Chalmers University of Technology

CFD with OpenSource software, Project

Droplet collisions in dieselSpray and implementations of collisions in solidParticle

Developed for OpenFOAM-1.5.dev

Author:
Josef Runsten

Peer reviewed by:
Anne Koesters
Jelena Andric

November 3, 2010

Preface

This tutorial is part of the examination of the MSc/PhD course in CFD with OpenSource software, 2010, http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/, at Chalmers University of Technology, Göteborg. The project described in this document is developed and tested in OpenFOAM-1.5-dev and describes a possible way to implement a collision model of solid particles. A thank you to examinator Håkan Nilsson and assistant Jelena Andric for their patience and support during the project.

Contents

1 Droplet collisions in dieselSpray **3**

1.1 Introduction . . . 3

1.2 Collisions in dieselSpray . . . 3

1.2.1 Selecting collision model . . . 3

1.2.2 The O'Rourke collision model . . . 3

1.2.3 The trajectory collision model . . . 6

2 Tutorial: Particle collisions **9**

2.1 Collisions in solidParticle . . . 9

2.1.1 The new solid particle collision model . . . 9

2.1.2 Creating collidingSolidParticleFoam . . . 10

2.1.3 Detecting collisions . . . 11

2.1.4 Velocity change upon collision . . . 12

2.1.5 Creating the case . . . 13

2.1.6 Running the case . . . 14

2.1.7 Issues and recommendations . . . 15

Chapter 1

Droplet collisions in dieselSpray

1.1 Introduction

In this chapter the modeling of collisions in the class dieselSpray is described. Today there exist no way to model collisions between solid particles in OpenFOAM-1.5.dev. The aim of the project is to implement such a model in the solidParticle class and to model a collision of two solid particles of equal density and diameter. This could be used as a basic work to create a more advanced collision model.

1.2 Collisions in dieselSpray

In order to implement collision models for solid particles it can be useful to understand how they are used for droplets. Hence, we will have a look at the spray submodels in the class dieselSpray.

1.2.1 Selecting collision model

A case using the dieselSpray class is aachenBomb, located in
`$FOAM_TUTORIALS/dieselFoam/aachenBomb`. In the dictionary `constant/sprayProperties` the collision model is set to `off` as default. This can be changed to either `ORourke` or `trajectory`, which are two different collision models, found in
`/$FOAM_SRC/src/lagrangian/dieselSpray/spraySubModels/collisionModel`.

1.2.2 The O'Rourke collision model

In the O'Rourke collision model, if a calculated collision probability is high enough and two particles are in the same cell they will collide regardless of their direction. The probability of a collision is the same for parcels heading towards each other as for parcels with opposite directions.. The collision frequency increases with smaller cell size, see eq (1.1), but there is no condition saying that two parcels has be close enough that they actually can meet within the time step.

$$\nu = N_{min}\frac{\pi}{4V_{cell}}(D_{min} + D_{max})^2|U_{rel}|\Delta t \tag{1.1}$$

where N_{min} is the number of droplets in the smaller parcel, V_{cell} is the volume of the cell, D is the diameter, U_{rel} is the relative velocity between the droplets and Δt is the time step.
To use this model, make the following changes in `/constant/sprayProperties`

```
collisionModel  ORourke;//off;
ORourkeCoeffs
{
    coalescence     off;
}
```

Switching `coalescence` on or off determines if droplets will coalesce after collision or not. If we would like to use this model for solid particles, which can not coalesce, this option would have to be disabled. Let's have a look at the files associated with the O'Rourke model, located in:
`/$FOAM_SRC/src/lagrangian/dieselSpray/spraySubModels/collisionModel/ORourke/`

The O'Rourke collision model class inherits attributes from `collisionModel.C`, which is an abstract class. We can see this in the beginning of `ORourkeCollisionModel.H`

```
class ORourkeCollisionModel
:
public collisionModel
{
```

In `ORourkeCollisionModel.C`, the member function `ORourkeCollisionModel::collideParcels` is used to check if two parcels (a parcel represents a group of droplets with the same physical properties and can be traced as one single droplet.This is used in order to avoid calculating for all droplets, which can be very costly in time) collide. First the size of the spray is checked. If the number of particles is less than 2 there can be no collision.

```
if (spray_.size() < 2)
    {
        return;
    }
```

