Chalmers University of Technology

CFD with OpenSource software, project assignment

Lagrangian particle tracking of spheres and cylinders

Prepared by: Jelena Andric

Reviewed by: Håkan Nilsson
Piero Iudiciani

Gothenburg, 2009

Contents:

Introduction

The main part of this tutorial is description of two OpenFOAM classes :*solidParticle* and *solidParticleCloud* in high details. Those classes stand for solid *spherical* particles. Moreover two new classes, *solidCylinder* and *solidCylinderCloud,* are built in order to track *cylindrical* particles. The implementation of the new solver is described as well.

On the other hand, the first part of the tutorial is dedicated to theoretical background of the problem, while the last one shows the results for spherical and cylindrical particles,using the existing and created classes.

1. About multiphase flows

The equations for motion and thermal properties of single-phase flows are well accepted (Navier-Stokes equations). The major difficulty is the modeling and quantification of turbulence and its influence on mass, momentum and energy transfer. Computational Fluid Dynamics has already a long history and a number of commercially available CFD-tools for this type of flow.
On the other hand, the correct formulation of the governing equations for multiphase flows is still a subject of debate. Interaction between different phases makes these flows complicated and very difficult to describe theoretically.
Multiphase flows are of great importance in the industrial practice, since they can occur even more frequently than single phase flows . They are present in various forms. For example, there are transient flows with a transition from pure liquid to a vapor flow as a result of external heating, separated flows and dispersed two-phase flows ,where one phase is present in the form of particles, droplets, or bubbles in a continuous carrier phase (i.e. gas or liquid).

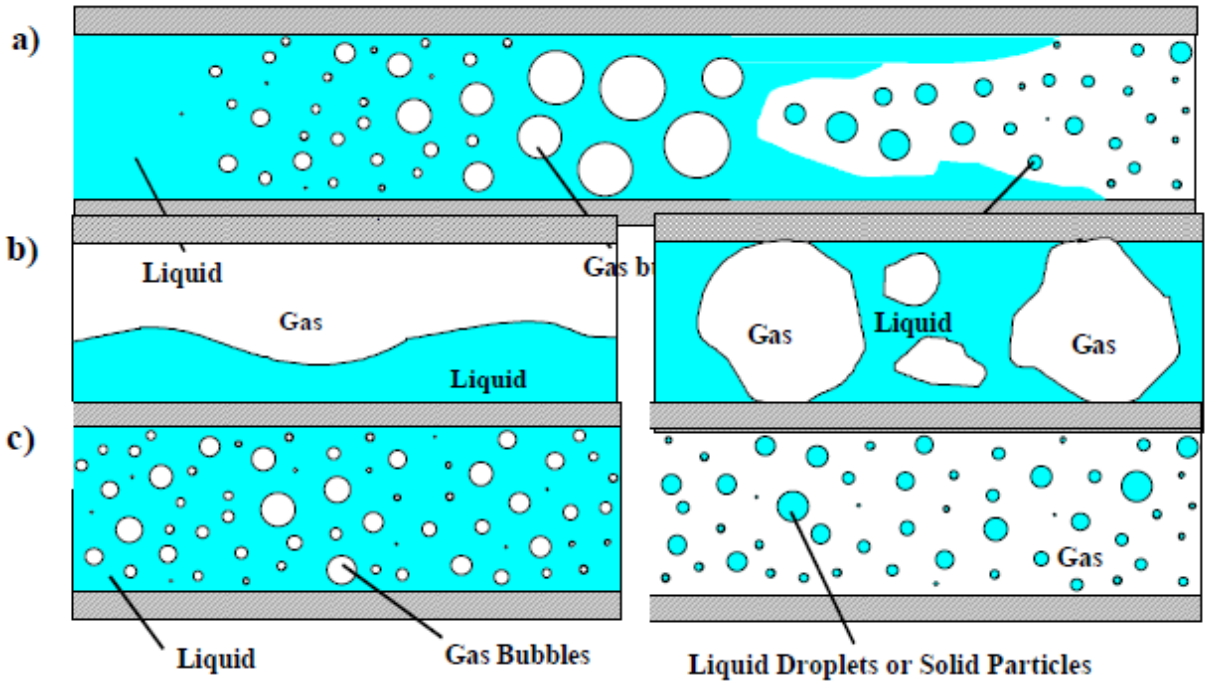

Figure1:Different regimes of two-phase flows, a)transient two-phase flow, b)separated two-phase flow, c)dispersed two-phase flow.(Source:ERCOFTAC , The Best Practice Guidelines for Computational Fluid Dynamics of turbulent dispersed multiphase flows, 2008.)

For instance, dispersed two-phase flows are encountered in numerous technical and industrial processes and may be classified in terms of the different phases being present:

- Gas-solid flows
- Liquid-solid flows
- Gas-droplet flows
- Liquid-droplet flows

Dispersed two-phase flows are usually separated into two flow regimes:

1. Dilute dispersed systems – the spacing between the particles is large, a direct interaction is rare and fluid dynamic forces are governing the particle transport.
2. Dense dispersed systems- inter –particle spacing is low (smaller than about 10 particle diameters) and the transport of particles is dominated by collisions.

The dispersed flows can be characterized by the volume fraction of the dispersed phase which represents the volume occupied by the particles in a unit volume.
A classification of dispersed two-phase flows with respect to the importance of interaction mechanisms was provided by Elghobashi (1994). Generally, it is a distinction between dilute and dense two-phase flows as can be seen in Figure2.

