Coupling of VOF with LPT in OpenFOAM

Aurélia Vallier [1]

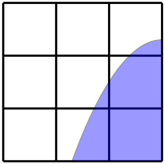
[1] Fluid Mechanics/Energy Sciences, LTH Lund University, Sweden

2011-09-12

1 Multiphase flow methods
- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

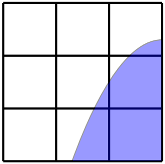
2 Tutorial
- Add a LPT solver to interFoam
- Add a particle injector
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

VOF (Volume Of Fluid)



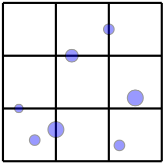
- Bubbles/droplets larger than the grid size
- Irregular structures: need to describe the interface

VOF (Volume Of Fluid)



- Bubbles/droplets larger than the grid size
- Irregular structures: need to describe the interface

LPT (Lagrangian Particle Tracking)



- Bubbles/droplets smaller than the grid size
- Shape can be considered spherical

1 Multiphase flow methods
- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

2 Tutorial
- Add a LPT solver to interFoam
- Add a particle injector
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha \rho_l + (1 - \alpha)\rho_g,$$

$$\mu = \alpha \mu_l + (1 - \alpha)\mu_g,$$

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha \rho_l + (1 - \alpha) \rho_g,$$

$$\mu = \alpha \mu_l + (1 - \alpha) \mu_g,$$

Transport equation for the liquid volume fraction

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{U}) + \nabla \cdot (\alpha (1 - \alpha) \mathbf{U_r}) = 0$$

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha\rho_l + (1-\alpha)\rho_g,$$

$$\mu = \alpha\mu_l + (1-\alpha)\mu_g,$$

Transport equation for the liquid volume fraction

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{U}) + \nabla \cdot (\alpha(1-\alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st}\kappa\delta\mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla\alpha}{|\nabla\alpha|}\ ,\ \kappa = \nabla\mathbf{n}.$$

ls $WM_PROJECT_DIR/applications/solvers/multiphase/interFoam

```
alphaEqn.H
alphaEqnSubCycle.H
correctPhi.H
createFields.H
interFoam.C
Make
pEqn.H
UEqn.H
```

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha \rho_g + (1 - \alpha)\rho_l,$$

$$\mu = \alpha \mu_g + (1 - \alpha)\mu_l,$$

Transport equation for the vapor volume fraction

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{U}) + \nabla \cdot (\alpha(1 - \alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st} \kappa \delta \mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla \alpha}{|\nabla \alpha|}\ ,\ \kappa = \nabla \mathbf{n}.$$

Liquid volume fraction $\alpha \in [0, 1]$.

vi createFields.H

```
Info<< "Reading field alpha1" << endl;
    volScalarField alpha1
    (
        IOobject
        (
            "alpha1",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
```

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha\rho_g + (1 - \alpha)\rho_l,$$

$$\mu = \alpha\mu_g + (1 - \alpha)\mu_l,$$

Transport equation for the vapor volume fraction

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{U}) + \nabla \cdot (\alpha(1 - \alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st}\kappa\delta\mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla\alpha}{|\nabla\alpha|}\ ,\ \kappa = \nabla\mathbf{n}.$$

$$\rho = \alpha\rho_g + (1-\alpha)\rho_l$$

vi createFields.H

```
Info<< "Reading transportProperties" << endl;
twoPhaseMixture twoPhaseProperties(U, phi, "alpha1");
const dimensionedScalar& rho1 = twoPhaseProperties.rho1();
const dimensionedScalar& rho2 = twoPhaseProperties.rho2();
 volScalarField rho
    (
        IOobject
        (
            "rho",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT
        ),
        alpha1*rho1 + (scalar(1) - alpha1)*rho2,
        alpha1.boundaryField().types()
    );
```

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha\rho_g + (1-\alpha)\rho_l,$$

$$\mu = \alpha\mu_g + (1-\alpha)\mu_l,$$

Transport equation for the vapor volume fraction

$$\frac{\partial\alpha}{\partial t} + \nabla\cdot(\alpha\mathbf{U}) + \nabla\cdot(\alpha(1-\alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\otimes\mathbf{U}) = -\nabla p + \mu\nabla^2\mathbf{U} + \rho\mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st}\kappa\delta\mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla\alpha}{|\nabla\alpha|}\ ,\ \kappa = \nabla\mathbf{n}.$$

$$\mu = \alpha\mu_g + (1-\alpha)\mu_l$$

vi $WM_PROJECT_DIR/src/transportModels/incompressible/incompressibleTwoPhaseMixture/twoPhaseMixture.C

```
Foam::tmp<Foam::volScalarField> Foam::twoPhaseMixture::mu() const
{
   const volScalarField limitedAlpha1
   (
       min(max(alpha1_, scalar(0)), scalar(1))
   );
   return tmp<volScalarField>
    (
       new volScalarField
        (
           "mu",
           limitedAlpha1*rho1_*nuModel1_->nu()
          + (scalar(1) - limitedAlpha1)*rho2_*nuModel2_->nu()
         )
     );
 }
```

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha\rho_g + (1-\alpha)\rho_l,$$

$$\mu = \alpha\mu_g + (1-\alpha)\mu_l,$$

Transport equation for the vapor volume fraction

$$\frac{\partial\alpha}{\partial t} + \nabla\cdot(\alpha\mathbf{U}) + \nabla\cdot(\alpha(1-\alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\otimes\mathbf{U}) = -\nabla p + \mu\nabla^2\mathbf{U} + \rho\mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st}\kappa\delta\mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla\alpha}{|\nabla\alpha|}\ ,\ \kappa = \nabla\mathbf{n}.$$