An iterator is defined that goes through all the parcels in the domain, checks in which cells they are in and if two parcels in an iteration are in the same cell.

```
spray::iterator secondParcel = spray_.begin();
    ++secondParcel;
    spray::iterator p1 = secondParcel;

    while (p1 != spray_.end())
    {
        label cell1 = p1().cell();

        spray::iterator p2 = spray_.begin();

        while (p2 != p1)
        {
            label cell2 = p2().cell();

            // No collision if parcels are not in the same cell
```

If that is the case, the file `sameCell.H` is included, providing the collision behavior of the droplets in the two parcels.

```
            if (cell1 == cell2)
            {
#               include "sameCell.H"
            } // if - parcels in same cell
```

The rest of the code in this file deals with removal of the coalesced parcels. The smaller of the two coalesced parcels will be deleted. In `sameCell.H` the destiny of the parcels in the same cell is decided, based on probabilities depending on cell volume, relative velocity between parcels , size and number of droplets in each parcel. The implementation is the same as described in the KIVA manual [1]. Commented code describing the collision model is shown below.

```
vector v1 = p1().U();                   //Velocity of particle 1
vector v2 = p2().U();                   //Velocity of particle 2

vector vRel = v1 - v2;                  //Relative velocity
scalar magVRel = mag(vRel);             //Magnitude of the relative velocity

scalar sumD = p1().d() + p2().d();      //Sum of particle diameters
scalar pc = spray_.p()[p1().cell()];    //Pressure in the cell

//Decided which parcel is smaller
spray::iterator pMin = p1;
spray::iterator pMax = p2;

scalar dMin = pMin().d();
scalar dMax = pMax().d();

if (dMin > dMax) {
    dMin = pMax().d();
    dMax = pMin().d();
    pMin = p2;
    pMax = p1;
}

//Density of the big and small parcel
scalar rhoMax = spray_.fuels().rho(pc, pMax().T(), pMax().X());
scalar rhoMin = spray_.fuels().rho(pc, pMin().T(), pMin().X());

scalar mMax = pMax().m();               //Mass of the big parcel
scalar mMin = pMin().m();               //Mass of the small parcel
scalar mTot = mMax + mMin;              //Sum of the two parcel masses

scalar nMax = pMax().N(rhoMax);         //Number of droplets in the big parcel
scalar nMin = pMin().N(rhoMin);         //Number of droplets in the small parcel

scalar mdMin = mMin/nMin;               //Mean mass of droplets in the small parcel

//Calculating the probablity of collisions, according to the KIVA manual
scalar nu0 = 0.25*mathematicalConstant::pi*sumD*sumD*magVRel*dt/vols_[cell1];

scalar nu = nMin*nu0;                   //Collision frequency
scalar collProb = exp(-nu);             //Collision probability
scalar xx = rndGen_.scalar01();         //Random number between 0 and 1
// collision occur
if (( xx > collProb) && (mMin > VSMALL) && (mMax > VSMALL)) {
```

The intermediate part of the code deals with the probability for coalescence. In this project the focus is not on this. For grazing collisions (no coalescence) the parcel speed and direction after collision depends on the random restitution coefficient `gf`, and is calculated ensuring conservation of momentum.

```
// Grazing collision (no coalescence)
    else {

        scalar gf = sqrt(prob) - sqrt(coalesceProb);    //Restitution coefficient
        scalar denom = 1.0 - sqrt(coalesceProb);
```

```
        if (denom < 1.0e-5) {
            denom = 1.0;
        }
        gf /= denom; //Not used if coalescence is turned off

        // if gf negative, this means that coalescence is turned off
        // and these parcels should have coalesced
        gf = max(0.0, gf);

        //Density of parcel 1 and 2
        scalar rho1 = spray_.fuels().rho(pc, p1().T(), p1().X());
        scalar rho2 = spray_.fuels().rho(pc, p2().T(), p2().X());

        scalar m1 = p1().m();           //Mass of parcel 1
        scalar m2 = p2().m();           //Mass of parcel 2
        scalar n1 = p1().N(rho1);       //Number of droplets in of parcel 1
        scalar n2 = p2().N(rho2);       //Number of droplets in of parcel 2

        //New velocities are calculated using the random restitution coefficient
        // gf -> 1 => v1p -> p1().U() ...
        // gf -> 0 => v1p -> momentum/(m1+m2)
        vector mr = m1*v1 + m2*v2;

        //Conservation of momentum is assured
        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;
        }
```