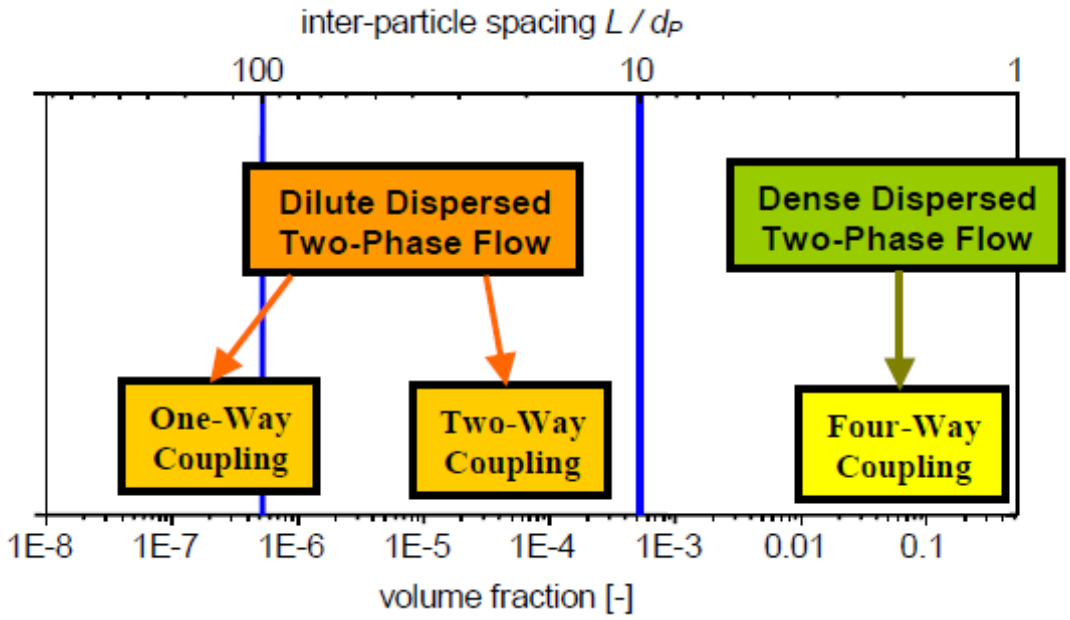

Figure2: Regimes of dispersed two-phase flows as a function of particle volume fraction.
(Source: ERCOFTAC , The Best Practice Guidelines for Computational Fluid Dynamics of turbulent dispersed multiphase flows, 2008.)

.A two-phase system may be regarded as dilute for volume fractions up to $\alpha_P > 10^{-3}$ (i.e. $L/d_P \approx 8$ .In this flow regime the influence of a particle on the fluid flow may be neglected for $\alpha_P < 10^{-6}$ .This is referred to as one-way coupling. For higher volume fractions it is necessary to account for the influence of the particles on the fluid flow – two-way coupling .In a dense regime, inter-particle interactions such as collisions and fluid dynamic interactions between particles, become important. This is so-called four way coupling.

1.1Forces acting on particles

The motion of particles in fluids is described in a Lagrangian way by solving a set of ordinary differential equations along the trajectory .The goal is to calculate the change of particle location and the linear and angular components of the particle velocity. The relevant forces acting on the particle need to be taken into account. Hence, if spherical particles are considered,the differential equations for calculating the particle location and velocity are given by Newtonian second law:

$$\frac{dx_P}{dt} = u_P \ ,$$
$$m_P \frac{d\,u_P}{dt} = \sum F_i \ , \quad (1)$$
$$I_P \frac{d\,\omega_P}{dt} = T \ .$$

where $m_P = \rho_P \, d_P^3 \pi / 6$ is mass of a particle (ρ_P is particle density and d_P is particle diameter), $I_P = 0.1 m_P \, d_P^2$ is a moment of inertia for a sphere, F_i represents the relevant forces acting on the particle, u_P is the linear velocity of a particle , ω_P is the angular velocity of a particle and T is the torque acting on a rotating particle due to the viscous interaction with the fluid.
Analytical solutions for the different forces are available for small Reynolds numbers (Stokes flow).An extension to higher Reynolds numbers is usually obtained by including a coefficient C in front of the force, where C is based on empirical correlations derived from experiments or direct numerical simulations. In most fluid-particle systems, the drag force is dominating the particle motion. Its extension to higher particle Reynolds number is based on the introduction of a drag coefficient C_D , defined as:

$$C_D = \frac{F_D}{\frac{\rho_F}{2}\left(u_F - u_P\right)^2 A_P} \quad (2)$$

where u_F is the linear velocity of a fluid , $A_P = \frac{d_P^2 \pi}{4}$ is the cross-section of a spherical particle. Multiplying by m_P and dividing by the expression for m_P for a spherical particle, the drag force can be expressed by:

$$F_D = \frac{3}{4}\frac{\rho_F\, m_P}{\rho_P\, d_P}\, C_D(u_F - u_P)\,|u_F - u_P|. \quad (3)$$

The drag coefficient is given as a function of particle Reynolds number that is defined as the ratio of inertial force to friction force:

$$\mathrm{Re}_P = \frac{\rho_F\, d_P\,|u_F - u_P|}{\mu_F}. \quad (4)$$

where is ρ_P a fluid density and μ_F is a fluid dynamic viscosity.
The dependence of the drag coefficient of a spherical particle on the Reynolds number is shown in Figure3 and it is based on numerous experimental investigations (Schlichting 1965). From this dependence, several regimes associated with flow characteristics can be identified, as presented in figure Figure3:

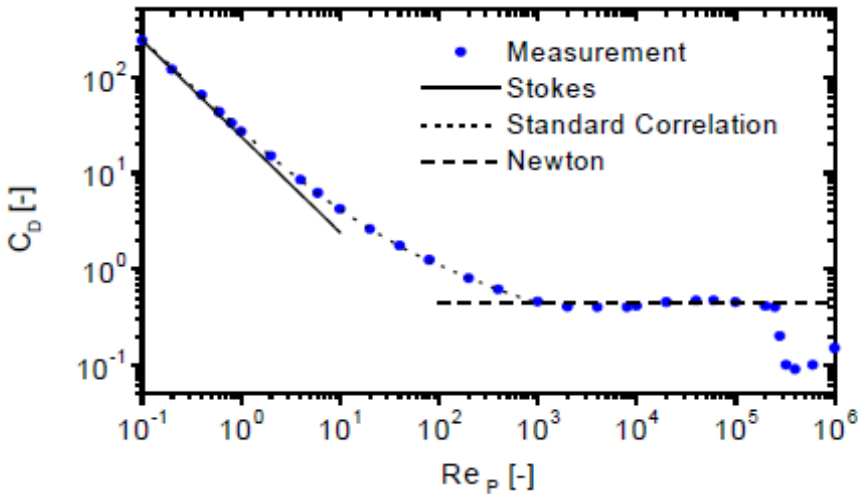

Figure3: Drag coefficient as a function of particle Reynolds number. Comparison of experimental data with the correlation for the different regimes. (Source: ERCOFTAC , The Best Practice Guidelines for Computational Fluid Dynamics of turbulent dispersed multiphase flows, 2008)

For small Reynolds number (i.e. $\mathrm{Re}_P < 0.5$) the viscous effects dominate and no separation occurs. The analytic solution for the drag is obtained Stokes (1851):

$$C_D = \frac{24}{\mathrm{Re}_P}. \quad (5)$$

This regime is often referred to as Stokes flow.

