Strömningsteknik - Energivetenskaper

LUND TEKNISKA HÖGSKOLA

CFD WITH OPENSOURCE SOFTWARE, ASSIGNMENT 3

Tutorial icoLagrangianFoam / solidParticle

Author:
AURÉLIA VALLIER

Peer reviewed by:
JAN POTAC
HÅKAN NILSSON

January 11, 2010

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

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

$$\rho_f \frac{\partial \mathbf{U}}{\partial t} + \rho_f(\mathbf{U} \cdot \nabla \mathbf{U}) = -\nabla p + \mu_f \nabla^2 \mathbf{U} + \rho g - \mathbf{S_P} \tag{1.1}$$

where $\mathbf{U}$, p, ρ_f and μ_f are the velocity, pressure, density and viscosity of the fluid. The additional source term $\mathbf{S_P}$ in the momentum equation (1.1) is due to the influence of the particles on the fluid. Take a particle P of mass m_P and velocity $\mathbf{U_p}$. The force exerted by a particle on a unit volume of fluid is proportional to the difference in particle momentum between the instant it enters (t_{in}) and leaves (t_{out}) the control volume:

$$m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}}). \tag{1.2}$$

A momentum source contribution is generated by the particle P in each cell visited along its path during one eulerian timestep dt. The contribution of all the particles which have been in the cell k (of volume V_k) during the eulerian time step dt is written as

$$S_P@cellk = \frac{1}{V_k dt} \sum_P m_P((\mathbf{U_p})_{t_{outcellk}} - (\mathbf{U_p})_{t_{incellk}}). \tag{1.3}$$

To explain how this contribution is calculated for each lagrangian time step Δt within one eulerian time step dt, we give 2 exemples.

- Exemple 1: Take a particle P, with velocity Ua, at initial position A in the cell cellA such that its position B after the time dt is still in the cell cellA. The particle doesn't hit a face. The velocity of this particle at the position B is updated (see next part) and called Ub. The contribution of this particle on the cell cellA is $\frac{m_P}{V_{cellA} dt}(\mathbf{Ub} - \mathbf{Ua})$.
- Exemple 2: Take a particle P, with velocity Ua, at initial position A in the cell cellA such that its position B after the time dt should be in the cell cellB, neighboor with cellA. The particle has to cross a face at the posistion F. AF is the fraction of trajectory completed to reach the position F. We call stepfraction the fraction of the eulerian time-step completed when the particle reach the position F. The eulerian time step dt has to be divided in

2 lagrangian time steps, one to cover the distance AF (Δt=stepfration) and one to cover the distance FB (Δt=dt-stepfraction). The velocity of the particle at the position F is updated (see next part) and called Uf. The contribution of the particle on the cell cellA is $\frac{m_P}{V_{cellA}dt}(\mathbf{Uf}-\mathbf{Ua})$. Then the particle moves to the position B, its velocity Ub is evaluated and the contribution of this particle on the cell cellB is $\frac{m_P}{V_{cellA}dt}(\mathbf{Ub}-\mathbf{Uf})$.

In practice the particle trajectory cross several cells and each Eulerian time step is divided in a set of Lagrangian time steps that is specific to each particle.

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, its velocity $\mathbf{U_P}$ and its 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

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P} \tag{1.4}$$

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

$$m_p \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}. \tag{1.5}$$

In dilute flow, the dominant force acting on the particle is drag $\mathbf{F_D}$ 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.

$$\sum \mathbf{F} = \mathbf{F_D} + m_p \mathbf{g}. \tag{1.6}$$

The particle Reynolds number is defined as

$$\mathrm{Re}_P = \frac{\rho_f D_p |\mathbf{U}-\mathbf{U_p}|}{\mu_f}. \tag{1.7}$$

The drag force can be expressed as

$$\mathbf{F_D} = -m_p \frac{\mathbf{U_p}-\mathbf{U}}{\tau_P}. \tag{1.8}$$

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.

