Stromninsteknik - Energivetenskaper

LUND TEKNISKA HOGSKOLA

CFD WITH OPENSOURCE SOFTWARE, ASSIGNMENT 3

Tutorial icoLagrangianFoam / solidParticle

Author:
Aurelia VALLIER

December 13, 2009

Chapter 1

Theory

In this chapter we present the equations solved when modeling particles and incompressible flow.

1.1 Equations in the Eulerian frame

The fluid phase is governed by the incompressible Navier Stokes equations

$$\nabla \cdot \mathbf{U} = 0$$

$$\rho \frac{\partial \mathbf{U}}{\partial t} + \rho(\mathbf{U} \cdot \nabla \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho g - \sum_N \mathbf{f_P} \tag{1.1}$$

The additional source term in the momentum equation (1.1) is due to particle-fluid interactions. The force $\mathbf{f_P}$ exerted by a particle on a unit volume of fluid ΔV is equal to the difference in particle momentum between the instant it enters (t_{in}) and leaves (t_{out}) the control volume .

$$\mathbf{f_P} = \frac{m_P}{\Delta V} \frac{(\mathbf{U_P})_{t_{out}} - (\mathbf{U_P})_{t_{in}}}{t_{out} - t_{in}} \tag{1.2}$$

The term $\mathbf{f_P}$ is calculated for each particle and each Lagrangian time step Δt.

After calculating the particle movement during an Eulerian time step (divided in several Lagrangian time step), the source term generated by the particles is evaluated with relation (1.2) and the fluid velocity is updated with equation (1.1).

1.2 Equations in the lagrangian frame

A particle P is defined by the position of its center $\mathbf{x_P}$, its diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P. The mass of the particle is $m_P = \frac{1}{6}\rho_P \pi D_P^3$. In a Lagrangian frame, each particle position vector $\mathbf{x_P}$ is calculated from the equation

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P} \tag{1.3}$$

and the motion of particles is governed by Newton's equation.

$$m_p \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F} \tag{1.4}$$

In dilute flow, the dominant force acting on the particle is drag from the fluid phase: we neglect particles Magnus force (assuminging that particle rotation is small compare to particle translation) and other forces such as added mass, Basset history term, buoyancy force.

$$\sum \mathbf{F} = \mathbf{F_D} + m_p \mathbf{g} \tag{1.5}$$

The particle Reynolds number is defined as

$$\mathrm{Re}_P = \frac{\rho_f D_p |\mathbf{U} - \mathbf{U_p}|}{\mu_f} \tag{1.6}$$

The drag force can be expressed as

$$\mathbf{F_D} = -m_p \frac{\mathbf{U_p} - \mathbf{U}}{\tau_P} \tag{1.7}$$

The relaxation time τ_p of the particles is the time it takes for a particle to respond to changes in the local flow velocity. In the case of particles with very small diameter (Stokes regime, $Re_p < 0.1$), the particle relaxation time is

$$\tau_p = \frac{\rho_p D_p^2}{18\mu_f} \tag{1.8}$$

For higher particle Reynolds number, the drag force is overpredicted by this relation and a better estimation is given by

$$\tau_p = \frac{4}{3} \frac{\rho_p D_p}{\rho_f C_D |\mathbf{U} - \mathbf{U_p}|} \tag{1.9}$$

where the standard definition of the drag coefficient C_D is

$$C_D = \begin{cases} \frac{24}{Re_p}(1 + \frac{1}{6} Re_p^{2/3}) = \frac{24}{Re_p} f_D & \text{if } \mathrm{Re}_p \leq 1000 \\ 0.44 & \text{if } \mathrm{Re}_p > 1000 \end{cases} \tag{1.10}$$

Since the fluid velocity $\mathbf{U}$, calculated in the Eulerian frame, is needed for the calculation of the drag force in the Lagrangian frame, it has to be interpolated at the position of the particle from the neighbour grid points :$\mathbf{U_{@P}}$. Finally the velocity of the particle is calculated using equations (1.4), (1.5) and (1.7) :

$$\mathbf{U_P^{\Delta t+1}} = \frac{\mathbf{U_P^{\Delta t}} + \mathbf{U_{@P}} \frac{dt}{\tau_p} + \mathbf{g} dt}{1 + \frac{dt}{\tau_p}} \tag{1.11}$$

and the position of the particle is evaluated using equation (1.3):

$$\mathbf{x_P^{\Delta t+1}} = \mathbf{x_P^{\Delta t}} + \mathbf{U_P^{\Delta t+1}} dt \tag{1.12}$$