For a transition region (i.e. $0.5 < \mathrm{Re}_P < 1000$), numerous correlations have been proposed. The one frequently used is proposed by Schiller and Neumann (1933) which fits well the data up to $\mathrm{Re}_P = 1000$.

$$C_D = \frac{24}{\mathrm{Re}_P}\left(1 + 0.15\ \mathrm{Re}_P^{0.687}\right) = \frac{24}{\mathrm{Re}_P}\ f_D . \quad (6)$$

Above $\mathrm{Re}_P \approx 1000$ the flow is fully turbulent and the drag coefficient remains almost constant up to the critical Reynolds number. This regime is often referred to as the Newton-regime with:

$$C_D \approx 0.44. \quad (7)$$

At the critical Reynolds number ($\mathrm{Re}_{crit} \approx 2.5 \cdot 10^5$), there is a drastic decrease in the drag coefficient due to transition from a laminar to turbulent boundary layer around the particle.

1.2 Drag of non-spherical solid particles

Non-spherical solid bodies can be classified as either regularly shaped particles (ellipsoids, cones, disks) or irregularly-shaped particles (non-symmetric rough surfaces).Circular cylinders belong to the former class. For cylindrical particles it is straightforward to define an aspect ratio E_{cyl} in the form $E_{cyl} = L_{cyl} / D_{cyl}$ (L-length, D-diameter). Regularly shaped non-spherical particles do not typically have analytical solutions for the drag even in the creeping flow limit. Firstly their shape and the corresponding drag corrections may be approximated as ellipsoids by determining an effective aspect ratio. Additional accuracy may be obtained introducing the shape factor defined as:

$$f_{shape} \equiv \frac{C_{D,shape}}{C_{D,sphere}}\ f_{shape} \equiv \left.\frac{C_{D,shape}}{C_{D,sphere}}\right|_{\mathrm{Re}_P << 1\ \&\ const.vol.} . \quad (8)$$

To estimate the shape factor for the regularly-shaped non-spherical particles, it is common to consider two dimensionless area parameters. Those parameters are the surface and the projected area ratios. Each of those can be normalized by the surface area of a sphere that has the same volume:

$$A^*_{surf} \equiv \frac{A_{surf}}{\pi d^2}\ ,\ A^*_{proj} \equiv \frac{A_{proj}}{1/4\pi d^2} . \quad (9)$$

The inverse of the surface area ratio is more commonly defined as the “sphericity ratio”

or the “shape factor”. For a cylinder with an aspect ratio E_{cyl} , the surface area ratio and equivalent volume diameter are given as:

$$E_{cyl} \equiv \frac{L_{cyl}}{d_{cyl}} \; , \; A^*_{surf} = \frac{2E_{cyl}+1}{\left(18E^2_{cyl}\right)^{1/3}} \; , \; d = d_{cyl}\left(\frac{3E_{cyl}}{2}\right)^{1/3} . \quad (10)$$

The projected area ratio will depend on the orientation of the particle as well as its shape. For example, a long cylinder will have $A^*_{proj} > 1$ if it falls broadside, but $A^*_{proj} < 1$ if it falls vertically along its axis.

Generally it is expected that larger values of A^*_{proj} or A^*_{surf} would correspond to larger drag values, and indeed this is the case.

The following correlation of these two area ratios was suggested by Loth- for the Stokes shape correction factor:

$$f_{shape} = \frac{1}{3}\sqrt{A^*_{proj}} + \frac{2}{3}\sqrt{A^*_{surf}} \quad \text{for } \mathrm{Re}_P << 1. \quad (11)$$

The relation is based on the assumption that one-third of the drag of the sphere is given as a form drag (related to the projected area), while two thirds are the friction drag (related to the surface area).Both the form and friction drags are proportional to the particle diameter. It holds for non-spherical particles with small deviations from the sphere, so it does not hold for very high or very low aspect ratios. It gives reasonable results for many well- defined shapes with moderate aspect ratios. If a particle has a surface area ratio close to that of a spheroid, the following relationships stand for a given aspect ratio:

$$A^*_{surf} = \frac{E^{-2/3}}{2} + \frac{E^{4/3}}{4\sqrt{1-E^2}} \ln\left(\frac{1+\sqrt{1-E^2}}{1-\sqrt{1-E^2}}\right) \quad \textit{for } E < 1 , \quad (12)$$

$$A^*_{surf} = \frac{1}{2E^{2/3}} + \frac{E^{1/3}}{2\sqrt{1-E^{-2}}} \sin^{-1}\left(\sqrt{1-E^{-2}}\right) \quad \textit{for } E \geq 1. \quad (13)$$

1.2.1 Non-spherical particles in the Newtonian-drag regime

It is helpful to consider the drag at high Reynolds numbers before proceeding to intermediate values. Non-spherical particles tend to have drag coefficients that are approximately independent of Reynolds number ($10^4 < \mathrm{Re}_P < 10^5$), so that an approximately constant critical drag coefficient can be defined in the Newton-drag regime. Similar to the definition of f_{shape} this drag coefficient can be normalized by that of a sphere with the same volume:

$$C_{shape} = \left. \frac{C_{D,shape,crit}}{C_{D,sphere,crit}} \right|_{cont.vol.} . \quad (14)$$

The approximate average for a sphere is:

$$C_{D,sphere,crit} = 0.42 \quad for \quad 10^4 < \mathrm{Re}_P < 10^5 . \quad (15)$$