Transport equation for the vapor volume fraction

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{U}) + \nabla \cdot (\alpha(1-\alpha)\mathbf{U_r}) = 0$$

vi alphaEqn.H

Liquid volume fraction $\alpha \in [0, 1]$.

$$\rho = \alpha\rho_g + (1-\alpha)\rho_l,$$

$$\mu = \alpha\mu_g + (1-\alpha)\mu_l,$$

Transport equation for the vapor volume fraction

$$\frac{\partial\alpha}{\partial t} + \nabla\cdot(\alpha\mathbf{U}) + \nabla\cdot(\alpha(1-\alpha)\mathbf{U_r}) = 0$$

Mass and momentum equations for the mixture

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

$$\frac{\partial\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\otimes\mathbf{U}) = -\nabla p + \mu\nabla^2\mathbf{U} + \rho\mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st}\kappa\delta\mathbf{n}\ ,\ \mathbf{n} = \frac{\nabla\alpha}{|\nabla\alpha|}\ ,\ \kappa = \nabla\mathbf{n}$$

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_{st}} = \sigma_{st} \kappa \delta \mathbf{n} \ , \ \mathbf{n} = \frac{\nabla \alpha}{|\nabla \alpha|} \ , \ \kappa = \nabla \mathbf{n}.$$

vi UEqn.H

```
solve
   (
       UEqn
    ==
       fvc::reconstruct
       (
           (
               fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
             - ghf*fvc::snGrad(rho)
             - fvc::snGrad(pd)
           ) * mesh.magSf()
       )
   );
```

Now, what is interface.sigmaK() ?

- Go to the online documentation: http://foam.sourceforge.net/docs/cpp/
- Search for sigmaK and select the hint Foam::interfaceProperties::sigmaK()
- Select the link "line 140 of the file interfaceProperties.H."

```
00140        tmp<volScalarField> sigmaK() const
00141        {
00142            return sigma_*K_;
00143        }
00144
00145        void correct()
00146        {
00147            calculateK();
00148        }
```

- Go to the description of the function calculateK() in **interfaceProperties.C**

```
00123    // Face unit interface normal
00124    surfaceVectorField nHatfv(gradAlphaf/(mag(gradAlphaf) + deltaN_));
00126
00127    // Face unit interface normal flux
00128    nHatf_ = nHatfv & Sf;
00129
00130    // Simple expression for curvature
00131    K_ = -fvc::div(nHatf_);
```

- At line 164, you can also see that sigma is read in the dictionary "tranportProperties"

```
00164    sigma_(dict.lookup("sigma")),
```

1. Multiphase flow methods
   - VOF
   - VOF in OpenFOAM
   - LPT
   - LPT in OpenFOAM
2. Tutorial
   - Add a LPT solver to interFoam
   - Add a particle injector
   - Add two-way coupling
   - Add four-way coupling
     - A deterministic model for collision
     - Implementation in OpenFoam
     - The outcome of collision
     - Implementation in OpenFoam
   - Convert particle to VOF
     - Grid refinement
     - Set the volume fraction to 1 in the bottom region
     - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Two-way coupling:

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Two-way coupling:

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

Four-way coupling: particle-particle collisions.

1 Multiphase flow methods

- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

2 Tutorial

- Add a LPT solver to interFoam
- Add a particle injector
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

ls $WM_PROJECT_DIR/src/lagrangian/solidParticle/

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

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Two-way coupling:

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

Four-way coupling: particle collisions.

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

vi solidParticle.H

```
inline solidParticle
(
    const Cloud<solidParticle>& c,
    const vector& position,
    const label celli,
    const scalar m,
    const vector& U
);
```

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

vi solidParticle.H

```
inline solidParticle
(
    const Cloud<solidParticle>& c,
    const vector& position,
    const label celli,
    const scalar m,
    const vector& U
);
```

vi solidParticleI.H

```
inline Foam::solidParticle::solidParticle
(
    const Cloud<solidParticle>& c,
    const vector& position,
    const label celli,
    const scalar d,
    const vector& U
)
```

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

vi solidParticle.H

```
inline solidParticle
(
    const Cloud<solidParticle>& c,
    const vector& position,
    const label celli,
    const scalar m,
    const vector& U
);
```

vi solidParticleI.H

```
inline Foam::solidParticle::solidParticle
(
    const Cloud<solidParticle>& c,
    const vector& position,
    const label celli,
    const scalar d,
    const vector& U
)
```

vi solidParticleCloud.C

```
rhop_(dimensionedScalar(particleProperties_.lookup("rhop")).value()),
```

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Two-way coupling:

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

Four-way coupling: particle-particle collisions.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

vi solidParticle.C

```
dt *= trackToFace(position() + dt*U_, td);
tEnd -= dt;
stepFraction() = 1.0 - tEnd/deltaT;
...
scalar Dc = (24.0*nuc/d_)*ReFunc*(3.0/4.0)*(rhoc/(d_*rhop));
U_ = (U_ + dt*(Dc*Uc + (1.0 - rhoc/rhop)*td.g()))/(1.0 + dt*Dc);
```

Now, what is trackToFace ?