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

where the standard definition of the drag coefficient C_D is

$$C_D = \begin{cases} \frac{24}{Re_p} & \text{if } \mathrm{Re}_p \leq 0.1 \text{ Stokes regime} \\ \frac{24}{Re_p}(1+\frac{1}{6}Re_p^{2/3}) & \text{if } 0.1 \leq Re_p \leq 1000 \text{ Transition regime} \\ 0.44 & \text{if } \mathrm{Re}_p > 1000 \text{ Newtonian regime} \end{cases} \tag{1.10}$$

The drag coefficient for a solid sphere at low Reynolds number is evaluated analytically by the Stokes'law (G.G. Stokes, Camb. Phil. Trans. 9, 1851). It has been extended for higher Re_p with empirical non-linear correction to take account for both viscous and inertial effects. The correlation used here is due to Schiller and Naumann (Z. Ver. Dtsh. Ing. 1933,318). For high values of Re_p, inertial effects dominate viscous effects; the drag coefficient becomes independent of Reynolds number and a constant value is normally adopted (Geurts and Vreman, 2006).

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 neighboor grid points :$\mathbf{U_{@P}}$. Finally the velocity of the particle is calculated using equations (1.5), (1.6) and (1.8) :

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

Hence the velocity of the particle is updated after the first lagrangian time step Δt with

$$\mathbf{U_p^{t+\Delta t}} = \frac{\mathbf{U_P^t} + \mathbf{U_{@P}^t} \frac{\Delta t}{\tau_p} + \mathbf{g}\Delta t}{1 + \frac{\Delta t}{\tau_p}} \tag{1.12}$$

and the position of the particle is evaluated using equation (1.4):

$$\mathbf{x_p^{t+\Delta t}} = \mathbf{x_p^t} + \mathbf{U_P^t}\Delta t \tag{1.13}$$

If the eulerian time step is completed in several lagragian time steps, the velocity and position are evaluated at the n-th lagrangian time step by:

$$\mathbf{U_p^{t+\sum_{i=1}^{n} \Delta t_i}} = \frac{\mathbf{U_P^{t+\sum_{i=1}^{n-1} \Delta t_i}} + \mathbf{U_{@P}^t} \frac{\Delta t_n}{\tau_p} + \mathbf{g}\Delta t_n}{1 + \frac{\Delta t_n}{\tau_p}} \tag{1.14}$$

$$\mathbf{x_p^{t+\sum_{i=1}^{n} \Delta t_i}} = \mathbf{x_p^{t+\sum_{i=1}^{n-1} \Delta t_i}} + \mathbf{U_P^{t+\sum_{i=1}^{n-1} \Delta t_i}} \Delta t_n \tag{1.15}$$

1.3 Definitions

1.3.1 1-,2-,4- way coupling

The concentration of particles influences the interaction between the two phases. This is classified by Elgobashi (Particle-Laden Turbulent Flows: Direct Numerical Simulation and Closure Models. Appl. Sci. Res., 48, 3-4, 301-304. 1991). In a dilute suspension the distance between 2 particles P_i and P_j is larger than 10 particle diameter: $\frac{\mathbf{x_{P_i}} - \mathbf{x_{P_j}}}{D_P} > 10$. Otherwise the suspension is called dense. The volume fraction of particles in a control volume $cellk$ of volume V_{cellk} is defined by $volfrac@cellk = \frac{NP_{cellk} V_P}{V_{cellk}}$ where NP_{cellk} is the number of particles in $cellk$ and V_P is the volume of one particle.

- In the case of a dilute suspension 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 ($volfrac \in [10^{-6}, 10^{-3}]$) 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 and τ_p is the particle relaxation time). This is two-way coupling and the soucre term is added in the momentum equation.
- For a dense suspension the particle-particle interactions must also be taken into account. This is called four-way coupling. Collision modelling is described in the next part.

1.3.2 Spray, cloud and parcels

- 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.

- A cloud is a collection of lagrangian particles.
- A spray is a cloud of parcels.