The drag at high Reynolds number is normally defined using a projected area. It is difficult to determine the A^*_{proj} for some particles, since the trajectories will generally include secondary motion. This means that they are not always falling in a broadside orientation (vs. area associated with volumetric diameter). For cylinders, secondary motion was found to be important at extreme aspect ratios. In general, cylinders and prolate ellipsoids can be approximately represented for a wide variety of density ratios by the following expression:

$$\left\langle C_{shape} \right\rangle \approx 1 + 0.7\sqrt{A^*_{surf} - 1} + 2.4\left(A^*_{surf} - 1\right) \quad for \; E > 1 . \quad (16)$$

1.2.2 Non-spherical particles at intermediate Reynolds numbers

There are many forms of correlations trying to predict the drag coefficient of non-spherical particles at intermediate Re_P. The most successful approaches are those which use a combination of the Stokes drag correction and the Newton-drag correction. These approaches assume that the dependence- from $\mathrm{Re}_P << 1$ to $\mathrm{Re}_{P,crit}$ is similar for all particle shapes and the difference is simply a correction at the extremes, given by f_{shape} and $C_{D,shape}$. The dependency is obtained from dimensional analysis and can be expressed as $C^*_D = f\left(\mathrm{Re}^*_P\right)$ by normalizing the drag coefficient and the Reynolds number as:

$$C^*_D = \frac{C_D}{C_{shape}} , \quad \mathrm{Re}^*_P = \frac{C_{shape}\,\mathrm{Re}_P}{f_{shape}} . \quad (17)$$

This gives good correlation for particles for a wide range of Reynolds number whose relative cross-section (C/S) is approximately circular, e.g. spheres and cylinders. For moderate particle Reynolds numbers, a normalized Schiller-Neumann expression may be similarly defined:

$$f = f_{shape}\left[1 + 0.15\left(\mathrm{Re}^*_P\right)^{0.687}\right] \quad for \; \mathrm{Re}^*_P < 800 \; \& \approx circular \; C/S . \quad (18)$$

2. Implementation in OpenFOAM

This section describes how spherical particles are implemented in *solidParticle* and solidParticleCloud classes.The implementation mainly includes the description of particle geometry, different functions for particle tracking and interpolates for the continuous phase. Then the classes *solidCylinder* and *solidCylinderCloud* are described as addition, to OpenFOAM, based on the theory described in section1.

2.1Class solidParticle

The source code is given by files *solidParticle.H* and *solidParticle.C* located in `$WM_PROJECT_DIR/src/lagrangian/solidParticle`.
This is a simple solid spherical particle class with *one-way coupling* with the continuous phase. Main parts of the code are commented bellow.
Complex inheritance as one of the main C++ and OpenFOAM characteristics can be observed while analyzing the class files.

solidParticle.H

```
#ifndef solidParticle_H
#define solidParticle_H

#include "particle.H"
#include "IOstream.H"
#include "autoPtr.H"
#include "interpolationCellPoint.H"
#include "contiguous.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

namespace Foam
{

class solidParticleCloud;

/*---------------------------------------------------------------------------*\
                           Class solidParticle Declaration
\*---------------------------------------------------------------------------*/

class solidParticle

     //It is inherited from class Particle. Inheritance is one of the key
     //features of C++     classes. Class (called a subclass or derived
     //type) can inherit the characteristics of another class(es) (super
     //class or base type) plus include its own.

     //In order to derive a class from another, a colon (:) in the
     //declaration of the derived class is used.

:
    public particle<solidParticle>
{
```

```
    //Private member data are diameter of the spherical particle
    //and velocity of particle

       //Diameter
      scalar d_;

       // Velocity of the particle
       vector U_;


public:

    //Class Cloud<solidParticle> is defined as a friend of ofsolidParticle
    //class
  friend class Cloud<solidParticle>;

    //Class trackData is defined
    //This class is used to pass tracking data to the trackToFace function

   class trackData
   {
    //- Private member is the reference to the cloud containing this
    // particle

       solidParticleCloud& spc_;

         // Interpolators for continuous phase fields are private members
         // of the class
         // Continuous phase fields are density, velocity and viscosity
         // Continuous phase fields are defined as objects of
         // interpolationCellPoint class

           const interpolationCellPoint<scalar>& rhoInterp_;
           const interpolationCellPoint<vector>& UInterp_;
           const interpolationCellPoint<scalar>& nuInterp_;

       // Local gravitational or other body-force acceleration is another
       // private member of trackData class
       const vector& g_;


   public:

       bool switchProcessor;
       bool keepParticle;


       // Constructor for trackData class

           inline trackData
           (
               solidParticleCloud& spc,
               const interpolationCellPoint<scalar>& rhoInterp,
               const interpolationCellPoint<vector>& UInterp,
               const interpolationCellPoint<scalar>& nuInterp,
               const vector& g
           );
```

```
    // Member functions to allow access to class private members

        inline solidParticleCloud& spc();

        inline const interpolationCellPoint<scalar>& rhoInterp() const;

        inline const interpolationCellPoint<vector>& UInterp() const;

        inline const interpolationCellPoint<scalar>& nuInterp() const;

        inline const vector& g() const;
};


// Constructors for solidParticle class

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

    // Construct from Istream
    solidParticle
    (
        const Cloud<solidParticle>& c,
        Istream& is,
        bool readFields = true
   );

    // Construct and return a clone
    autoPtr<solidParticle> clone() const
    {
        return autoPtr<solidParticle>(new solidParticle(*this));
    }


// Member Functions are defined in this part

    //Those allow access to class private members


        // Return diameter
       inline scalar d() const;

        // Return velocity
        inline const vector& U() const;

    // The nearest distance to a wall that
    // the particle can be in the n direction
        inline scalar wallImpactDistance(const vector& n) const;
```

```
        // Tracking is done using Boolean function move which
        //argument is an object of trackData class
        bool move(trackData&);

        // The remaining part of the code provides the functions that handle
        // the particle hitting the wall patch; this part is not listed here


    // Ostream Operator is defined as a friend function of the class

        friend Ostream& operator<<(Ostream&, const solidParticle&);
};

template<>
inline bool contiguous<solidParticle>()
{
    return true;
}

// template functions to read and write the data
template<>
void Cloud<solidParticle>::readFields();

template<>
void Cloud<solidParticle>::writeFields() const;


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#include "solidParticleI.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

#endif

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

solidParticle.C