- Go to the online documentation:
  http://foam.sourceforge.net/docs/cpp/
- Search for trackToFace and select the hint Foam::particle

```
Foam::scalar trackToFace(const vector & endPosition,
                         TrackData & td
                         )
```

```
Track 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. on entry 'stepFraction()'
should be set to the fraction of the time-step at which the tracking starts.
Definition at line 202 of file particleTemplates.C.
```

Particle P: position $\mathbf{x_P}$, diameter D_P, velocity $\mathbf{U_P}$ and density ρ_P.

$$\frac{d\mathbf{x_P}}{dt} = \mathbf{U_P},$$

$$m_P \frac{d\mathbf{U_P}}{dt} = \sum \mathbf{F}.$$

Two-way coupling:

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

Four-way coupling: particle-particle collisions.

NOT IMPLEMENTED

1 Multiphase flow methods
- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

2 Tutorial
- Add a LPT solver to interFoam
- Add a particle injector
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

1. Add a LPT solver to track small particles in the multiphase solver interFoam
2. Add a particle injector
3. Add two-way coupling (source term S_P in the momentum equation of interFoam)
4. Add four-way coupling (add a model for particle-particle collision)
5. Convert particle to VOF when it comes close to a VOF interface

```
cd $WM_PROJECT_USER_DIR/applications
mkdir myLPTVOF
cd myLPTVOF

cp -r $WM_PROJECT_DIR/applications/solvers/multiphase/interFoam/* .

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

vi Make/files

```
interFoam.C
solidParticle.C
solidParticleIO.C
solidParticleCloud.C
EXE = $(FOAM_USER_APPBIN)/myLPTVOF
```

vi Make/options

```
-I$(LIB_SRC)/lagrangian/basic/lnInclude\
-I$(LIB_SRC)/finiteVolume/lnInclude

-lfiniteVolume \
-llagrangian \
-lsolidParticle \
```

vi interFoam.C

```
:47
#include "solidParticleCloud.H"

:63
solidParticleCloud particles(mesh);

:106
particles.move(g);
Info<< "Cloud size= "<< particles.size() <<endl;
runTime.write();
```

Compile the solver

wmake

1 Multiphase flow methods
- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

2 Tutorial
- Add a LPT solver to interFoam
- **Add a particle injector**
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

Goal: Inject 2 particles (P1 and P2).
We'll give in a dictionary (particleProperties):

- position (posP1 and posP2)
- diameter (Dp1 and Dp2)
- velocity (Up1 and Up2)
- time when injection starts and ends.

And the constructor needs to know

- position
- cell
- diameter (and not mass!)
- velocity

solidParticle.H

```
//- Construct from components
 inline solidParticle
  (
   const Cloud<solidParticle>& c,
   const vector& position,
   const label celli,
   const scalar m,
   const vector& U
  );
```

vi solidParticleCloud.C

```
mu_(dimensionedScalar(particleProperties_.lookup("mu")).value()),
posP1_(dimensionedVector(particleProperties_.lookup("posP1")).value()),
dP1_(dimensionedScalar(particleProperties_.lookup("dP1")).value()),
UP1_(dimensionedVector(particleProperties_.lookup("UP1")).value()),
posP2_(dimensionedVector(particleProperties_.lookup("posP2")).value()),
dP2_(dimensionedScalar(particleProperties_.lookup("dP2")).value()),
UP2_(dimensionedVector(particleProperties_.lookup("UP2")).value()),
tInjStart_(dimensionedScalar(particleProperties_.lookup("tInjStart")).value()),
tInjEnd_(dimensionedScalar(particleProperties_.lookup("tInjEnd")).value())
```

vi solidParticleCloud.H

```
scalar mu_;
vector posP1_;
scalar dP1_;
vector UP1_;
vector posP2_;
scalar dP2_;
vector UP2_;
scalar tInjStart_;
scalar tInjEnd_;
```

vi solidParticleCloud.C

```
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //
 void Foam::solidParticleCloud::inject(solidParticle::trackData &td)
{
  label cellI=mesh_.findCell(td.spc().posP1_);
  solidParticle* ptr1= new solidParticle(*this,td.spc().posP1_,cellI,
   td.spc().dP1_,td.spc().UP1_);
  Cloud<solidParticle>::addParticle(ptr1);

  cellI=mesh_.findCell(td.spc().posP2_);
  solidParticle* ptr2= new solidParticle(*this,td.spc().posP2_,cellI,
   td.spc().dP2_,td.spc().UP2_);
  Cloud<solidParticle>::addParticle(ptr2);
}
```

In the function move

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

vi solidParticleCloud.H

```
void move(const dimensionedVector& g);
//- Inject particles according to the dictionnary particleProperties
void inject(solidParticle::trackData &td);
```

- Compile the solver :

  wmake

- Go to the test case boxLPT directory :

  cd $WM_PROJECT_USER_DIR/run/boxLPT

- Have a look at the injector dictionary

  vi constant/particleProperties

