$$C = \frac{u\Delta t}{\Delta x} \tag{3.1}$$

Being u the velocity, Δx the space interval and Δt the time interval. Keeping the courant number under the value of 1 will help the solution converge, specially when dealing with coupled simulations.

3.5 Running the case

Once the case is properly set up, it can be run by typing:

```
pimpleKinematicFoam >&log&
```

which will allow later the use of `foamLog` to extract the residuals relative to each of the variables for plotting.

Chapter 4

Postprocessing

4.1 Lagrangian Particles in Paraview

There are actually two ways for visualization of the lagrangian cloud in paraview. The first one is to transform the case data into VTK format by doing:

```
foamToVTK
```

The second one is to run `paraFoam` in the terminal window and then click on "Skip Zero Time" (no parcels have been released at zero time. That is why no lagrangian fields or cloud are available for display). Once this is done, any of the lagrangian fields as well as the kinematic cloud can be displayed with paraview just by checking the box relative to each one of them.

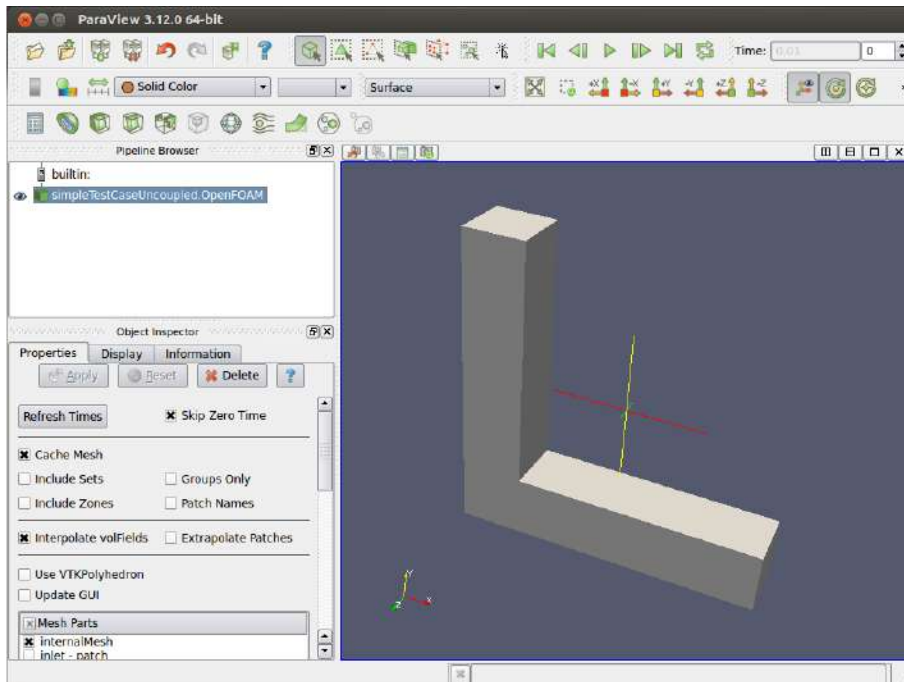

Figure 4.1: Check "Skip Zero Time" box

Figure 4.2: Check "kinematicCloud-lagrangian" and any of the available lagrangian fields

4.2 Results of Coupled and Uncoupled Simulations

As expected, in this particular case, the case with two-way coupling is practically identical to the one-way coupled. This is due to the fact that the momentum transferred between phases is actually negligible, being the values very close to zero. In case of the uncoupled simulation, the solver allows the user to set up a higher courant number (`maxCo` in `controlDict`), which, consequently will enlarge the time step, and will be reflected in a much faster simulation. However, in the coupled case, instabilities might appear when trying to set a high courant number, causing the solver to diverge. Thus, the courant number to be set up in the `controlDict` dictionary must be carefully chosen, taking into account the size of our mesh the velocity and the time-step.

4.3 Post-processing erosion in Paraview 3.12.0

In order to postprocess erosion in Paraview, the "kinematicCloudQ" field box has to be checked ito display the erosion field as shown in picture 4.4.