```
#include "solidParticleCloud.H"

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

bool Foam::solidParticle::move(solidParticle::trackData& td)
{
    td.switchProcessor = false;
    td.keepParticle = true;

    const polyMesh& mesh = cloud().pMesh();
    const polyBoundaryMesh& pbMesh = mesh.boundaryMesh();

    scalar deltaT = mesh.time().deltaT().value();
    scalar tEnd = (1.0 - stepFraction())*deltaT;
    scalar dtMax = tEnd;
```

```
while (td.keepParticle && !td.switchProcessor && tEnd > SMALL)
{
    if (debug)
    {
        Info<< "Time = " << mesh.time().timeName()
            << " deltaT = " << deltaT
            << " tEnd = " << tEnd
            << " stepFraction() = " << stepFraction() << endl;
    }

    // set the lagrangian time-step
    scalar dt = min(dtMax, tEnd);

  // remember the cell the particle is in since this will change if the
  //face is hit
   label celli = cell();

   dt *= trackToFace(position() + dt*U_, td);

   tEnd -= dt;
   stepFraction() = 1.0 - tEnd/deltaT;

   cellPointWeight cpw(mesh, position(), celli, face());

    //continuous phase density
    scalar rhoc = td.rhoInterp().interpolate(cpw);
    //continuous phase velocity
    vector Uc = td.UInterp().interpolate(cpw);
    //continuous phase viscosity
    scalar nuc = td.nuInterp().interpolate(cpw);
    //particle density
    scalar rhop = td.spc().rhop();

    //particle relative velocity
    scalar magUr = mag(Uc - U_);

    scalar ReFunc = 1.0;

    // calculating particle Reynolds number
    scalar Re = magUr*d_/nuc;

    //checking particle flow region
    if (Re > 0.01)
    {
        // Schiller and Neumann correlation for transition region
        ReFunc += 0.15*pow(Re, 0.687);
    }

    //calculating drag function( inverse of particle relaxation time)
    scalar Dc = (24.0*nuc/d_)*ReFunc*(3.0/4.0)*(rhoc/(d_*rhop));

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

    if (onBoundary() && td.keepParticle)
```

```
        {
            if (isType<processorPolyPatch>(pbMesh[patch(face())]))
            {
                td.switchProcessor = true;
            }
        }
    }

    return td.keepParticle;
}

// function that handles the particle hitting the wall patch
// this is done through the particle restitution ratio
// the velocity field is separated into the normal and tangential component

void Foam::solidParticle::hitWallPatch
(
    const wallPolyPatch& wpp,
    solidParticle::trackData& td
)
{
    vector nw = wpp.faceAreas()[wpp.whichFace(face())];
    nw /= mag(nw);

    scalar Un = U_ & nw;
    vector Ut = U_ - Un*nw;

    if (Un > 0)
    {
        U_ -= (1.0 + td.spc().e())*Un*nw;
    }

    U_ -= td.spc().mu()*Ut;
}

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

2.2 Class solidParticleCloud

The source code is given by files *solidParticleCloud.H* and *solidParticleCloud.C* located in `$WM_PROJECT_DIR/src/lagrangian/solidParticle`. The main parts of the code are listed and commented bellow.

solidParticleCloud.H

```
#ifndef solidParticleCloud_H
#define solidParticleCloud_H
#include "Cloud.H"
#include "solidParticle.H"
#include "IOdictionary.H"

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

```
namespace Foam
{

// Class forward declarations
class fvMesh;
/*---------------------------------------------------------------------------*\
                           Class solidParticleCloud Declaration
\*---------------------------------------------------------------------------*/


class solidParticleCloud

//It is inherited from class Cloud
:
    public Cloud<solidParticle>
{
     // Private data are mesh (object of fvMesh class) and particle
     //properties:density, restitution ratio and friction coefficient


        const fvMesh& mesh_;

        IOdictionary particleProperties_;

        scalar rhop_;
        scalar e_;
        scalar mu_;


    // Private Member Functions are copy constructor and
    // assignment operator

        solidParticleCloud(const solidParticleCloud&);

        void operator=(const solidParticleCloud&);


public:

    // Constructors for solidParticleCloud class
    // construct the given mesh

        solidParticleCloud(const fvMesh&);


    // Member functions to access the class private members

            // Mesh
            inline const fvMesh& mesh() const;
            // Particle density
            inline scalar rhop() const;

           //Restitution ratio
            inline scalar e() const;
           //Friction coefficient
            inline scalar mu() const;
```

```
        // Move the particles under the influence of the given
        //  gravitational acceleration
        void move(const dimensionedVector& g);


        // Write fields
        virtual void writeFields() const;

};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

#include "solidParticleCloudI.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

#endif

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

3. Creating two new classes and the new solver

3.1 Classes solidCylinder and solidCylinderCloud

Two new classes called *solidCylinder* and *solidCylinderCloud* ,which stand for cylindrical particles are built from *solidParticle* and *solidParticleCloud* class respectively. In `$WM_PROJECT_USER_DIR` new directory called *solidCyliner* is created as a copy of solidParicle directory located in a `$WM_PROJECT_DIR/ src/lagrangian.`

```
cp –r $FOAM_APP/lagrangian/solidParticle/ solidCylinder
cd solidCylinder
```

The files `options` and `files` in `Make` directory should be changed as follows:

`Make/files:`

```
solidCylinder.C
solidCylinderIO.C
solidCylinderCloud.C

LIB = $(FOAM_USER_LIBBIN)/libsolidCylinder
```

`Make/options:`

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

LIB_LIBS = \
    -llagrangian \
    -lfiniteVolume
```

The file names must be modified:

```
rename solidParticle solidCylinder *
```

In files `solidCylinder.H`, `solidCylinde.C`, `solidCylinderCloud.H`, `solidCylinderCloud.C`, `solidCylinderI.H`, `solidCylinderCloudI.H`, `solidCylinderIO.C` all occurrences of `solidParticle` should be changed to `solidCylinder:`

```
sed –i s/solidParticle/solidCylinder/g solidCylinder.H
sed –i s/solidParticle/solidCylinder/g solidCylinder.C
sed –i s/solidParticle/solidCylinder/g solidCylinderCloud.H
sed –i s/solidParticle/solidCylinder/g solidCylinderCloud.C
sed –i s/solidParticle/solidCylinder/g solidCylinderI.H
```