1.3 Definitions

1.3.1 1-,2-,4- way coupling

- In the case of a dilute suspension ($\frac{\mathbf{x_{P_i}} - \mathbf{x_{P_j}}}{D_P} > 10$) with a volume fraction of particles lower than 10^{-6}, the particles' effects on turbulence are negligible. This is one-way coupling: the flow affects the particles but the particles don't affect the flow (the additional source term in the momentum equation is neglected).
- When the volume fraction is higher (in $[10^{-6}, 10^{-3}]$) the particles enhance turbulence (if $\tau_p/\tau_k > 10^2$) or dissipation (if $\tau_p/\tau_k < 10^2$) (where τ_k is the Kolmogorov time scale). This is two-way coupling and the soucre term is added in the momentum equation.
- For a dense suspension ($\frac{\mathbf{x_{P_i}} - \mathbf{x_{P_j}}}{D_P} < 10$) the particle-particle interactions must also be taken into account. Collision modelling is described in the next part.

1.3.2 Spray, cloud and parcels

- A spray is a cloud of parcels.
- A cloud is a collection of lagrangian particles.
- A parcel is a computational particle. It can be difficult to track a very large amount of particles. So a smaller number of computational particles are chosen to represent the actual particles. If a computational particle represents n physical particles, we only need to track N/n computational particles instead of tracking N particles. It is assumed that a parcel moves trough the field with the same velocity as a single physical particle. All particles within a parcel have the same properties.

1.4 Collision

Two particles P_i and P_j will collide with a certain probability if their trajectories intersect within the lagrangian time step and if their relative displacement is larger than the distance between them ie

$$\begin{cases} \mathbf{U_{ij}} = (\mathbf{U_{P_i}} - \mathbf{U_{P_j}})\mathbf{n_{ij}} > 0 \\ \mathbf{U_{ij}}\Delta t > |\mathbf{x_{P_j}} - \mathbf{x_{P_i}}| - \frac{1}{2}(D_{P_i} + D_{P_j}) \end{cases} \tag{1.13}$$

where the velocity and the unit normal vector are defined as

$$\begin{aligned} \mathbf{U}_{P_i} &= U^n_{P_i}\mathbf{n_{ij}} + U^t_i\mathbf{t_{ij}} \\ \mathbf{n_{ij}} &= \frac{\mathbf{x_{P_j}} - \mathbf{x_{P_i}}}{|\mathbf{x_{P_j}} - \mathbf{x_{P_i}}|} \end{aligned} \tag{1.14}$$

If the particle rotation is neglected, we can assume that the tangential component of the velocities does not change after the collision. Then the velocities $U^{n'}_P$ of the particles along the normal direction are evaluated using results for one-dimensional inelastic collision: we write the conservation of momentum and define the coefficient of restitution of the particle $\epsilon_P = -\frac{\mathbf{U^{n'}_{P_j}} - \mathbf{U^{n'}_{P_i}}}{\mathbf{U_{P_j}} - \mathbf{U_{P_i}}}$ which account for the energy lost during the dissipative collision.

$$U^{n'}_{P_i} = \frac{m_{P_i}U^n_{P_i} + m_{P_j}U^n_{P_j} + \epsilon_P m_{P_j}(U^n_{P_j} - U^n_{P_i})}{m_{P_i} + m_{P_j}} \tag{1.15}$$

The particle velocity after a collision with the wall is evaluated in the same way by

$$U^{n'}_P = \epsilon_w U^n_P \tag{1.16}$$

where $\epsilon_w \in [0,1]$ is the coefficient of restitution of the wall. The tangential component of the velocity will decrease after the collision with the wall

$$U^{t'}_P = (1 - \mu_w)U^t_P \tag{1.17}$$

where $\mu_w \in [0,1]$ is the coefficient of friction of the wall.

Chapter 2

OpenFOAM

Within OpenFOAM, lagrangian particle tracking is used to track spray like in DieselFoam, a solver for diesel spray and combustion.

icoLagragianFoam is available on the wiki page

`http://openfoamwiki.net/index.php/Contrib_icoLagrangianFoam`

where the description is

```
The particle code is a heavily lobotomized version of stuff found in the dieselSpray
 classes. It features:

    * a simple random injector
    * a drag force model that is horrible and not very physical
    * particles can bounce from walls or die (switchable)
    * particles leave the system at in or outlet. All other boundary types are not
    treat correctly
    * the particles can add a source term to the moment equation of the gas (switchable)
    * there is no particle-particle interaction

This solver is not to be used for simulations that resemble the real world.
 It's just a demo.
```

In this chapter we compare the way icolagrangianFoam and dieselFoam solve the equation described in Chapter 1.

2.1 Momentum equation (1.1)

2.1.1 icoLagrangianFoam : icoLagrangianFoam.C

```
fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
            == cloud.momentumSource()
        );

        solve(UEqn == -fvc::grad(p));
```

This corresponds to equation (1.1) divided by the density. Gravity is neglected and there is no turbulence. icolagrangianFoam is based on icoFoam which is a transient solver for incompressible, laminar flow of Newtonian fluids.