1.2.3 The trajectory collision model

If we instead want to use the trajectory model [2] we specify this in `constant/sprayProperties` and add the coalescence switch again (set to `off` even in this case if we want to use it for solid particles). `cSpace` and `sTime` are model constants related to spatial and temporal collision probability decay.

```
collisionModel  trajectory;//off;
trajectoryCoeffs
{
    cSpace          1;
    cTime           0.3;
    coalescence     off;
}
```

`trajectoryModel.C` and `trajectoryModel.H` look the same as the O'Rourke model files. The difference is how the collision is handled once two parcels are in the same cell. This is decided in the header file `trajectoryCM.H` which is similar to the header file `sameCell.H` in the O'Rourke collision model. The main difference is that the direction of travel of the parcels is taken into account, see figures 1.1-1.2.

```
vector p = p2().position() - p1().position();   //Distance vector between parcels
scalar dist = mag(p);                           //Distance between parcels

//Dot product of vRel and norm. distance vector
scalar vAlign = vRel & (p/(dist+SMALL));

if (vAlign > 0)    //Collision ONLY IF the parcels are traveling towards each other
{
```

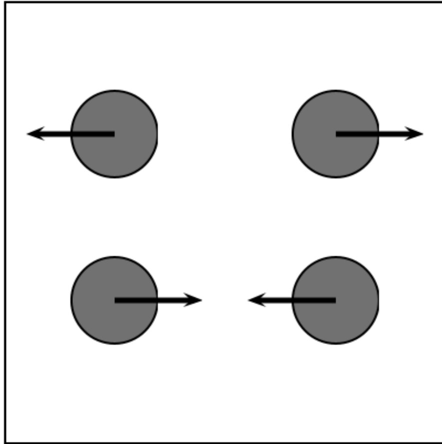

Figure 1.1: Velocity direction does not matter in O'Rourke. Opposite directions have just as high chance of collision as two particles heading towards each other.

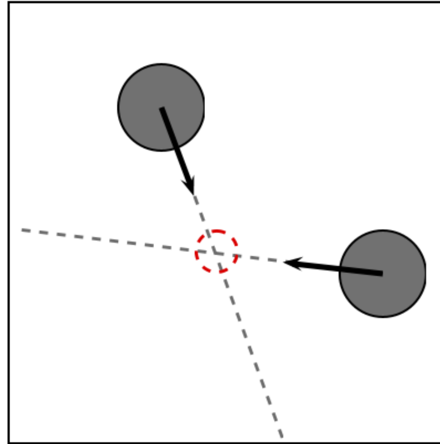

Figure 1.2: Trajectory model takes direction into account, as well as checks if collision can occur within the current time step.

The second difference to the O'Rourke model is the consideration if a collision is possible within the current time step, see eq. 1.2. Compared to the O'Rourke model the trajectory model takes care of the position and direction of travel of the parcels. The distance between the parcels must be smaller than the maximal distance defined by the relative velocity between the parcels times the time step.

$$\mathbf{U}_{align}\Delta t > |\mathbf{x}_2 - \mathbf{x}_1| - \frac{D_{max} + D_{min}}{2} \tag{1.2}$$

where

$$\mathbf{U}_{align} = U_{rel} \cdot \frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