```
sed –i s/solidParticle/solidCylinder/g solidCylinderCloudI.H
sed –i s/solidParticle/solidCylinder/g solidCylinderIO.C
```

The change in drag force due to the particle shape change must be taken into account. This is carried out in `solidCylinder.C` file in the following way:

`solidCylinder.C`

```
#include "solidCylinderCloud.H"

// * * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
// This part of the code remains unchanged
//************************************************************************** //

bool Foam::solidCylinder::move(solidCylinder::trackData& td)
{
    td.switchProcessor = false;
    td.keepParticle = true;

    const polyMesh& mesh = cloud().pMesh();
    const polyBoundaryMesh& pbMesh = mesh.boundaryMesh();

    scalar deltaT = mesh.time().deltaT().value();
    scalar tEnd = (1.0 - stepFraction())*deltaT;
    scalar dtMax = tEnd;

    while (td.keepParticle && !td.switchProcessor && tEnd > SMALL)
    {
        if (debug)
        {
            Info<< "Time = " << mesh.time().timeName()
                << " deltaT = " << deltaT
                << " tEnd = " << tEnd
                << " stepFraction() = " << stepFraction() << endl;
        }

        scalar dt = min(dtMax, tEnd);

        label celli = cell();

 dt *= trackToFace(position() + dt*U_, td);

 tEnd -= dt;
 stepFraction() = 1.0 - tEnd/deltaT;

 cellPointWeight cpw(mesh, position(), celli, face());
        scalar rhoc = td.rhoInterp().interpolate(cpw);
        vector Uc = td.UInterp().interpolate(cpw);
        scalar nuc = td.nuInterp().interpolate(cpw);
        scalar rhop = td.spc().rhop();
        scalar magUr = mag(Uc - U_);
//************************************************************************** //
//This part of the code is added to take into account for drag change//
```

```
        scalar ReFunc = 1.0;
        scalar factor = 1.0;
        scalar Re = magUr*d_/nuc;
        scalar E = 3.0;
        scalar Asurfs = (2*E+1)/pow(18*E*E,1/3);
        scalar fshape = (1/3)*pow(1.59,1/2)+(2/3)*pow(Asurfs,1/2);
        scalar Cshape = 1+0.7*pow(Asurfs-1,1/2)+2.4*(Asurfs-1);

       //it must be checked that fshape!=0,otherwise the solver will not   //
       work even though the compilation will be successful
        if (fshape!=0)
        {
           // calculating the scale factor
           scalar factor=Cshape/fshape;

        }

        if (Re > 0.01)
        {
            //taking into account for a particle shape
            scalar Re = factor*magUr*d_/nuc;
            ReFunc += 0.15*pow(Re, 0.687);
        }
        // calculating the drag function
        scalar Dc = (24.0*nuc/d_)*ReFunc*(3.0/4.0)*(rhoc/(d_*rhop));

        //calculating the particle velocity
        U_ = (U_ + dt*(Dc*Uc + (1.0 - rhoc/rhop)*td.g()))/(1.0 + dt*Dc);

//************************************************************************//

// **************The remaining part of the code is unchanged***************//
```

The command `wclean` is necessary before compiling.
The compilation is done using `wmake libso.`

3.2 SolidCylinderFoam solver as an application of solidCylinderCloud class

Firstly SolidCylinderFoam solver is obtained through svn:

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

In order to do the compilation in OpenFOAM -1.6.x, the file `readEnviromentalproperies.H` has to be copied to the solver from the OpenFOAM-1.5.x version:

```
cp /chalmers/sw/unsup/OpenFOAM-1.5.x/src/finiteVolume/ \
```

`/cfdTools/general/include/readEnvironmentalProperties.H .`

The purpose of this file is to take into account the gravity. The gravitational vector, defined in this file is used by the function *move* in *solidParticleCloud* class and while constructing an object of *trackData* class.
The solver can now be compiled in OpenFOAM-1.6.x using `wmake`.
This solver solves for the particle position and velocity. Particles are considered as spherical rigid bodies, whose properties are density, restitution ratio and friction coefficient.
In order to apply this solver to cylinder particles the following steps should be performed:

```
cp-r solidParticleFoam solidCylinderFoam
cd solidCylinderFoam
mv solidParticleFoam.C solidCylinderFoam.C
sed –i s/solidParticle/solidCylinder/g solidCylinderFoam.C
```

The files and options files in Make directory of the solver must be changed as well:

`Make/files:`

```
solidCylinderFoam.C

EXE = $(FOAM_USER_APPBIN)/solidCylinderFoam
```

`Make/options:`

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

EXE_LIBS = \
    -lfiniteVolume \
    -llagrangian \
    -lsolidParticle \
    -L$(FOAM_USER_LIBBIN) \
    -lsolidCylinder
```

Finally,it is compiled using `wmake.`

In `$WM_PROJECT_USER_DIR/solidParticleFoam` there should be four subdirectories: `box,solidParticleFoam, solidCylinderFoam,`

solidCylinder.This is where the solidCylinder directory should be located, otherwise there will be some errors while compiling.
The Box directory is used as a test case. The particles at different initial velocities are inserted into the box with given dimensions, in which the fluid is at rest. The goal is to track the motion of these particles, i.e. to solve for their positions and velocities.
Box directory contains three directories: 0, constant and system.
Directory 0 includes :lagrangian nu rho U. Files nu, rho and U contain information about the viscosity, density and velocity of the continuous phase.Subirectory lagrangian contains another directory named defaultCloud, which includes the following files: d positions U. These files contain the information about particle diameters, positions and velocities respectively.
The directory constant includes the directory polyMesh and the files environmentalProperies(in order to take into account for gravity acceleration as previously explained) and particleProperties(density, restitution ratio and friction coefficient).
Directory system as usual contains controlDict, fvSchemes and fvSolution.
More details regarding the files are provided below.