```
rhop rhop [ 1 -3  0  0  0  0  0] 1000;
e    e    [ 0  0  0  0  0  0  0] 0.2;
mu   mu   [ 0  0  0  0  0  0  0] 0.02;
posP1 posP1 [ 0  1  0  0  0  0  0] (0.005 0.0 0.0125);
dP1 dP1 [ 0  1  0  0  0  0  0] 0.0002;
UP1 UP1 [ 0  1 -1   0  0  0  0] (-7.071 0 -7.071);
posP2 posP2 [ 0  1  0  0  0  0  0] (-0.0054 0.0 0.0125);
dP2 dP2 [ 0  1  0  0  0  0  0] 0.0002;
UP2 UP2 [ 0  1 -1   0  0  0  0] (7.071 0 -7.071);
tInjStart tInjStart [ 1 0   0  0  0  0  0] 0;
tInjEnd tInjEnd [ 1 0   0  0  0  0  0] 0.0000252;
```

- Run the solver

  myLPTVOF > log&

- Have a look at the log file to check if particles were injected.

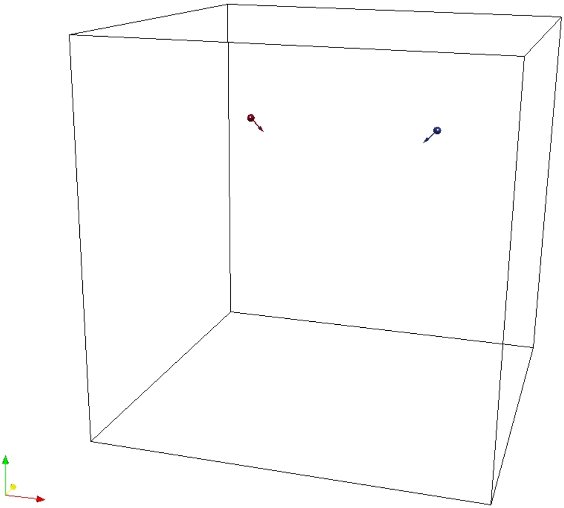
- Load the results in paraview

  touch boxLPT.foam
  paraview
  File Open boxLPT.foam

- Visualize the particles:

  Mesh regions: lagrangian/DefaultCloud
  Glyph: Type Sphere, Mode Scalar, Edit Set Scalar Factor 1

Add a particle injector



1. Multiphase flow methods
   - VOF
   - VOF in OpenFOAM
   - LPT
   - LPT in OpenFOAM
2. Tutorial
   - Add a LPT solver to interFoam
   - Add a particle injector
   - **Add two-way coupling**
   - Add four-way coupling
     - A deterministic model for collision
     - Implementation in OpenFoam
     - The outcome of collision
     - Implementation in OpenFoam
   - Convert particle to VOF
     - Grid refinement
     - Set the volume fraction to 1 in the bottom region
     - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

Two-way coupling: We want to account for the influence of the particles on the continuous phase, i.e. we add the source term S_p in the momentum equations for the mixture:

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{g} - \mathbf{S_{st}} + \mathbf{S_P}.$$

$$\mathbf{S_P} = \frac{1}{V_{cell} \Delta t} \sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t}\sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

vi solidParticle.C

```
scalar m=rhop*4/3*mathematicalConstant::pi*pow(d_/2,3);
vector oldMom=U_*m;
U_ = (U_ + dt*(Dc*Uc + (1.0 - rhoc/rhop)*td.g()))/(1.0 + dt*Dc);
vector newMom=U_*m;
td.spc().smom()[celli] += newMom-oldMom;
```

vi solidParticleCloud.C

```
mu_(dimensionedScalar(particleProperties_.lookup("mu")).value()),
smom_(mesh_.nCells(), vector::zero)

smom_=vector::zero;
solidParticle::trackData td(*this, rhoInterp, UInterp, nuInterp, g.value());
```

vi solidParticleCloud.H

```
// Private data
       ...
  scalar mu_;
  vectorField smom_;

inline scalar mu() const;
inline vectorField& smom();
inline const vectorField& smom() const;
inline tmp<volVectorField> momentumSource() const;
```

$$\mathbf{S_P} = \frac{1}{V_{cell}\Delta t}\sum_P m_P((\mathbf{U_p})_{t_{out}} - (\mathbf{U_p})_{t_{in}})$$

vi solidParticleCloudI.H

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

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

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

start the file with

```
 namespace Foam
{
```

end the file with

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

Add the source term in **UEqn.H**

```
if (momentumPredictor)