2.1.2 dieselFoam : dieselEngineFoam/UEqn.H

```
fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      + turbulence->divDevRhoReff(U)
     ==
        rho*g
      + dieselSpray.momentumSource()
    );

    if (momentumPredictor)
    {
        solve(UEqn == -fvc::grad(p));
    }
```

where divDevRhoReff is the deviatoric stress tensor defined by

```
divDevRhoReff =- fvm::laplacian(muEff(), U)- fvc::div(muEff()*dev2(fvc::grad(U)().T()))
```

This is similar to equation (1.1) for laminar or turbulent flow.

2.2 equation (1.2)

2.2.1 icoLagrangianFoam : IncompressibleCloudI.H

```
tsource().internalField() = smoment_/runTime_.deltaT().value()/mesh_.V()
                                                  /constProps().density_;
```

where `smoment` is calculated in the function `move` of the class HardBallParticle. (in HarBall-Particle.C)

```
vector oMom=U()*m();
updateProperties(deltaT,data,cellI,face());
vector nMom=U()*m();
data.cloud().smoment()[cellI] += oMom-nMom;
```

This corresponds to equation (1.2) divided by ρ to be consistent with the momentum equation.

2.2.2 dieselFoam : lagrangian/dieselSpray/lnInclude/sprayI.H

```
tsource().internalField() = sms_/runTime_.deltaT().value()/mesh_.V();
```

where `sms` is calculated in the function `move` of the class parcel. (in src/lagrangian/diesel-Spray/parcel/parcel.C)

```
vector oMom = m()*U();
// update the parcel properties (U, T, D)
updateParcelProperties(dt,sDB,celli,face());
vector nMom = m()*U();
// Update the Spray Source Terms
sDB.sms()[celli]   += oMom - nMom;
```

This is similar to equation (1.2)

2.3 equation (1.11)

2.3.1 icoLagrangianFoam : HardBallParticle.C

```
vector Upos=data.UInterpolator().interpolate(position(),cellI,faceI);
scalar coeff=dt/relax;
U()=( U() + coeff*Upos + data.constProps().g()*dt)/(1. + coeff);
```

This is similar to equation (1.11) with $Upos = U@P$ and $relax = \tau_p$

2.3.2 dieselFoam : lagrangian/dieselSpray/parcel/parcel.C

```
vector Up = sDB.UInterpolator().interpolate(position(), celli, facei)+ Uturb();
scalar timeRatio = dt/tauMomentum;
U() = (U() + timeRatio*Up + sDB.g()*dt)/(1.0 + timeRatio);
```

This is similar to equation (1.11) with $U_p = U@P$ and $tauMomentum = \tau_p$

2.4 equations 1.8 and 1.9

2.4.1 icoLagrangianFoam : HardBallParticle.C

```
relax=1/(data.constProps().dragCoefficient()*(d_*d_)/mass_);
```

The relaxation time is defined as $\tau_p = \frac{m}{C_D d^2}$ witch doesn't correspond to the relaxation time defined above. We propose a modification of $relax$ in Chapter 5.

2.4.2 dieselFoam : lagrangian/dieselSpray/parcel/setRelaxationTimes.C

```
tauMomentum = sDB.drag().relaxationTime(Urel(Up),d(),rho,liquidDensity, nuf,dev());
```

The relaxationTime function is described in src/lagrangian/dieselSpray/spraySubModels/dragModel/standardDragModel/standardDragModel.C

```
scalar standardDragModel::relaxationTime
    scalar Re = mag(URel)*diameter/nu;
    if (Re > 0.1)
    {
        time = 4.0*liquidDensity*diameter / (3.0*rho*Cd(Re, dev)*mag(URel));
    }
    else
    {
         time = liquidDensity*diameter*diameter/(18*rho*nu*(1.0 + Cdistort_*dev));
    }
```

This is similar to equations (1.8) and (1.9).