The collision probability is calculated roughly the same way as in O'Rourke, with a few modifications. `alpha` and `beta` are used to ensure that the parcels, when they reach the projected collision point, will be there at the same time.

```
scalar v1Mag = mag(v1); //Magnitude of velocity of parcel 1
scalar v2Mag = mag(v2); //Magnitude of velocity of parcel 2
vector nv1 = v1/v1Mag;  //Normalized velocity of parcel 1
vector nv2 = v2/v2Mag;  //Normalized velocity of parcel 2

scalar v1v2 = nv1 & nv2;//Dot product of the normalized velocities

//Dot product of the normalized velocity and distance vector
scalar v1p = nv1 & p;
```

```
scalar v2p = nv2 & p;

//The probability is calculated using a special condition
scalar det = 1.0 - v1v2*v1v2;

scalar alpha = 1.0e+20;
scalar beta = 1.0e+20;

if (mag(det) > 1.0e-4)
{
    beta = -(v2p - v1v2*v1p)/det;
    alpha = v1p + v1v2*beta;
}

alpha /= v1Mag*dt;
beta /= v2Mag*dt;

// is collision possible within this time step
if ((alpha>0) && (alpha<1.0) && (beta>0) && (beta<1.0))
{
    vector p1c = p1().position() + alpha*v1*dt; //New parcel position
    vector p2c = p2().position() + beta*v2*dt;  //New parcel position

    scalar closestDist = mag(p1c-p2c); //New distance between parcels

    scalar collProb =  //Probability of collision
        pow(0.5*sumD/max(0.5*sumD, closestDist), cSpace_)
       *exp(-cTime_*mag(alpha-beta));
```

Chapter 2

Tutorial: Particle collisions

2.1 Collisions in solidParticle

Currently, there is no collision model implemented in the solidParticle class. In order to introduce such a model, several changes need to be made. Without going into too much detail on how the solidParticle class is built, this section describes the parts needed in order to implement a simple collision model in this class. The O'Rourke and trajectory collision model implemented in the dieselSpray class that were described previously will not be implemented for the collisions of solid particles. Instead a much simpler model of only two identical solid particles will be used. The chosen model is developed for solid particles, which makes more sense than to use a model created for droplets (where phonomena like breakup and coalescence are present). Implementing the entire structure of the collision models in dieselSpray was considered too complex and more suitable for a larger project, hence we will study only two particles.

2.1.1 The new solid particle collision model

The collision model we will use in this project is from ERCOFTAC [3]. It is based on preserving momentum between particles. The model is simplified by saying that the angular velocity of the particles is **0** and that the collisions are non-sliding. The velocities are then reduced to the following expressions.

$$u^*_{p1} = u_{p1} + \tfrac{J_x}{m_{p1}}, \qquad u^*_{p2} = -\tfrac{J_x}{m_{p2}}$$

$$v^*_{p1} = v_{p1} + \tfrac{J_y}{m_{p1}}, \qquad v^*_{p2} = -\tfrac{J_y}{m_{p2}}$$

$$w^*_{p1} = +\tfrac{J_z}{m_{p1}} = 0, \quad w^*_{p2} = -\tfrac{J_z}{m_{p2}} = 0$$

where * denotes the new values and J_x, J_y, J_z are the components of the impulsive force and reads (again reduced by the assumption of no angular velocity or sliding):

$$J_x = -(1+e)u_{p1}\tfrac{m_{p1}m_{p2}}{m_{p1}+m_{p2}}$$

$$J_y = -\tfrac{2}{7}v_{p1}\tfrac{m_{p2}m_{p2}}{m_{p1}+m_{p2}}$$

$$J_z = 0$$

where e is the coefficient of restitution. Note that the new velocities for BOTH particles depend only on mass of the particles and the old velocity of the FIRST particle. This will turn out to be an important part of programming the new model.

2.1.2 Creating collidingSolidParticleFoam

Now let us get started with the modifications. First we need to get solidParticleFoam, created by Håkan Nilsson. Do this through subversion:

```
cd $FOAM_RUN
svn checkout http://openfoam-extend.svn.sourceforge.net/svnroot/openfoam-extend/\
trunk/Breeder_1.5/solvers/other/solidParticleFoam/
cd solidParticleFoam/
```