1.4 Collision

1.4.1 Collision between two particles

We consider two particles P_i and P_j. We define a unit normal vector from particle P_i to particle P_j:

$$\mathbf{n_{i->j}} = \frac{\mathbf{x_{P_j}} - \mathbf{x_{P_i}}}{|\mathbf{x_{P_j}} - \mathbf{x_{P_i}}|} \tag{1.16}$$

and $\mathbf{t_{i->j}}$ is the unit vector in the tangential direction (i.e. orthogonal to $\mathbf{n_{i->j}}$ on the plan where the collision occurs). The velocity of the particle P_i is written as

$$\mathbf{U}_{P_i} = U^n_{P_i}\mathbf{n_{i->j}} + U^t_{P_i}\mathbf{t_{i->j}} \tag{1.17}$$

Two particles P_i and P_j will collide with a certain probability if

- their trajectories intersect within the lagrangian time step i.e. $(\mathbf{U_{P_i}} - \mathbf{U_{P_j}})\mathbf{n_{i->j}} > 0$
- and their relative displacement is larger than the distance between them i.e. $(\mathbf{U_{P_i}} - \mathbf{U_{P_j}})\mathbf{n_{i->j}}\Delta t > |\mathbf{x_{P_j}} - \mathbf{x_{P_i}}| - \frac{1}{2}(D_{P_i} + D_{P_j})$

If the particle rotation is neglected, we can assume that the tangential component U^t_P of the velocities does not change after the collision. The velocity of the particle P_i after collision is

$$\mathbf{U}^{'}_{P_i} = U^{n'}_{P_i}\mathbf{n_{i->j}} + U^t_{P_i}\mathbf{t_{i->j}} \tag{1.18}$$

The post-collision 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 ($\frac{1}{2}m_{P_i}(U^n_{P_i})^2 + \frac{1}{2}m_{P_j}(U^n_{P_j})^2 = \frac{1}{2}m_{P_i}(U^{n'}_{P_i})^2 + \frac{1}{2}m_{P_j}(U^{n'}_{P_j})^2$) and define the coefficient of restitution of the particle $\epsilon_P = -\frac{\mathbf{U^{n'}_{P_j}} - \mathbf{U^{n'}_{P_i}}}{\mathbf{U^{n}_{P_j}} - \mathbf{U^{n}_{P_i}}}$ which account for the energy lost during the dissipative collision. It gives:

$$U^{n'}_{P_i} = \frac{m_{P_i}U^n_{P_i} + m_{P_j}U^n_{P_j} + \epsilon_P m_{P_j}(U^n_{P_j} - U^n_{P_i})}{m_{P_i} + m_{P_j}} \tag{1.19}$$

1.4.2 Collision of a particle with the wall

In this part the unit vector $\mathbf{n}$ and $\mathbf{t}$ are respectively the unit vector normal and tangential to the wall. The velocity of the particle P after collision is

$$\mathbf{U}^{'}_P = U^{n'}_P\mathbf{n} + U^{t'}_P\mathbf{t} \tag{1.20}$$

The normal component of the particle velocity after a collision with the wall is evaluated as

$$U^{n'}_P = -\epsilon_w U^n_P \tag{1.21}$$

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

$$U^{t'}_P = (1 - \mu_w)U^t_P \tag{1.22}$$

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

The coefficients of restitution and friction are determined experimentally. They depend mainly on the materials, the surface and the impact velocity.

Chapter 2

OpenFOAM

In OpenFOAM, lagrangian particle tracking is used to track spray like in dieselFoam, a solver for diesel spray and combustion. Another alternative is icoLagragianFoam which 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 equations 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 except that it also allows for variable density.

2.2 Equations (1.2) and (1.3)

2.2.1 icoLagrangianFoam : IncompressibleCloudI.H

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

This corresponds to equation (1.3) divided by ρ to be consistent with the momentum equation in icoLagrangianFoam. And `smoment` is calculated in the function `move` of the class HardBallParticle. (in HardBallParticle.C)

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

This is similar to equation (1.2).

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

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

This is similar to equation (1.3). And `sms` is calculated in the function `move` of the class parcel. (in src/lagrangian/dieselSpray/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 (1.12)

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 (1.12) 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 (1.12) with $U_p = U@P$ and $tauMomentum = \tau_p$

2.4 Equations ?? 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 in (1.9). 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/drag-Model/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 (**??**) and (1.9).