2.5 equation (1.10)

2.5.1 icoLagrangianFoam

The drag coefficient is a constant given in the dictionnary `/constant/cloudproperties` and read in HardBallParticle.C

```
dragCoefficient_(readScalar(dict.lookup("drag")))
```

2.5.2 dieselFoam : lagrangian/dieselSpray/spraySubModels/dragModel/standardDragModel/standardDragModel.C

```
scalar standardDragModel::Cd
(const scalar Re,const scalar dev) const{
    scalar drag = CdLimiter_;
    if (Re < ReLimiter_)
    {drag =  24.0*(1.0 + preReFactor_*pow(Re, ReExponent_))/Re;
    }
    return drag;
}
```

The coefficients are defined in a dictionnary in `/constant/sprayProperties`

```
standardDragModelCoeffs
{
    preReFactor     0.166667;
    ReExponent      0.666667;
    ReLimiter       1000;
    CdLimiter       0.44;
    Cdistort        2.632;
}
```

This is similar to equations (1.10)

2.6 equation (1.12)

This equation is used both in icoLagrangianFoam (HardBallParticle.C) and dieselFoam (lagrangian/dieselSpray/parcel/parcel.C)

```
          // set the lagrangian time-step
            scalar dt = min(dtMax, tEnd);
          // Track and adjust the time step if the trajectory is not completed
            dt *= trackToFace(position() + dt*U_, sDB);
          // Decrement the end-time acording to how much time the track took
            tEnd -= dt;
            // Set the current time-step fraction.
             stepFraction() = 1.0 - tEnd/deltaT;
```

The function trackToFace (described in Particle.C) tracks particle to a given position and returns 1.0 if the trajectory is completed without hitting a face otherwise stops at the face and returns the fraction of the trajectory completed.

2.7 equation (1.16)

This equation is used both in icoLagrangianFoam (HardBallParticle.C) and dieselFoam (lagrangian/dieselSpray/spraySubModels/wallModel/reflectParcel/reflectParcel.C)

```
           vector Sf = mesh_.Sf().boundaryField()[patchi][facei];
           Sf /= mag(Sf);
             scalar Un = p.U() & Sf;
            if (Un > 0)
            {
                 p.U() -= (1.0 + elasticity_)*Un*Sf;
             }
```

2.8 equation (1.15)

- In icoLagrangianFoam the particle collision is not implemented.
- Two particle collision models are avalaible in

  `lagrangian/dieselSpray/spraySubModels/collisionModel`.

  In O'rourk model collision occurs if particles are in the same cell, even if they are not moving towards each other. The trajectory model described in the previous chapter is presented here.

```
vector v1 = p1().U();
vector v2 = p2().U();
vector vRel = v1 - v2;
scalar prob = rndGen_.scalar01();
gf = sqrt(prob)
n1 = p1().N(rho1);
n2 = p2().N(rho2);
vector p = p2().position() - p1().position();
scalar v1p = nv1 & p;
scalar v2p = nv2 & p;
scalar m1 = p1().m();
scalar m2 = p2().m();
vector mr = m1*v1 + m2*v2;
vector v1p = (mr + m2*gf*vRel)/(m1+m2);
vector v2p = (mr - m1*gf*vRel)/(m1+m2);
 if (n1 < n2) {
p1().U() = v1p;
p2().U() = (n1*v2p + (n2-n1)*v2)/n2;
 }
else {
 p1().U() = (n2*v1p + (n1-n2)*v1)/n1;
 p2().U() = v2p;
}
```

Here the coefficient of restitution is not a fixed value. It is a random scalar.

2.9 equations (1.16)

This equation is used both in icoLagrangianFoam (in the function move of HardBallParticle) and dieselFoam (called in the function move of parcel (with boundaryTreatment.H and described in lagrangian/dieselSpray/spraySubModels/wallModel/reflectParcel))).

```
     // wallNormal defined to point outwards of domain
        vector Sf = mesh_.Sf().boundaryField()[patchi][facei];
        Sf /= mag(Sf);
        scalar Un = p.U() & Sf;
        if (Un > 0)
         {
             p.U() -= (1.0 + elasticity_)*Un*Sf;
         }
```

Chapter 3

Update icoLagrangianFoam for OpenFOAM-1.6

3.1 createParticles.H

Change from

```
volPointInterpolation vpi(mesh, pMesh);
```

to

```
volPointInterpolation vpi(mesh);
```

3.2 HardBallParticle.H

Add

```
bool hitPatch
(
    const polyPatch&,
    HardBallParticle::trackData& td,
    const label patchI
);
bool hitPatch
(
    const polyPatch& p,
    int& td,
    const label patchI
);
```

3.3 HardBallParticle.C

Add

```
bool Foam::HardBallParticle::hitPatch
(
    const polyPatch&,
    HardBallParticle::trackData&,
    const label
)
```

```
 {
     return false;
 }
 bool Foam::HardBallParticle::hitPatch
 (
     const polyPatch&,
     int&,
     const label
)
 {
     return false;
 }
```