Rename the directory for the new solver and copy the needed solidParticle files into it. Find and replace all occurances of `solidParticle` with `collidingSolidParticle` in all the files and then rename the files.

```
mv solidParticleFoam collidingSolidParticleFoam
cd collidingSolidParticleFoam
wclean
cp $FOAM_SRC/lagrangian/solidParticle/solidParticle* .
cp -r $FOAM_SRC/lagrangian/solidParticle/lnInclude .

sed -i s/solidParticle/collidingSolidParticle/g solidParticle.C \
solidParticleCloud.C solidParticleCloud.H solidParticleCloudI.H \
solidParticleFoam.C solidParticle.H solidParticleI.H solidParticleIO.C

mv solidParticle.C collidingSolidParticle.C
mv solidParticleCloud.C collidingSolidParticleCloud.C
mv solidParticleCloud.H collidingSolidParticleCloud.H
mv solidParticleCloudI.H collidingSolidParticleCloudI.H
mv solidParticleFoam.C collidingSolidParticleFoam.C
mv solidParticle.H collidingSolidParticle.H
mv solidParticleI.H collidingSolidParticleI.H
mv solidParticleIO.C collidingSolidParticleIO.C
```

Edit the files Make/files

```
collidingSolidParticleFoam.C
collidingSolidParticle.C
collidingSolidParticleIO.C
collidingSolidParticleCloud.C

EXE = $(FOAM_USER_APPBIN)/collidingSolidParticleFoam
```

and Make/options

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

EXE_LIBS = \
  -lfiniteVolume \
  -llagrangian
```

We can now compile the solver and run the supplied case, in order to make sure that everything is working properly. In the solver directory, type

```
wmake
cd ../box
blockMesh
collidingSolidParticleFoam >log
```

To post process, type

```
foamToVTK
paraview
```

Load by clicking `File>Load State>baseState.pvsm`. A box with two moving particles should open up. Notice that no collision occurs using this solver. We can now modify this without affecting the original code.

2.1.3 Detecting collisions

The condition used to detect the occurrence of a collisions is similar to that of the O'Rourke model in the dieselSpray class in the sense that the particles have to be in the same cell. A second condition is that the distance between the two particles at the current time has to be less or equal to the average particle diameter. This latter condition is a simpler version of the one used in the trajectory model, where a possible collision is checked for the entire duration of the time step. To use this model, changes in collidingSolidParticleCloud.C have to be made. After the member functions, add the following:

```
void Foam::collidingSolidParticleCloud::checkCell()
{

  List<label> lcell((*this).size());
  List<scalar> ld((*this).size());
  List<vector> lU((*this).size());
  List<vector> lposition((*this).size());
  bool collision;

  label i=0;
  forAllConstIter(Cloud<collidingSolidParticle>,*this,iter)
  {
      const collidingSolidParticle& p=iter();
      lcell[i]=p.cell();
    //Info <<"Particle " <<i<< " is in cell "<<lcell[i]<<endl;
      lU[i]=p.U();
      ld[i]=p.d();
      lposition[i]=p.position();
      i++;
  }

    //Info <<"p0-p1 = "<<mag(lposition[0]-lposition[1])<<endl;
    //Info <<"Diameter = "<<(ld[0]+ld[1])/2<<endl;

      //Only works for two particles.
      if (lcell[0]==lcell[1] && mag(lposition[0]-lposition[1])<=(ld[0]+ld[1])/2)
      {
          collision=true;
      }
      else
      {
          collision=false;
      }

      collision_=collision;
```

```
    U0_=lU[0];
  }
```

Also, make sure to add `#include "vector.H"` to the list of included files. The above function checks if the two particles are in the same cell. Lists of particle cell, diameter, velocity and position are created and corresponding values are assigned in the following for-loop. In the if-statement we check if the collision requirements are satisfied. At the end we save the velocity of the first particle for the current time step, which will be used to calculate new values after collision. In `collidingSolidParticleCloud.H`, add

```
  bool collision_;
  vector U0_;
```

to the private member data, and

```
  void checkCell();
  bool collision(){return collision_;};
  inline vector  U0(){return U0_;};
```

to public member functions.

2.1.4 Velocity change upon collision

In the solidParticle class the particles are collected in what is called a cloud, similar to a spray in the dieselSpray class. This cloud, here declared as `particles` is of the collidingSolidParticleCloud class. In order to get the particles in the cloud to move, we have the member function `Foam::collidingSolidParticleCloud::move(const dimensionedVector& g)`. This function, in turn, uses another `move` function located in the `collidingSolidParticle` class. This is where the velocity of the particle is changed, and here the expressions for the new velocity after a collision will be inserted. We need to add a substantial piece of code. In `collidingSolidParticle.C` replace

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

with