2.5 Equation (1.10)

2.5.1 icoLagrangianFoam

The drag coefficient is a constant given in the dictionary `/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 dictionary in `/constant/sprayProperties`

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

This is similar to equation (1.10)

2.6 Equation (1.13)

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 lambdaMin which is the fraction of the trajectory completed to reach the first face.

```
if (faces.empty())  // inside cell
 {
 trackFraction = 1.0;
position_ = endPosition;
}
else // hit face
 {
   scalar lambdaMin = GREAT;
   if (faces.size() == 1)   //the particle has to cross only one face
   {
     lambdaMin = lambda(position_, endPosition, faces[0], stepFraction_);
     facei_ = faces[0];
```

```
   }
   else     // If the particle has to cross more than one cell, find the first one
  {
    forAll(faces, i)
    {
     scalar lam = lambda(position_, endPosition, faces[i], stepFraction_);
     if (lam < lambdaMin)
     {  lambdaMin = lam;
         facei_ = faces[i];
     }
    }
 }
trackFraction = lambdaMin;
position_ += trackFraction*(endPosition - position_);
```

If the particle reaches the final position $endPosition = position + dt * U$ without crossing a cell, the function return 1 and $position$ is updated to $endPosition$. If the particle has to cross one or more cells, the fraction of trajectory lambdaMin for this lagrangian time step is evaluated and $position$ is updated at the face with $position = position + lambdaMin * (endPosition - position)$. This is similar to equation (1.13).

2.7 Equation (1.19)

- 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);
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;
}
```

This is similar to equation (1.19) exept that the coefficient of restitution is not a fixed value. It is the random scalar `gf`.

2.8 Equation (1.21)

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;
    }
```

This is similar to equation (1.21) .

Chapter 3

Update icoLagrangianFoam for OpenFOAM-1.6

The solver icoLagrangianFoam is at the time of writing this tutorial only avilable for OpenFOAM-1.5. This chapter describes how to update it to OpenFOAM-1.6. The 1.5 version can be downlowded with

```
svn checkout https://openfoam-extend.svn.sourceforge.net/svnroot/openfoam-extend/trunk/
Breeder_1.5/solvers/other/IcoLagrangianFoam/
```

See the OpenFOAM wiki page for more informations:

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

3.1 createParticles.H

Change from

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

to

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

3.2 HardBallParticle.H

After the already available hitPatch functions, 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

After the already available hitPatch functions, 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

Change in the function `evolve()` from

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

to

```
autoPtr<interpolation<vector> > UInt = interpolation<vector>::New
    (
        interpolationSchemes_,
        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 pisoFoam solver. pisoFoam is a transient solver for incompressible flow where turbulence modelling is generic, i.e. laminar, RAS or LES may be selected. We call the LPT solver based on pisoFoam pisoLagrangianFoam.

In the following the `sed` command has been written on several lines to make it easier to read the tutorial. But the command shall be written without breaking the line. It should be written on one line without space : `sed 's/a/b/g' file1>file2`

```
First, update icoLagrangianFoam to version 1.6.
cd $WM_PROJECT_USER_DIR/applications
cp -r $WM_PROJECT_DIR/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' pisoFoam.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 $thres$. $tStart, tEnd$ 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 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);
}
```

- In IncompressibleCloud.C, after `smoment_(mesh_.nCells(), vector::zero)`, add

```
nbPVp_(mesh_nCells(),0.0);
```

and in the function `evolve()`, after `smoment_ = vector::zero;`, add

```
nbPVp_=0 ;
```

- In IncompressibleCloud.H, in `private data`, add

```
scalarField nbPVp_;
```

and in public member functions add

```
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;
}
```

- 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 as dieselFoam. There is no injector. 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

The injector send $nbInjByDt$ particles at each time step between $tStart$ and $tEnd$, in a rectangle of $center$ and side $r0$ defined by the user in the dictionary (`constant/particleProperties`). The initial velocity of the particle is $U_p = vel + randomscalar * velprime$. The parameters for the velocity vel and $velprime$, the diameter d, the density $rhop$, the coeffitient of restitution e and friction mu are also defined by the user.