In order to enable reading of the position of the particles at restart, change from

```
       Particle<HardBallParticle>(cloud, is)
```

to

```
       Particle<HardBallParticle>(cloud, is,readFields)
```

3.4 HardBallParticleIO.C

Change in `void HardBallParticle::writeFields(const IncompressibleCloud &c)` from

```
     IOField<scalar> d(c.fieldIOobject("d"),np);
     IOField<scalar> m(c.fieldIOobject("m"),np);
     IOField<vector> U(c.fieldIOobject("U"),np);
```

to

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

Change in `void HardBallParticle::readFields(IncompressibleCloud &c)` from

```
     IOField<scalar> d(c.fieldIOobject("d"));
     IOField<scalar> m(c.fieldIOobject("m"));
     IOField<vector> U(c.fieldIOobject("U"));
```

to

```
     IOField<scalar> d(c.fieldIOobject("d", IOobject::MUST_READ));
     IOField<scalar> m(c.fieldIOobject("m", IOobject::MUST_READ));
     IOField<vector> U(c.fieldIOobject("U", IOobject::MUST_READ));
```

3.5 IncompressibleCloud.C

```
 autoPtr<interpolation<vector> > UInt = interpolation<vector>::New
       (
           interpolationSchemes_,
//remove     volPointInterpolation_,
           U_
       );
```

Chapter 4

pisoLagrangianFoam

To be able to model the lagrangian particle tracking and a turbulent flow we need to include the LPT files from icoLagrangianFoam in the application pisoFoam. pisoFoam is a transient solver for incompressible flow where turbulence modelling is generic, i.e. laminar, RAS or LES may be selected. We call pisoLagrangianFoam the LPT solver based on pisoFoam.

```
cd user-1.6/applications
cp -r OpenFOAM/OpenFOAM-1.6/applications/solvers/incompressible/pisoFoam .
cp icoLagrangianFoam/*Particle* pisoFoam/
cp icoLagrangianFoam/IncompressibleCloud* pisoFoam/
mv pisoFoam pisoLagrangianFoam
cd pisoLagrangianFoam

sed 's/#include "turbulenceModel.H" /
#include "turbulenceModel.H"
#include "HardBallParticle.H"
#include "IncompressibleCloud.H"
/g' icoLagrangianFoam.C > temp1

sed 's/ #include "createFields.H" /
#include "createFields.H"
#include "createParticles.H"
 /g' temp1 > temp2

sed 's/ #include "CourantNo.H" /
#include "CourantNo.H"
#include "moveParticles.H"
 /g' temp2 > temp3

sed 's/ + turbulence->divDevReff(U) /
+ turbulence->divDevReff(U)
== cloud.momentumSource()
/g' temp3 > pisoLagrangianFoam.C

sed 's/pisoFoam.C /
pisoLagrangianFoam.C
HardBallParticle.C
IncompressibleCloud.C
HardBallParticleIO.C
IncompressibleCloudIO.C
/g' Make/files > Make/temp4
```

```
sed '/(FOAM_APPBIN)/pisoFoam /
(FOAM_USER_APPBIN)/pisoLagrangianFoam
/g' Make/temp4 > Make/files

sed 's/-I$(LIB_SRC)/finiteVolume/lnInclude /
-I$(LIB_SRC)/finiteVolume/lnInclude \
-I$(LIB_SRC)/lagrangian/basic/lnInclude
/g' Make/options > Make/temp5

sed 's/-lfiniteVolume /
-lfiniteVolume \
-llagrangian
/g' Make/temp5 > Make/files

rm temp*
wmake
```

Chapter 5

Improvement in icolagrangianFoam

5.1 Particle injection

The function inject is defined in IncompressibleCloud.C. It injects one particle at each time step between tStart and tEnd, only if a calculated random number (in $[0, 1]$) is smaller than $thres$. $tSart, tEnd$ and $thres$ are defined by the user in a dictionary (`constant/cloudProperties`). The position of the particle injected is in a sphere of $center$ and radius $r0$ defined by the user. The initial velocity and diameter of the particle are $U_p = vel1 + randomscalar * vel0$ and $D_p = randomscalar * d1 + d0$ where $vel0, vel1, d0, d1$ are also defined by the user.

To be able to inject several particles at each time step, we make the following changes:

```
//      scalar prop=random().scalar01();
//      if(prop<td.constProps().thres_) {
        for (label iter=1; iter<=td.constProps().nbInjByDt_; iter++) {
```

where $nbInjByDt$ is the number of particles injected at each time step, defined in `cloudProperties` and read in HardBallParticle.C

```
//        thres_(readScalar(dict.subDict("injection").lookup("thres"))),
         nbInjByDt_(readScalar(dict.subDict("injection").lookup("nbInjByDt"))),
```