```
        scalar e = td.spc().e();                        //Restitution coefficient
        scalar m = rhop*d_*d_*d_*mathematicalConstant::pi*4.0/3.0; //Particle mass
        bool checkcoll=td.spc().collision();
        //if(checkcoll){Info<<"Particles collide!"<<endl;}
        vector V0=td.spc().U0(); //OLD velocity of particle 1

        scalar Jx = -(1.0+e)*V0.x()*m*m/(2.0*m);        //Impulsive force x-comp
        scalar Jy = -2.0/7.0*V0.y()*m*m/(2.0*m);        //Impulsive force y-comp

        if(ID_==0 && checkcoll) //Collision for particle 1
        {
            U_.x()=V0.x()+Jx/m;
            U_.y()=V0.y()+Jy/m;
            U_.z()=0.0;
        }
        if(ID_==1 && checkcoll) //Collision for particle 2
        {
            U_.x() = -Jx/m;
            U_.y() = -Jy/m;
            U_.z() = 0.0;
        }
```

```
        //If no collision
        U_ = (U_ + dt*(Dc*Uc + (1.0 - rhoc/rhop)*td.g()))/(1.0 + dt*Dc);
```

The restitution coefficient is defined, particle mass is computed from density and diameter, occurrence of collision is checked and the velocity for the first particle is assigned variable V0. In the next section we calculate the new velocities for the particles. This is why we need V0. The `ID_` variable is used to distinguish between the two particles and is specified in a dictionary in the 0 directory of the case (see section 2.1.5). This is a rather complex way of keeping track of both particles, but was found to be necessary. If/when expanding the code to apply for more general cases with large amounts of particles, this needs to be taken care of. In addition to the changes in the .C file, we also need some additional lines in `collidingSolidParticle.H` and `collidingSolidParticleIO.C`.
Add

```
//- Particle ID
scalar ID_;
```

after `vector U_;` in the private member data in `collidingSolidParticle.H`.

In `collidingSolidParticleIO.C` add

```
IOField<scalar> ID(c.fieldIOobject("ID"));
c.checkFieldIOobject(c, ID);
```

after `c.checkFieldIOobject(c, d);` and

```
p.ID_ = ID[i];
```

after `p.U_ = U[i];` in the `readFields` function.
Similarly, add

```
IOField<scalar> ID(c.fieldIOobject("ID"), np);
```

after `IOField<vector> U(c.fieldIOobject("U"), np);` and

```
ID[i] = p.ID_;
```

after `U[i] = p.U_;` and

```
ID.write();
```

after `U.write();` in the `writeFields` function.
The last step is to call the `checkcell` function from `collidingSolidParticleFoam.C`. Before `particles.move(g);` add

```
particles.checkCell();
```

Compile again by typing `wmake`. Now you should have a working solver `collidingSolidParticleFoam`. It is now time to set up a case and make some particles collide.

2.1.5 Creating the case

We will make some changes to the `box` case supplied. First change the mesh parameters . In `constant/polyMesh/blockMeshDict` change the blocking to

```
hex (0 1 2 3 4 5 6 7) (3 3 3) simpleGrading (1 1 1)
```

Run `blockMesh` to get a new mesh. In `0/lagrangian/defaultCloud/` we have files for particle diameter as well as particle initial positions and velocities. We need to add a file for particle IDs. Copy the diameter file

```
cp d ID
```

and change it's context to

```
FoamFile
{
  version     2.0;
  format      ascii;
  class       scalarField;
  location    "0";
  object      ID;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
2(0 1)
// ************************************************************************* //
```

In the `U` file, change the velocities to `2((1.0 0 0) (-1.0 0 0))`. In `positions`, set the coordinates to