```
cd $WM_PROJECT_USER_DIR/applications
cp -r $WM_PROJECT_DIR/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, add

```
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`, add

```
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);
    // if 2D and cell center is z=0, all particles should be injected at z=0
    // and the position should be set to
    // vector pos=center+vector(posx,posy,0);
    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));
```

As presented in the first chapter, the threshold value of Re_p is 0.1. There is a misprint in the code. It is important to correct it in order to get a good estimation of the drag coefficient: change `if (Re > 0.01)` to `if (Re > 0.1)`.

6.3 Volume fraction of particles

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

```
if (td.keepParticle)
 {
  label cellnew = cell();
```

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

- In solidParticleCloud.C (after `random_(666),`) add

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

  and in the function `move` (before `solidParticle::trackData ...`) add

  ```
  nbPVp_ = 0.0;
  ```

- In solidParticleCloud.H, in `private data`, add

  ```
   scalarField nbPVp_;
  ```

  and in public member functions, add

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

- Add at the end of 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 volScalarField `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 use the converged solution (i.e. the solution at t=1000) as initial flow condition for LPT computations. The files used in this test case and the LPT solver are available at the homepage of the course.

- copy the tutorial pitzDaily
- run blockMesh
- copy the provided folder 1000 or run simpleFoam and add the file 1000/volFracorg (cp 1000/p 1000/volFracorg and change dimension to [0 0 0...] and `object p` to `object volFracorg`)
- add the provided file g in constant/

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

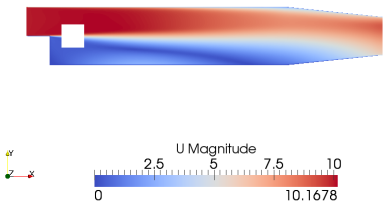
- add density in constant/transportProperties

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

- add the provided file particleProperties in constant/

```
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;
```

These entries means that 500 particles of diameter 5e-5m and velocity 0m/s are injected per time step from a flat box of center (0.02,0,0) and sides ([-0.01,0.01], [-0.01,0.01], 0.0) , as shown here:



Box of injection of particles, velocity at t=1000

- The lengh of the domain is 0.311m. The maximum flow 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, until t=1001. We start the LPT computation and the particles injection at t=1000, i.e. the flow already converged to a steady state solution. The LPT solver only tracks particles in the flow, it doesn't solve for flow velocity and pressure. The file system/controlDict is now

```
startFrom       startTime;
startTime       1000;
stopAt          endTime;
endTime         1000.3;
deltaT          0.003;
writeControl    timeStep;
writeInterval   1;
timePrecision   12;
```

  To be able to write the results in a folder with name 1000.003, the entry `timePrecision` was changed from the default value 6 to 12. (Otherwise the folder 1000 will be overwritten several times).

- run the solver mySolidParticleFoam provided

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

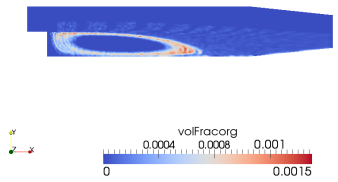


Figure 1: distribution of the volume fraction of particles at t=1000.3, dt=3e-3.

We see that the distribution of particles draw some kind of stripes. This is due to the injector that injects particles as a pulsation at every time step. Reducing the time step would produce a more continuous inflow of particles and a more homogeneous distribution of particles, as shown in figure 2 (dt= 3e-5 and nbInjByDt=5 in order to have the same final amount of particles as in the first case).

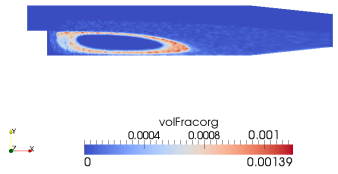


Figure 2: distribution of the volume fraction of particles at t=1000.3, dt=3e-5.