In HardBallParticle.H, we add

```
// scalar thres_;
scalar nbInjByDt_;
```

5.2 Drag and relaxation Time

In order to have a more reliable value for the relaxation time, we make the following change in HardBallParticle.C:

```
//     scalar relax=1/(data.constProps().dragCoefficient()*(d_*d_)/mass_);
     scalar Rep=data.rho_*mag(U()-Upos)*d_/data.mu_;
     scalar Cd=0.44;
     if (Rep<1000)
     {Cd=24.0/Rep*(1.0+1.0/6.0*pow(Rep,2/3)); }
     scalar relax =GREAT;
     if (Rep>0.1)
     {relax =4/3*data.constProps().density_*d_/(data.rho_*Cd*mag(U()-Upos));}
```

```
else
{ relax =data.constProps().density_*d_*d_/(18*data.mu_);}
```

In order to evaluate the particle Reynolds number and the relaxation time, we introduced references to the density and viscosity of the physical field. Therefore we also need to include rho and mu as member of the class trackData. This implies the following changes:

- in HardBallParticle.H:

```
public:
         trackData(
             IncompressibleCloud &cloud,
             interpolation<vector> &Uint_,
             scalar &rho_,                                     //added
             scalar &mu_                                       //added
         );
         IncompressibleCloud &cloud() { return cloud_; }
         scalar &rho_;                                         //added
         scalar &mu_;                                          //added
```

- in HardBallParticle.C:

```
HardBallParticle::trackData::trackData(
        IncompressibleCloud &cloud,
        interpolation<vector> &Uint,
        scalar &rho,                                           //added
        scalar &mu                                             //added
    )
        :
        Particle<HardBallParticle>::trackData(cloud),
    cloud_(cloud),
    constProps_(cloud.constProps()),
    wallCollisions_(0),
    leavingModel_(0),
    injectedInModel_(0),
    changedProzessor_(0),
    UInterpolator_(Uint),
    rho_(rho),                                                 //added
    mu_(mu)                                                    //added
    {
    }
```

- in IncompressibleCloud.C, inside the part `Construct from components`

```
  IncompressibleCloud::IncompressibleCloud(
               const volPointInterpolation& vpi,
               const volVectorField& U,
               scalar& rho,                                    //added
               scalar& mu)                                     //added
:
rho_(rho),                                                     //added
mu_(mu),                                                       //added
```

- in IncompressibleCloud.C, inside the function evolve()

```
HardBallParticle::trackData td(*this,UInt(),rho_,mu_);
```

- in IncompressibleCloud.H

```
// References to the physical fields
    const volVectorField& U_;
    scalar& rho_;                                              //added
    scalar& mu_;                                               //added
// Constructors
IncompressibleCloud(
                        const volPointInterpolation& vpi,
                        const volVectorField& U,
                        scalar& rho_,                          //added
                        scalar& mu_                            //added
                    );
```

- In createFields.H add

```
scalar rho(readScalar(transportProperties.lookup("rho")));
scalar mu(readScalar(transportProperties.lookup("mu")));
```

5.3 Volume fraction of particles

For postprocessing purpose we store the position, velocity, mass and diameter for each particle. Particles can be seen using foamToVTK, paraview and the glyph utility. But this is not convenient for a case with a large number of particles. By introducing a volScalarField $volFrac$ that represents the volume fraction of particles in each cell, we can simply see the particle distribution in the domain with paraFoam. The volume fraction is $volfrac = nbP.V_p/\Delta V$, where nbP is the number of particles in a unit volume of fluid ΔV and V_p is the volume of a particle.

- In IncompressibleCloud.H, add

```
    scalarField nbPVp_;
    scalarField &nbPVp() { return nbPVp_; }
    inline tmp<volScalarField> volFrac() const;
```

- In IncompressibleCloudI.H, add

```
inline tmp<volScalarField> IncompressibleCloud::volFrac() const
{
    tmp<volScalarField> vF
    (
        new volScalarField
        (
            IOobject
            (
                "vF",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            mesh_,
            dimensionedScalar
```

```
                (
                    "vF",
                    dimless,
                   0.0
                )
            )
        );
            vF().internalField() = nbPVp_/mesh_.V();

        return vF;
    }
```

- In HardBallParticle.C, in the function `move` (just before the return statement) add

```
        if (data.keepParticle)
        {
         data.cloud().nbPVp()[cell()]+=(4/3*3.14*pow(d(),3)/8);
        }
```

- The volScalaField `volfrac` is defined in createParticles.H by adding

```
    volScalarField volfrac
    (
        IOobject
        (
            "volfrac",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
```

- And it is updated in icoLagrangianFoam.C

```
#include "moveParticles.H"
volfrac=cloud.volFrac();  //added
```

Chapter 6

Improvement in solidParticle

The solidParticle class has been recently introduced in OpenFOAM. It is a one-way coupling LPT which tracks a defaultCloud and uses the same libraries than dieselFoam. There is no injetor. In this part we describe how to add an injector and the scalarField `volFrac` as we did in icoLagrangianFoam in the previous chapter.