        solve
        (
            UEqn
         ==
            fvc::reconstruct
            (
                fvc::interpolate(rho)*(g & vofMesh.Sf())
              + (
                    fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
                  - fvc::snGrad(p_rgh)
                ) * vofMesh.magSf()
            )
            +particles.momentumSource()
        );
```

- Compile the solver :

  wmake

- Go to the test case boxLPT directory :

  cd $WM_PROJECT_USER_DIR/run/boxLPT

- Run the solver

  myLPTVOF > log&

- Load the results in paraview

  paraview

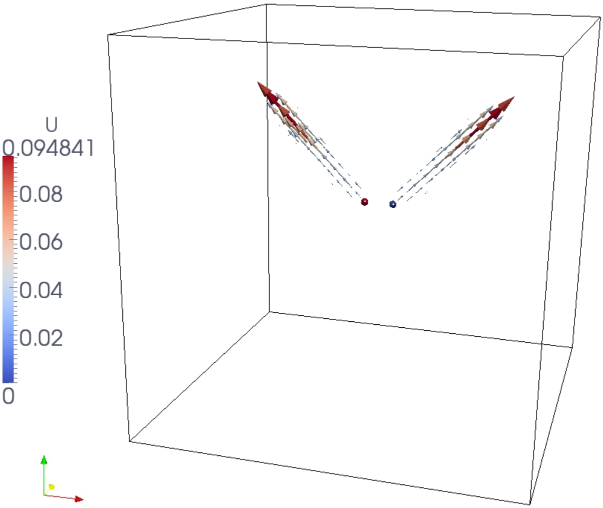
- Visualize the particles:

  Mesh regions: lagrangian/DefaultCloud
  Glyph: Type Sphere, Mode Scalar, Edit Set Scalar Factor 1

- Plot the vectors of the continuous phase in the y-plane

  Mesh regions: internalMesh
  Slice: Y Normal
  Glyph: Type Arrow, Mode Vector, Edit Set Scalar Factor 0.03, No Mask Points, No Random Mode

Add two-way coupling



Four-way coupling: particle-particle collision.

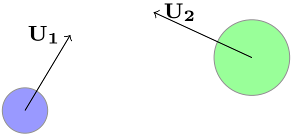
In OpenFoam, there are 2 models available for modeling particle collision:

vi $FOAM_SRC/lagrangian/dieselSpray/spraySubModels/collisionModel

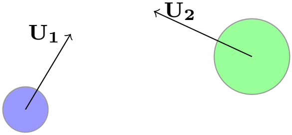
- O'rourke model: collision occurs with a certain probability if 2 particles are in the same cell.
- Trajectory model: collision occurs with a ceratin probability if 2 particles are close enough and if they move toward each other.

We want a deterministic model. So we derive one and implement it.

We want to determine when collision occurs between particles P_1 and P_2.

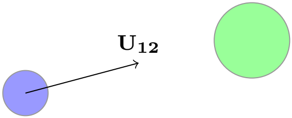


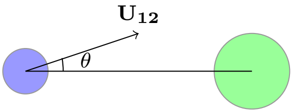
We want to determine when collision occurs between particles P_1 and P_2.



Let particle 2 be the new coordinate reference system,

- Particle 2 has a velocity $\mathbf{U_{22}} = 0$
- Particle 1 has a velocity $\mathbf{U_{12}} = \mathbf{U_1} - \mathbf{U_2}$



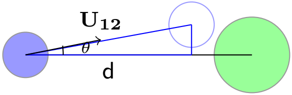


The angle θ is the angle between line (1,2) and $\mathbf{U_{12}}$:

$$cos\theta = \frac{\mathbf{U_{12}}}{|\mathbf{U_{12}}|} \cdot \frac{\mathbf{posP_2} - \mathbf{posP_1}}{|\mathbf{posP_2} - \mathbf{posP_1}|}$$

There exists a critical value θ_c such that if $\theta < \theta_c$ then collision is possible.

Let's consider the case $\theta < \theta_c$.



The distance d is the projection on line (1,2) of the distance travelled during the time step dt :

$$d = |\mathbf{U_{12}}| cos\theta dt$$

Then the particle 1 should travel long enough during this time step. So there exists also a critical distance d_c such that if $d > d_c$ then collision occurs.

We create the file **myCM.H**

```
//evaluate thetac
vector p = p2().position() - p1().position();
scalar dist = mag(p);
scalar sumR = (p1().d() + p2().d())/2.0;
scalar thetac = atan(sumR/ sqrt(sqr(dist)+sqr(sumR)));
// evaluate theta
vector v1 = p1().U();
vector v2 = p2().U();
vector vRel = v1 - v2;
scalar magVRel = mag(vRel);
scalar theta = acos( (vRel/(magVRel+SMALL)) & (p/(dist+SMALL)) );
if ( theta <thetac)
{
 Info<< "theta smaller than thetac"<<endl;
 // evaluate dcoll
 scalar dt = mesh_.time().deltaT().value();
 scalar vcoll = vRel & (p/(dist+SMALL));
 scalar dcoll = vcoll * dt;
 // evaluate dcollc
 scalar denom=1.0+sqr(tan(theta));
 scalar dcollc = 2* dist -2* sqrt(sqr(dist) - denom*(sqr(dist)-sqr(sumR)));
 dcollc /= (2*denom);
 if (dcoll  > dcollc)
 {
  collision=true;
  Info<<"collision occurs"<<endl;
 } // if - collision distance
} // if - collision angle
```

vi interFoam.C

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

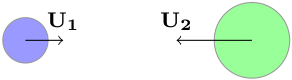
vi solidparticleCloud.H

```
void move(const dimensionedVector& g);
//- Check if there will be collision and update velocities
void checkCo();
```

vi solidparticleCloud.C