In `0/U` the velocity of the continuous phase is defined. The corresponding OpenFOAM file is shown bellow. The velocity is an object of `volVectorField` class. The units are specified in m/s. The internal field is set to `uniform (0 0 0)` (fluid at rest) and the boundary field is set to zero gradient.

```
/*--------------------------------------------------------------------------------------------------------------*\
|   =========                  |                                                                               |
|   \\      /  F ield          | OpenFOAM: The Open Source CFD Toolbox                                         |
|    \\    /   O peration      |                                                                               |
|     \\  /    A nd            | Version 1.5                                                                   |
|      \\/     M anipulation   | Web: http://www.OpenFOAM.org                                                  |
|                                                                                                              |
\*--------------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);
boundaryField
{

    Walls
    {
        type            fixedValue;
        value           uniform(0 0 0)
    }
}

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

In `0/rho` the density of the continuous phase is defined. From the corresponding OpenFOAM file It can be seen that the density is defined as an object of `volScalarField` class. The units are kg/m$^{3.}$ The internal field is set to 1 (air density at ambient temperature) while the boundary field is specified as zero gradients.

```
/*--------------------------------------------------------------------------------------------------------------*\
|   =========                 |                                                                                |
|   \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox                                          |
|    \\    /   O peration     |                                                                                |
|     \\  /    A nd           | Version 1.5                                                                    |
|      \\/     M anipulation  | Web: http://www.OpenFOAM.org                                                   |
|                                                                                                              |
\*--------------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      rho;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -3 0 0 0 0 0];

internalField   uniform 1;


boundaryField
{

    Walls
    {
        type            zeroGradient;
    }

}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

In `0/nu` the kinematic viscosity of the continuous phase is defined. The corresponding OpenFOAM file is given bellow. It can be seen that the viscosity is defined as an object of `volScalarField` class. The units are m^2/s and the boundary field is specified as zero gradient.

```
/*--------------------------------------------------------------------------------------------------------*\
|   =========                 |                                                                            |
|   \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox                                      |
|    \\    /   O peration     |                                                                            |
|     \\  /    A nd           | Version 1.5                                                                |
|      \\/     M anipulation  | Web: http://www.OpenFOAM.org                                               |
|                                                                                                          |
\*--------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      nu;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -1 0 0 0 0];

internalField   uniform 1e-6;

boundaryField
{

    Walls
    {
        type            zeroGradient;
    }

}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

In `O/lagrangian/defaultCloud/d` foam file the particle (cylinder) diameters are specified:

```
/*--------------------------------------------------------------------------------------------------------*\
|   =========                 |                                                                            |
|   \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox                                      |
|    \\    /   O peration     |                                                                            |
|     \\  /    A nd           | Version 1.5                                                                |
|      \\/     M anipulation  | Web: http://www.OpenFOAM.org                                               |
|                                                                                                          |
\*--------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       vectorField;
    location    "0";
    object      d;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
2
(
2.0e-3
2.0e-3
)
// ************************************************************************* //
```

In `O/lagrangian/defaultCloud/U` foam file the initial velocities of two particles (cylinders) are specified:

```
/*--------------------------------------------------------------------------------------------------------*\
|   =========                 |                                                                            |
|   \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox                                      |
|    \\    /   O peration     |                                                                            |
|     \\  /    A nd           | Version 1.5                                                                |
|      \\/     M anipulation  | Web: http://www.OpenFOAM.org                                               |
|                                                                                                          |
\*--------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       vectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
2
(
(1.7e-1 0 0)
(1.7 0 0)
)
// ************************************************************************* //
```

In `constant/particleProperties` foam file, the particle density, restitution ratio and friction coefficient are specified.

```
/*--------------------------------------------------------------------------------------------------------*\
|   =========                 |                                                                            |
|   \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox                                      |
|    \\    /   O peration     |                                                                            |
|     \\  /    A nd           | Version 1.5                                                                |
|      \\/     M anipulation  | Web: http://www.OpenFOAM.org                                               |
|                                                                                                          |
\*--------------------------------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      particleProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

rhop rhop [ 1-3 0 0 0 0 0 ] 1000;
e    e    [ 0 0 0 0 0 0 0 ]0.8;
mu   mu   [ 0 0 0 0 0 0 0 ]0.2;

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

In order to apply the new solver `solidCylinderFoam` ,the following steps need to be done:

```
cd ../box
blockMesh
solidCylinderFoam > log&
```

4. Results and discussion

Complete postprocessing is done in MatLab.The OpenFOAM solver ,described in previous section, provides the velocity field for each single particle that is tracked (in this particular case sphere1, sphere2, cylinder1, cylinder2) and for every time step. Corresponding values are manually extracted to Matlab and used to calculate the velocity magnitude, the particle Reynolds number and the drag coefficient.

- For each time step the velocity field is given in the form: $(u,v,0)$, where u and v stand for the velocity components in x and y directions respectively.
- The velocity magnitude (for each particle) is calculated as: $mag_P = \sqrt{u^2 + v^2}$.
- The particle Reynolds number is given by: $\mathrm{Re}_P = \dfrac{C_{shape} mag_P d}{f_{shape} \nu_C}$.
- The drag coefficient is calculated according to: $C_d = \dfrac{24}{\mathrm{Re}_P}\left(1 + 0.15\,\mathrm{Re}_P^{0.687}\right)$.

Particle diameter d , kinematic viscosity of continues phase ν_C and quantities C_{shape} and f_{shape} have already been specified in Chapter1.

4.1 Velocity field and Particle position

In Figure4 the velocity magnitudes for all four tracked particles are given. Considering sphere1 and cylinder1, which have the lower initial velocity, it can be seen that their velocity magnitudes coincide for the first 0.4 s. For sphere2 and cylinder2 that are given a higher initial velocity, the magnitudes coincide for t=0.6. Moreover, the particles with the higher initial velocity are reaching the rest phase faster comparing to those with the lower one. It can be also noticed that spherical particles will be at rest before the cylindrical ones.