6.1 Particle injection

```
cd user-1.6/applications
cp -r OpenFOAM/OpenFOAM-1.6/src/lagrangian/solidParticle .
mv solidParticle mySolidParticleFoam
```

Create the directory Make, the files solidParticleFoam.C and createFields.H exactly like in the solver solidParticleFoam available at
`http://openfoamwiki.net/index.php/Contrib_solidParticleFoam`

- In solidParticleCloud.C

```
mu_(dimensionedScalar(particleProperties_.lookup("mu")).value()),
nbInjByDt_(dimensionedScalar(particleProperties_.lookup("nbInjByDt")).value()),
center_(dimensionedVector(particleProperties_.lookup("center")).value()),
r0_(dimensionedVector(particleProperties_.lookup("r0")).value()),
d_(dimensionedScalar(particleProperties_.lookup("d")).value()),
vel_(dimensionedVector(particleProperties_.lookup("vel")).value()),
velprime_(dimensionedScalar(particleProperties_.lookup("velprime")).value()),
tInjStart_(dimensionedScalar(particleProperties_.lookup("tInjStart")).value()),
tInjEnd_(dimensionedScalar(particleProperties_.lookup("tInjEnd")).value()),
random_(666)
```

- In solidParticleCloud.C, in the function `move`

```
Cloud<solidParticle>::move(td);
if(mesh_.time().value()>td.spc().tInjStart_ &&
   mesh_.time().value()<td.spc().tInjEnd_)
  {
   this->inject(td);
  }
```

- In solidParticleCloud.C, add the function `inject`

```
void Foam::solidParticleCloud::inject(solidParticle::trackData &td) {
 for (label nbP=1; nbP<=td.spc().nbInjByDt(); nbP++) {
    vector tmp=(random().vector01()- vector(0.5,0.5,0.5))*2;
    vector center=td.spc().center();
    vector r0=td.spc().r0();
    scalar posx=tmp.x()*r0.x();
    scalar posy=tmp.y()*r0.y();
    scalar posz=tmp.z()*r0.z();
    vector pos=center+vector(posx,posy,posz);
    vector tmpv=vector(random().GaussNormal(),
    random().GaussNormal(),random().GaussNormal())/sqrt(3.);
    vector vel=tmpv*td.spc().velprime()+td.spc().vel();
    label cellI=mesh_.findCell(pos);
    if(cellI>=0)
     {
     solidParticle* ptr= new solidParticle(*this,pos,cellI,td.spc().d(),vel);
     Cloud<solidParticle>::addParticle(ptr);
     }
  }
}
```

- In solidParticleCloud.H

```
#include "IOdictionary.H"
#include "Random.H"

 scalar mu_;
 scalar nbInjByDt_;
 vector center_;
 vector r0_;
 scalar d_;
 vector vel_;
 scalar velprime_;
 scalar tInjStart_;
 scalar tInjEnd_;
 Random random_;

 inline scalar mu() const;
 inline scalar nbInjByDt() const;
 inline vector center() const;
 inline vector r0() const;
 inline scalar d() const;
 inline vector vel() const;
 inline scalar velprime() const;
 Random &random() {return random_;}

 void inject(solidParticle::trackData &td);
```

- Add in solidParticleCloudI.H

```
 inline Foam::scalar Foam::solidParticleCloud::nbInjByDt() const
 {
     return nbInjByDt_;
 }
```

```
    inline Foam::vector Foam::solidParticleCloud::center() const
    {
        return center_;
    }
    inline Foam::vector Foam::solidParticleCloud::r0() const
    {
        return r0_;
    }
    inline Foam::scalar Foam::solidParticleCloud::d() const
    {
        return d_;
    }
    inline Foam::vector Foam::solidParticleCloud::vel() const
    {
        return vel_;
    }
    inline Foam::scalar Foam::solidParticleCloud::velprime() const
    {
        return velprime_;
    }
```

6.2 Drag and relaxation Time

The inverse of the relaxation time ($Dc = 1/\tau_p$) is defined in solidParticle.C

```
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));
```

In the first chapter the threshold value of Re_p was 0.1 and not 0.01. It is important to correct this misprint in order to get a good estimation of the drag coefficient.