What paraview is representing in the erosion contours is the volume of material eroded, as the sum of the material eroded by each of the individual impacts of the particles at each face.

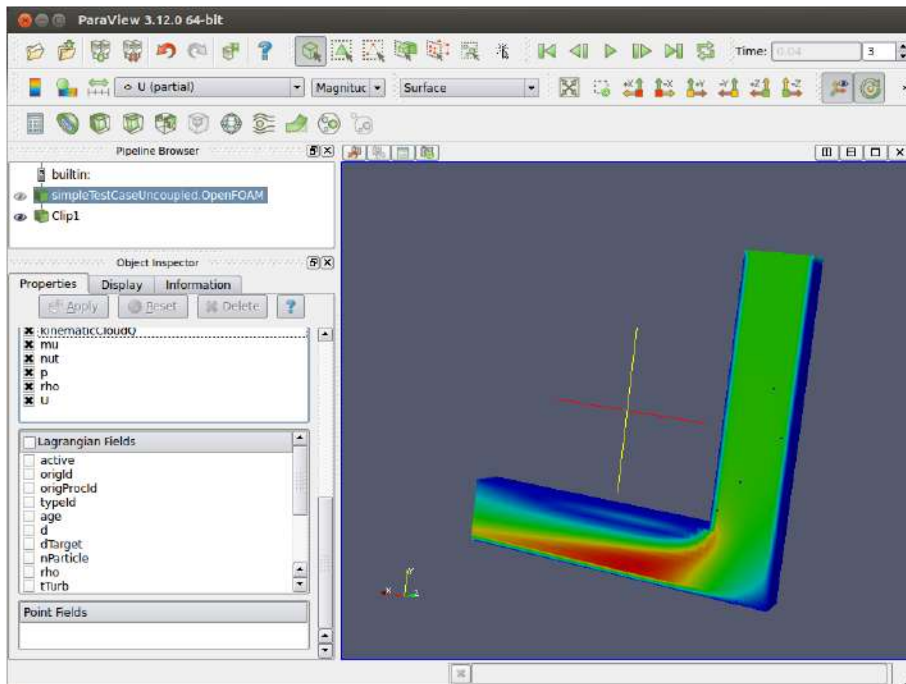

Figure 4.3: Visualization of the particles in paraview

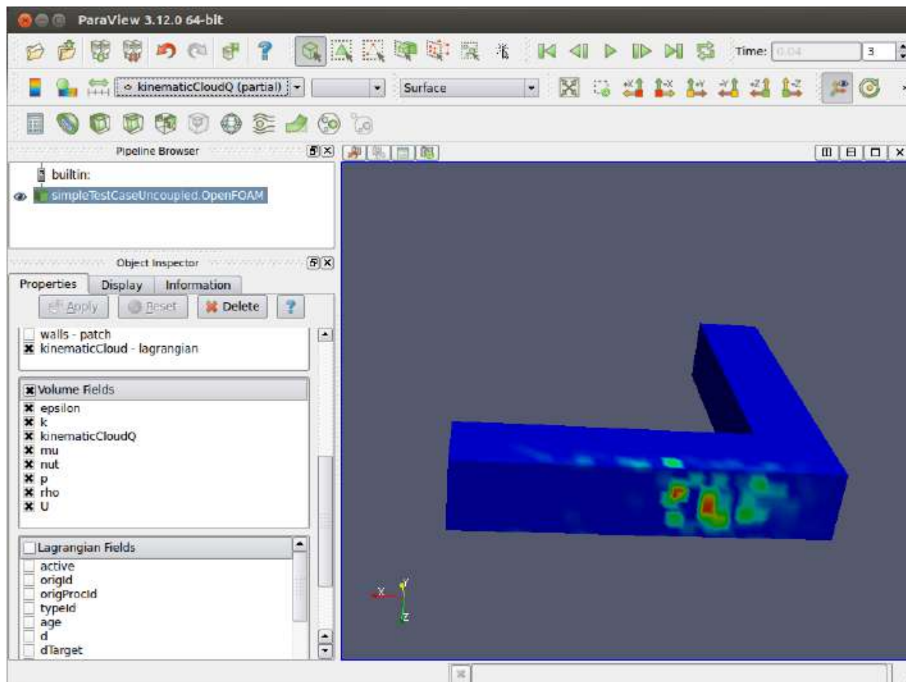

Figure 4.4: Erosion contours in paraview

Chapter 5

Appendix 1

5.1 kinematicCloudProperties Dictionary

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

solution
{
    active          true; //can be set to true or false
    coupled         false; //true or false for coupled or uncoupled simulations
    transient       yes; //yes or no, no for steady-state calculations
    cellValueSourceCorrection off; //when set to on it activated the
//correction of the momentum tranferred to the eulerian phase

    sourceTerms
    {
        schemes
        {
U          semiImplicit    1;//explicit or semiImplicit
     //ALSO  specify relaxCoeff for each of the fields
        }
    }

    interpolationSchemes
    {
        rho             cell;
```

```
        U               cellPoint;
        mu              cell;
      //curlUcDt cell; //field used for Lift force calculations
      //DucDt cell;//filed used for pressureGradient calculations
/*Available schemes are:
cell
cellPatchConstrained
cellPoint
cellPointFace
cellPointWallModified
pointMVC
*/
    }

    integrationSchemes
    {
        U               Euler;
/*Available schemes are:A dictionary with the value of
Euler
analytical
*/
    }
}


constantProperties
{
    //parcelTypeId 1:
    //rhoMin 1e-15;
    //minParticleMass 1e-15:
    rho0            3217;
    youngsModulus   700e9;
    poissonsRatio   0.187;
}


subModels
{
    particleForces
    {
        sphereDrag;
        gravity;
/*SaffmanMeiLiftForce        //TomiYamaLift may be chosen instead
{
U           U;
}
*/
      /*paramagnetic
{
magneticSusceptibility         -6.3e-9; //m^3/kg for graphite
HdotGradH           U;
*/ }

/*pressureGradient
```

```
{
U        U;
}
*/

/*virtualMass
{
Cvm        0.5;