```
 void Foam::solidParticleCloud::checkCo()
{
   if ((*this).size() < 2)
    {
       return;
    }
    Cloud<solidParticle>::iterator secondP = (*this).begin();
    ++secondP;
    Cloud<solidParticle>::iterator p1 = secondP;
    while (p1 != (*this).end())
    {
      Cloud<solidParticle>::iterator p2 = (*this).begin();
      while (p2 != p1)
        {
          bool collision=false;
          #            include "myCM.H"
      if (collision)
          {
           p1.U() = ??;
           p2.U() = ??;
           Info<< "new velocities "<< p1().U() << p2().U()<<endl;
         }
         ++p2;
        } // end - inner loop
      ++p1;
    } // end - outer loop
}
```

1D collision



Conservation of momentum

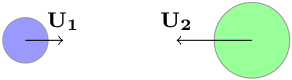
$$m_1\mathbf{U_1} + m_2\mathbf{U_2} = m_1\mathbf{U_1'} + m_2\mathbf{U_2'} \tag{1}$$

Coefficient of restitution (loss of kinetic energy) ϵ

$$\epsilon = \frac{\mathbf{U_2'} - \mathbf{U_1'}}{\mathbf{U_1} - \mathbf{U_2}} \tag{2}$$

Eq(1): $\mathbf{U_1'} = \frac{m_1\mathbf{U_1} + m_2\mathbf{U_2} - m_2\mathbf{U_2'}}{m_1}$
Eq(2): $\mathbf{U_2'} = \epsilon(\mathbf{U_1} - \mathbf{U_2}) + \mathbf{U_1'}$

1D collision



Conservation of momentum

$$m_1\mathbf{U_1} + m_2\mathbf{U_2} = m_1\mathbf{U_1'} + m_2\mathbf{U_2'} \tag{1}$$

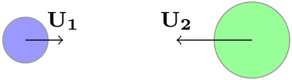
Coefficient of restitution (loss of kinetic energy) ϵ

$$\epsilon = \frac{\mathbf{U_2'} - \mathbf{U_1'}}{\mathbf{U_1} - \mathbf{U_2}} \tag{2}$$

Eq(1): $\mathbf{U_1'} = \frac{m_1\mathbf{U_1}+m_2\mathbf{U_2}-m_2\mathbf{U_2'}}{m_1}$
Eq(2): $\mathbf{U_2'} = \epsilon(\mathbf{U_1} - \mathbf{U_2}) + \mathbf{U_1'}$

$$\mathbf{U_1'} = \frac{m_1\mathbf{U_1} + m_2\mathbf{U_2} - m_2\epsilon(\mathbf{U_1} - \mathbf{U_2})}{m_1 + m_2}$$

1D collision



Conservation of momentum

$$m_1\mathbf{U_1} + m_2\mathbf{U_2} = m_1\mathbf{U_1'} + m_2\mathbf{U_2'} \tag{1}$$

Coefficient of restitution (loss of kinetic energy) ϵ

$$\epsilon = \frac{\mathbf{U_2'} - \mathbf{U_1'}}{\mathbf{U_1} - \mathbf{U_2}} \tag{2}$$

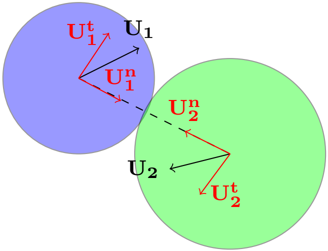
Eq(1): $\mathbf{U_1'} = \frac{m_1\mathbf{U_1} + m_2\mathbf{U_2} - m_2\mathbf{U_2'}}{m_1}$
Eq(2): $\mathbf{U_2'} = \epsilon(\mathbf{U_1} - \mathbf{U_2}) + \mathbf{U_1'}$

$$\mathbf{U_1'} = \frac{m_1\mathbf{U_1} + m_2\mathbf{U_2} - m_2\epsilon(\mathbf{U_1} - \mathbf{U_2})}{m_1 + m_2}$$

Similarly,

$$\mathbf{U_2'} = \frac{m_1\mathbf{U_1} + m_2\mathbf{U_2} - m_1\epsilon(\mathbf{U_2} - \mathbf{U_1})}{m_1 + m_2}$$

2D collision



The velocities are decomposed into normal and tangential component, $\mathbf{U_i} = \mathbf{U_i^n} + \mathbf{U_i^t}$
The unit normal vector is $\mathbf{n_{12}} = \frac{\mathbf{x_2} - \mathbf{x_1}}{|\mathbf{x_2} - \mathbf{x_1}|}$ so $\mathbf{U_i^n} = (\mathbf{U_i} \& \mathbf{n_{12}}) \cdot \mathbf{n_{12}}$
The normal component changes according to the expression derived for 1D collision.

The tangential component is unchanged (friction neglected).

In **solidparticleCloud.C**

```
p1.U() = ??;
p2.U() = ??;
```

become

```
Info<< "Velocities before collision: "<< p1().U() <<p2().U()<<endl;
vector p = p2().position() - p1().position();
vector n12=p/mag(p);
scalar v1n=p1().U() & n12;
vector v1t=p1().U() - v1n*n12 ;
scalar v2n=p2().U() & n12;
vector v2t=p2().U() - v2n*n12 ;
scalar COR=0.8;
scalar m1 = rhop()*mathematicalConstant::pi / 6.0*pow(p1().d(),3);
scalar m2 = rhop()*mathematicalConstant::pi / 6.0*pow(p2().d(),3);
scalar mrn = m1*v1n + m2*v2n;
scalar vnRel = v1n - v2n;
 p1().U() = v1t+ ((mrn - COR* m2*vnRel)/(m1+m2))*n12;
 p2().U() = v2t+ ((mrn + COR* m1*vnRel)/(m1+m2))*n12;
Info<< "new velocities "<< p1().U() << " and "<<p2().U()<<endl;
```

Compile the solver :

wmake

```
error: passing ‘const Foam::vector’ as ‘this’ argument of
‘Foam::Vector<double>& Foam::Vector<double>::operator=
(const Foam::Vector<double>&)’ discards qualifiers
```

Debug 1

```
p1().U() == ...
p2().U() == ...
```

Compile the solver :

wmake

It gives no error message.
Run the solver with the case boxLPT :

cd $WM_PROJECT_USER_DIR/run/boxLPT
myLPTVOF > log&

Have a look at the velocities after collision in the log file ...

Debug 2

U is a member of the class solidParticle, which has a **const** qualification, and therefore cannot be modified within solidParticleCloud!

We want to change the value of U after a collision .

So we use the C++ process **const_cast** which removes the const qualification of an object. It is called "casting away constness".