```
2((1e-2 9e-2 0.05) 15 (7e-2 9e-2 0.05) 16)
```

What is set is the x, y and z coordinate of the two particles, as well as specifying which cell they are in (in this case 72). The values above can of course be set to whatever you want, but this will show a nice collision.

2.1.6 Running the case

While in the `box` directory, run the case by typing

```
collidingSolidParticleFoam >log\
```

To post process, type

```
foamToVTK
paraview
```

Load by clicking `File>Load State>baseState.pvsm`. The particle ID, described in section 2.1.4, is helpful in post-processing. In the `Display` tab for `Glyph1` choose to color the glyphs by ID, see Figure 2.1. This will let us distinguish between the two particles. Change the glyph type to `Cone` in order to see the direction of the particles. Hitting the play button will start the animation of the two particles.

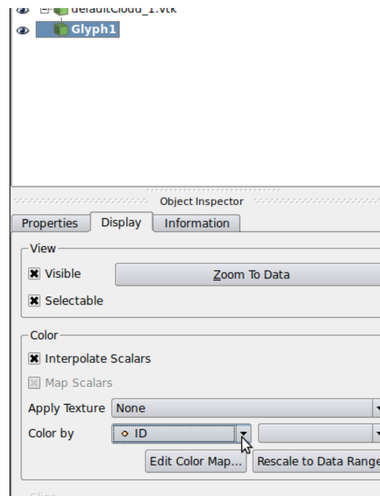

Figure 2.1: Color particles by ID

2.1.7 Issues and recommendations

Examining the collision closely in paraview, it can be seen that the colliding particles change their position before the new velocities have an effect, see figure 2.2. The velocities are calculated according to the collision model, but the new positions of the particles are calculated using previous values. In the figure it is shown that in the time step after the collision occurs, the direction of the cones are in the direction of the new velocity vectors, but in the wrong coordinates. In the following time step, however, the particles follow the correct path. As a consequence of this error, it is possible for two particles that have recently collided, to do so again in the next few time steps (almost happens in the third step after collision in figure 2.2) , which would not be possible if the velocity change was instantaneous. For a more accurate model this needs to be sorted out. As mentioned before,

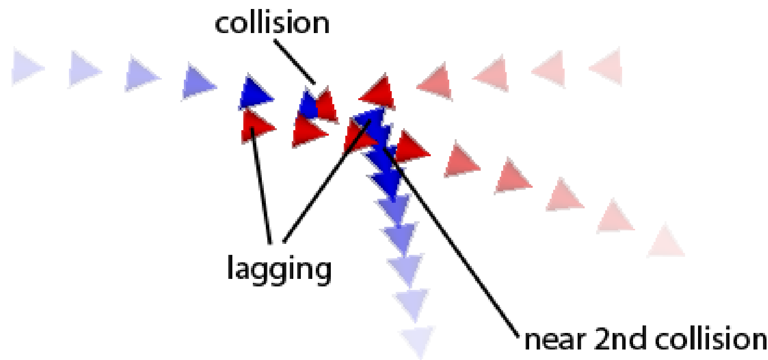

Figure 2.2: Lag in velocity switch after collision

the solver applied in this tutorial only works for two particles. If a greater number of particles is desired, the way of hard coding all particle parameters is not sustainable. The collision check used in `collidingSolidParticleCloud.C` needs to be looped for all particles instead of comparing only two. In `collidingSolidParticle.C` the velocities are computed depending on the particle ID. A more general code for this needs to be written, since now we only check if the ID is 0 or 1. Something similar to the structure of the collision models in dieselSpray should be possible. The collision model

applied in this tutorial requires the time step to be small, since collision conditions are checked only for the exact time step values and not for the duration of the time step, as in the trajectory model. This could be taken care of by implementing a similar expression.

Bibliography

[1] A.A. Amsden, P.J. O'Rourke, T.D. Butler *KIVA-II: A Computer Program for Chemically Reactive Flows with Sprays*. Los Alamos, New Mexico, 1989.

[2] N. Nordin *Complex chemistry modeling of diesel spraycombustion*. Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Göteborg, 2001.

[3] ERCOFTAC *The Best Practice Guidelines for Computational Fluid Dynamics of turbulent dispersed multiphase flows, 2008*