}
*/

    /*nonInertialFrame
        {
             linearAccelerationName   linearAc;
                     linearAcceleration     10;
               angularVelocityName    angVelo;
                     angularVelocity         5;
                     angularAccelerationName  angAcc
                     angularAcceleration    5;
          }
      */

//SRF;

    }

    injectionModels
    {
 model1
     {
         type            patchInjection;
         patchName       inlet;
  SOI 0; //Start of injection
  massFlowRate 0.01;
  massTotal 0.2;
  parcelBasisType mass;
  flowRateProfile 0.01;
  sizeDistribution
{
type RosinRammler;
RosinRammlerDistribution
{
minValue 200e-6;
maxValue 300e-6;
d 250e-6;
n 3;
}

}
         duration            20;
         parcelsPerSecond   500000;
         U0              ( 0 -15 0 );
     }
    }
```

```
dispersionModel none;

patchInteractionModel standardWallInteraction;

heatTransferModel none;

surfaceFilmModel none;

collisionModel none;

radiation off;

pairCollisionCoeffs
{
    // Maximum possible particle diameter expected at any time
   /* maxInteractionDistance  0.006;

    writeReferredParticleCloud no;

    pairModel pairSpringSliderDashpot;

    pairSpringSliderDashpotCoeffs
    {
        useEquivalentSize   no;
        alpha               0.12;
        b                   1.5;
        mu                  0.52;
        cohesionEnergyDensity 0;
        collisionResolutionSteps 12;
    };

    wallModel    wallLocalSpringSliderDashpot;

    wallLocalSpringSliderDashpotCoeffs
    {
        useEquivalentSize no;
        collisionResolutionSteps 12;
        walls
        {
            youngsModulus   1e10;
            poissonsRatio   0.23;
            alpha           0.12;
            b               1.5;
            mu              0.43;
            cohesionEnergyDensity 0;
        }
        frontAndBack
        {
            youngsModulus   1e10;
            poissonsRatio   0.23;
            alpha           0.12;
            b               1.5;
            mu              0.1;
```