```
solidParticle& changep1= const_cast<solidParticle&>(p1());
solidParticle& changep2= const_cast<solidParticle&>(p2());
changep1.U_ = ...
changep2.U_ = ...
```

Compile the solver :

wmake

```
error: 'Foam::vector Foam::solidParticle::U_' is private
```

Debug 3

vi solidParticle.H

```
class solidParticle
:
    public Particle<solidParticle>
{
    // Private member data
        //- Diameter
        scalar d_;
        //- Velocity of particle
        vector U_;
public:
    friend class Cloud<solidParticle>;
    friend class solidParticleCloud;
```

Compile the solver :

wmake

It gives no error message.
Run the solver with the case boxLPT

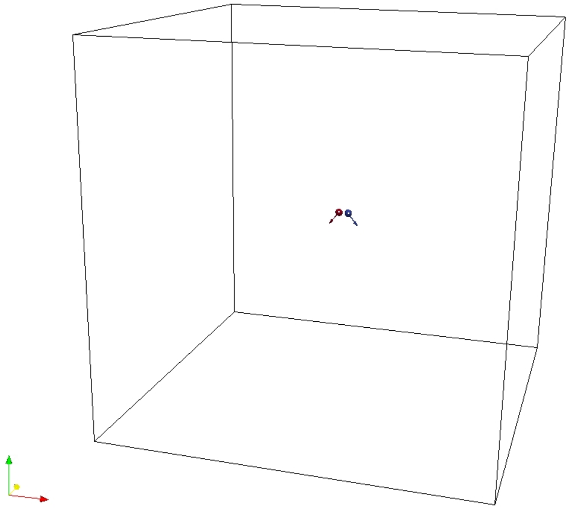
- **cd $WM_PROJECT_USER_DIR/run/boxLPT**
- Run the solver

  myLPTVOF >log&

- Have a look at the velocities after collision in the log file
- Visualize the particles:

  Mesh regions: lagrangian/DefaultCloud Glyph: Type Sphere, Mode Scalar, Edit Set Scalar Factor 1 Colored by origId

Add four-way coupling



1 Multiphase flow methods
- VOF
- VOF in OpenFOAM
- LPT
- LPT in OpenFOAM

2 Tutorial
- Add a LPT solver to interFoam
- Add a particle injector
- Add two-way coupling
- Add four-way coupling
  - A deterministic model for collision
  - Implementation in OpenFoam
  - The outcome of collision
  - Implementation in OpenFoam
- Convert particle to VOF
  - Grid refinement
  - Set the volume fraction to 1 in the bottom region
  - Implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

Last task for today... We'll add water in the bottom of the box. We want to switch to VOF approach when the droplets are in contact with the free surface.

- We will have a coarse mesh when we track the droplet with LPT (the cell must be larger than the particle because the particle is modeled as a point source)
- and a fine mesh to describe the free surface and underneath (because it is a requirement for accuracy with the VOF method).

So we need to

- refine the grid in a region of the domain (refineMesh -dict)
- set alpha to 1 in this region (setField)
- implement a algorithm which switch from LPT to VOF when the particle is close enough to a VOF interface

```
cd $WM_PROJECT_USER_DIR/run/boxLPT
cp $WM_PROJECT_DIR/applications/utilities/mesh/manipulation/
      refineMesh/refineMeshDict system/

cp $WM_PROJECT_DIR/applications/utilities/mesh/manipulation/
      cellSet/cellSetDict system/
```

Define a set of cell to be refined: in system/cellSetDict, topoSetSource, keep only

```
boxToCell
   {
      box   (-0.01 -0.01 0) (0.01 0.01 0.005);
   }
```

Create the set of cells c0

cellSet

Refine the cells listed in the set c0

refineMesh -dict

The refined mesh is written in the time directory 2.5e-05. We have to move it:

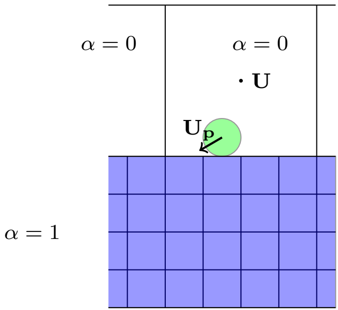
```
mv constant/polyMesh constant/polyMesh_br
mv 2.5e-05/polyMesh constant/
rm -r 2.5e-05/
```

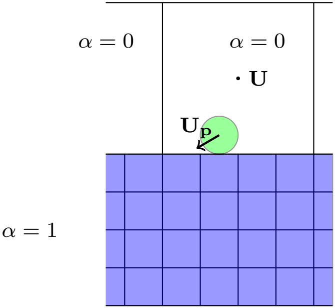
```
cp $WM_PROJECT_DIR/applications/utilities/preProcessing/
    setFields/setFieldsDict
```

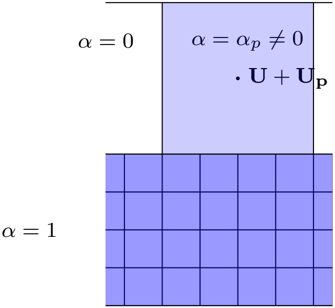
vi setFieldsDict

```
defaultFieldValues
(
    volScalarFieldValue alpha1 0
);
regions
(
    boxToCell
    {
        box (-0.01 -0.01 0) (0.01 0.01 0.005);
        fieldValues
        (
            volScalarFieldValue alpha1 1
        );
    }
);
```

setFields


$\alpha = 0$
$\alpha = 0$
$\cdot\, \mathbf{U}$
$\mathbf{U_P}$
$\alpha = 1$


$\alpha = 0$
$\alpha = 0$
$\mathbf{U}$
$\mathbf{U_p}$
$\alpha = 1$


$\alpha = 0$
$\alpha = \alpha_p \neq 0$
$\mathbf{U} + \mathbf{U_p}$
$\alpha = 1$

The algorithm is implemented in the file **LPTtoVOF.H**.
This file must be called in **solidParticle.C**, in the function move :