6.3 Volume fraction of particles

- In soliParticle.C, at the end of the function `move`

```
if (td.keepParticle)
 {
  label cellnew = cell();
  td.spc().nbPVp()[cellnew]+=(4/3*3.14*pow(d_,3)/8);
 }
```

- In soliParticleCloud.C

```
random_(666),
nbPVp_(mesh_.nCells(), 0.0)

interpolationCellPoint<scalar> nuInterp(nu);
nbPVp_ = 0.0;
```

- In soliParticleCloud.H

```
 scalar mu_;
 scalarField nbPVp_;

inline scalar mu() const;
scalarField &nbPVp() { return nbPVp_; }
inline tmp<volScalarField> volFrac() const;
```

- Add in solidParticleCloudI.H

```
 namespace Foam
 {
 inline tmp<volScalarField> solidParticleCloud::volFrac() const
  {
      tmp<volScalarField> vF
      (
          new volScalarField
          (
              IOobject
              (
                  "vF",
                  //runTime_.timeName(),
                  mesh_,
                  IOobject::NO_READ,
                  IOobject::AUTO_WRITE
              ),
              mesh_,
              dimensionedScalar
              (
                  "vF",
                  dimless,
                 0.0
              )
          )
      );
          vF().internalField() = nbPVp_/mesh_.V();

      return vF;
  }
 }
```

- The volScalaField `volfrac` is defined in createFields.H by adding

```
    volScalarField volfrac
    (
        IOobject
        (
            "volfrac",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
```

- And it is updated in solidParticleFoam.C

```
particles.move(g);
volfrac= particles.volFrac();                    //added
runTime.write();
```

Chapter 7

Test case

In this part we use the injector of particles in the tutorial case pitzDaily, and we plot volfrac to see the particles distribution in ParaFoam.

The tutorial pitzdaily illustrates the solver simpleFoam which is a solver for incompressible fluid. We merge simpleFoam and solidParticleFoam into mySolidParticleSimpleFoam.

- copy the tutorial pitzDaily,
- add the file 0/volFracorg (cp 0/p 0/volFracorg and change dimension to [0 0 0...])
- add the file constant/g

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

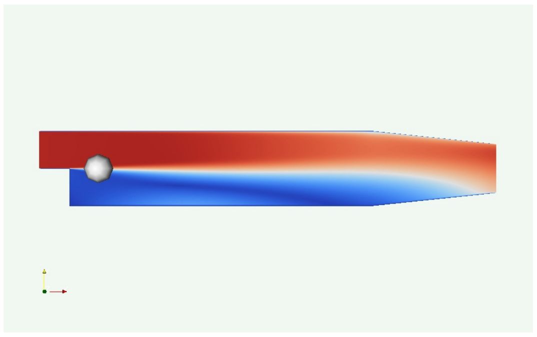
- add density in constant/transportProperties

```
 rho             rho [ 1 -3 0 0 0 0 0 ] 1000;
```

- add constant/particleProperties

```
rhop rhop [ 1 -3  0  0  0  0  0] 1000;
e    e    [ 0  0  0  0  0  0  0] 0.8;
mu   mu   [ 0  0  0  0  0  0  0] 0.2;
nbInjByDt nbInjByDt [ 0  0  0  0  0  0  0] 500;
center center [ 0  1  0  0  0  0  0] (0.02 0.0 0.0);
r0 r0 [ 0  1  0  0  0  0  0] (0.01 0.01 0.0);
d d [ 0  1  0  0  0  0  0]  5e-5;
vel vel [ 0  1 -1   0  0  0  0] (0  0 0);
velprime velprime [ 0  0  0   0  0  0  0] 0 ;
tInjStart tInjStart [ 1 0   0  0  0  0  0] 1000;
tInjEnd tInjEnd [ 1 0   0  0  0  0  0] 5000;
```

The particles are injected from a disc of center (0.02,0,0), as shown here:



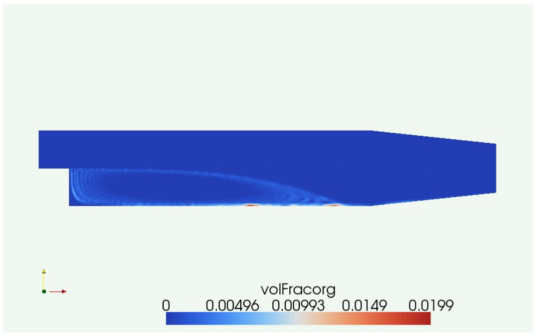
disc of injection of particles, velocity at t=1000

The lengh of the domain is 0.311m. The maximum velocity in this direction is 10.1m/s. It means that a particle following such a stream line will leave the domain 0.03s after its injection. As we want to follow the path of the particles, we need to have a time step much smaller than 0.03s. We choose 0.003s, and write the results every time step, untill t=1001. We start the computation and the particles injection at t=1000, ie the flow already converged to a steady state solution.

- change system/controlDict

```
startFrom       startTime;
startTime       1000;
stopAt          endTime;
endTime         1001;
deltaT          0.003;
writeControl    timeStep;
writeInterval   1;
```

The result show that a larger number of particles follow the stream lines in the region of low velocity and recirculation.



distribution of the volume fraction of particles at t=1001