```
                cohesionEnergyDensity 0;
            }
        };*/
    }

    standardWallInteractionCoeffs
    {
        type            rebound;
    }
}


cloudFunctions
{
      particleErosion
     {

          functionObjectLibs ("libcloudFunctionObjects.so");
          enabled         true;
          outputControl   outputTime;
          log             true;
         valueOutput     true;
         p 11000000; //yield stress for aluminium = 11000000 Pa or 11 MPa
psi 2;//Ratio of the depth of contact to the depth of cut (default value = 2 )
K 2; //Ratio of vertical to horizontal force components (2 for angular abrassive grains)
         patches
         (
             moving-wall
         );
     }
}

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

Chapter 6

Appendix 2

6.1 blockMeshDict

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

convertToMeters 0.001;

vertices
(
    (0 0 0)//0
    (0 50 0)//1
    (0 50 -10)//2
    (0 0 -10)//3
    (10 10 0)//4
    (10 50 0)//5
    (10 50 -10)//6
    (10 10 -10)//7
    (50 0 0)//8
    (50 10 0)//9
    (50 10 -10)//10
    (50 0 -10)//11
    (10 0 -10)//12
    (10 0 0)//13
    (0 10 -10)//14
    (0 10 0)//15
);
```

```
blocks
(
    hex (13 4 9 8 12 7 10 11) (10 40 10) simpleGrading (1 1 1)
    hex (0 15 4 13 3 14 7 12) (10 10 10) simpleGrading (1 1 1)
    hex (15 1 5 4 14 2 6 7) (40 10 10) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (1 5 6 2)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (9 8 11 10)
        );
    }
    walls
    {
        type wall;
        faces
        (
            (4 9 10 7)
            (4 13 8 9)
            (12 11 8 13)
            (7 10 11 12)
            (3 12 13 0)
            (0 13 4 15)
            (0 15 14 3)
        (3 14 7 12)
            (15 4 5 1)
            (5 4 7 6)
            (6 7 14 2)
        (14 15 1 2)
        );
    }
);

mergePatchPairs
(
```

```
);

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

Bibliography

[1] G.J. Brown. Erosion prediciton in slurry pipeline tee-junctions. *Applied Mathematical Modelling*, (26):155–170, 2002.

[2] cplusplus.com. C++ tutorial, 2012. `http://www.cplusplus.com/doc/tutorial/`.

[3] Nicolai M. Josuttis David Vandevoorde. *C++ Templates*. Addison Wesley, 2002.

[4] Iain Finnie. Erosion of surfaces by solid particles. *Wear*, (3):87–103, 1960.

[5] Iain Finnie. Some observations on the erosion of ductile metals. *Wear*, (19):81–90, 1972.

[6] OpenFOAM Foundation. Openfoam c++ documentation, 2011. `http://www.openfoam.org/docs/cpp/`.

[7] OpenFOAM Foundation. Openfoam programmer's guide, 2011. `http://foam.sourceforge.net/docs/Guides-a4/ProgrammersGuide.pdf`.

[8] OpenFOAM Foundation. Openfoam user guide, 2011. `http://www.openfoam.org/docs/user/`.

[9] Chalmers University of Technology Hå kan Nilsson. Msc/phd course in cfd with opensource software, 2013 and previous years, 2013. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/`.

[10] A.W. Heemink J.W. Stijnen and H.X. Lin. An efficient 3d particle transport model for use in stratified flow. *International journal for numerical methods in fluids*, (51):331–350, 2006.

[11] Special Interest Group on Multiphase Flows (SIG Multiphase). Tutorials for particle based methods, 2011. `http://openfoamwiki.net/index.php/Tutorials_for_particle_based_methods`.

[12] Aurélia Vallier. Coupling of vof with lpt in openfoam, 2011. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2011/OF_kurs_LPT_120911.pdf`.

[13] Aurélia Vallier. Tutorial lagrangian particle tracking, 2011. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/AureliaVallier/oscfd09_present_aureliapdf.pdf`.