```
     #include "LPTtoVOF.H"
    }
    return td.keepParticle;
}
```

- We want to change the value of alpha1 and U in some cells, so we create a function addToAlpha and addToU, almost as we did for adding a source term in the momentum equation.
- We need access to the cell value of the eulerian volScalarField alpha1, ie alpha1 interpolated at the cell center

vi solidParticle.H

```
#include "interpolationCellPoint.H"
 #include "interpolationCell.H"

class trackData
    {
        //- Reference to the cloud containing this particle
        solidParticleCloud& spc_;
        // Interpolators for continuous phase fields
            const interpolationCellPoint<scalar>& rhoInterp_;
            const interpolationCellPoint<vector>& UInterp_;
            const interpolationCellPoint<scalar>& nuInterp_;
            const interpolationCell<scalar>&  alpha1Interp_;
        // Constructors
            inline trackData
            (
                solidParticleCloud& spc,
                const interpolationCellPoint<scalar>& rhoInterp,
                const interpolationCellPoint<vector>& UInterp,
                const interpolationCellPoint<scalar>& nuInterp,
                const interpolationCell<scalar>& alpha1Interp,
                const vector& g
            );
        // Member function
            inline solidParticleCloud& spc();
            inline const interpolationCellPoint<scalar>& rhoInterp() const;
            inline const interpolationCellPoint<vector>& UInterp() const;
            inline const interpolationCellPoint<scalar>& nuInterp() const;
            inline const interpolationCell<scalar>& alpha1Interp() const;
```

vi solidParticleI.H

```
inline Foam::solidParticle::trackData::trackData
(
    solidParticleCloud& spc,
    const interpolationCellPoint<scalar>& rhoInterp,
    const interpolationCellPoint<vector>& UInterp,
    const interpolationCellPoint<scalar>& nuInterp,
    const interpolationCell<scalar>& alpha1Interp,
    const vector& g
)
:
    spc_(spc),
    rhoInterp_(rhoInterp),
    UInterp_(UInterp),
    nuInterp_(nuInterp),
    alpha1Interp_(alpha1Interp),
    g_(g)

inline const Foam::interpolationCellPoint<Foam::scalar>&
Foam::solidParticle::trackData::nuInterp() const

    return nuInterp_;

inline const Foam::interpolationCell<Foam::scalar>&
Foam::solidParticle::trackData::alpha1Interp() const
{
    return alpha1Interp_;
}
```

vi solidParticleCloud.C

```
smom_(mesh_.nCells(), vector::zero),
correctalpha1_(mesh_.nCells(), 0),
correctU_(mesh_.nCells(), vector::zero),

const volScalarField& nu = vofMesh_.lookupObject<const volScalarField>("nu");
const volScalarField& alpha1 = mesh_.lookupObject<const volScalarField>("alpha1");

interpolationCellPoint<scalar> nuInterp(nu);
interpolationCell<scalar> alpha1Interp(alpha1);

smom_=vector::zero;
correctalpha1_=0;
correctU_=vector::zero;
solidParticle::trackData td(*this, rhoInterp, UInterp, nuInterp,alpha1Interp,  g.value());
```

vi solidParticleCloud.H

```
vectorField smom_;
scalarField correctalpha1_;
vectorField correctU_

inline tmp<volVectorField> momentumSource() const;
inline scalarField& correctalpha1();
inline const scalarField& correctalpha1() const;
inline tmp<volScalarField> AddTOAlpha() const;
inline vectorField& correctU();
inline const vectorField& correctU() const;
inline tmp<volVectorField> AddToU() const;
```

vi solidParticleCloudI.H

```
inline tmp<volScalarField> solidParticleCloud::AddTOAlpha() const
{
    tmp<volScalarField> alphasource
    (
        new volScalarField
        (
            IOobject
            (
                "correctalpha1",
                mesh_.time().timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh_,
            dimensionedScalar
            (
                "zero",
                dimensionSet(0,0 , 0, 0, 0),
                0
            )
        )
    );
    alphasource().internalField() = correctalpha1_;
    return alphasource;
}
```

vi solidParticleCloudI.H

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

```
inline Foam::scalarField& Foam::solidParticleCloud::correctalpha1()
 {
     return correctalpha1_;
 }

inline const Foam::scalarField& Foam::solidParticleCloud::correctalpha1() const
 {
     return correctalpha1_;
 }

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

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

vi interFoam.C

```
particles.move(g);
particles.checkCo();
alpha1 += particles.AddTOAlpha();
U += particles.AddToU();
runTime.write();
```

- Compile the solver :

  wmake

- Go to the test case boxLPT directory :

  cd $WM_PROJECT_USER_DIR/run/boxLPT

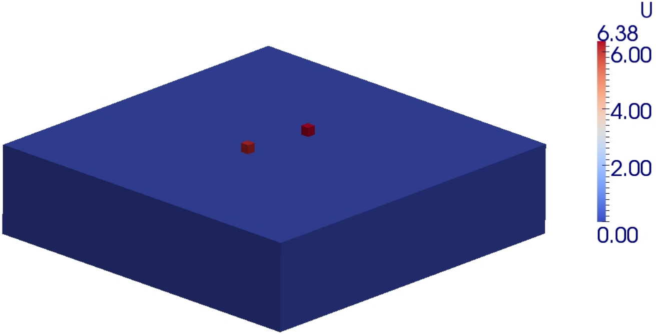
- Run the solver

  myLPTVOF > log&

- Load the results in paraview

  paraview

- Visualize the particles and then the cells with $\alpha > 0.03$ at t=0.001075.


U
6.38
6.00
4.00
2.00
0.00