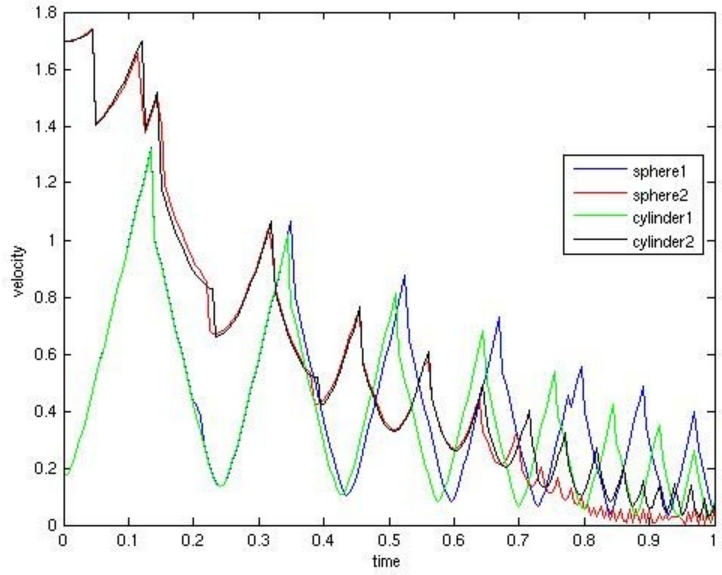

Figure4: Velocity fields for spherical and cylindrical particles

The positions of spherical particles, concerning the y-direction and for a given time are shown in Figure 5. It can be seen that the particles are bouncing against the lower wall, before they reach the rest phase.

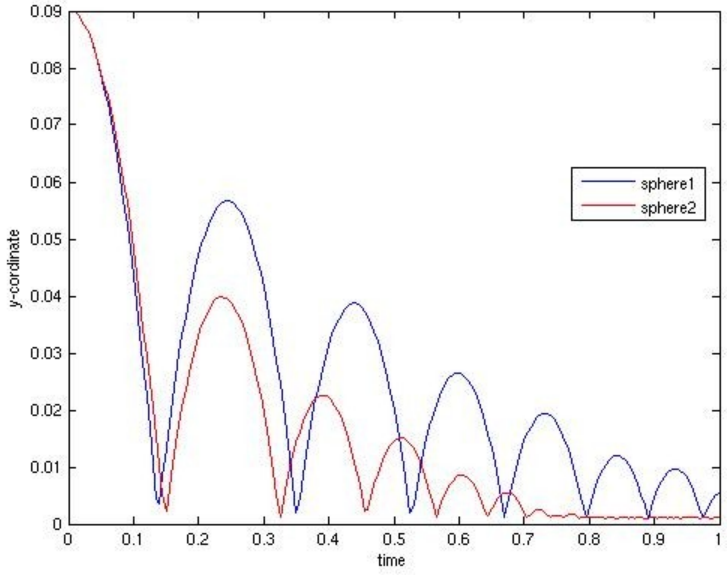

Figure5: Sphere position considering y-direction

4.2 Particle Reynolds numbers

The particle Reynolds numbers as functions of time are presented for both spherical and cylindrical particles. Analyzing the graphs in Figures 5 and 6, it can be noticed that the cylinders experience the higher Re-number than the spheres, which is physically correct. Moreover, the sphere and cylinder with the higher initial velocity first have rather high Reynolds numbers, but are fast approaching very low values. Regarding the sphere and cylinder with the lower initial velocity, the decreasing trend for Re-numbers can be noticed, except for the beginning of the simulation (first 0.2 s), where the values remain higher comparing to the sphere-cylinder pair with the higher initial velocities.

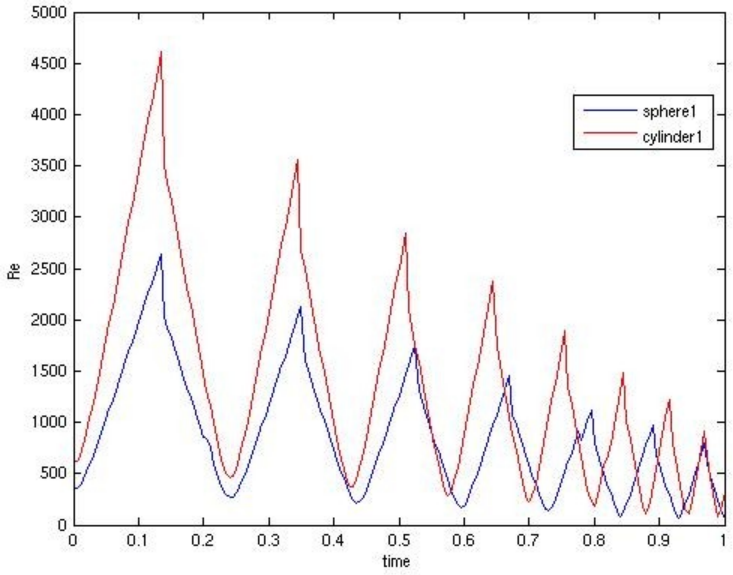

Figure6: Reynolds number as a function of time for sphere1 and cylinder1

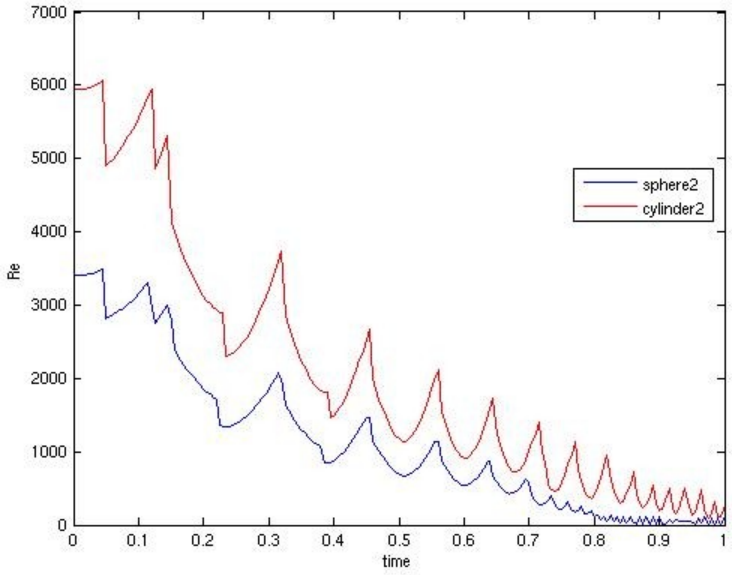

Figure7: Reynolds number as a function of time for sphere2 and cylinder2

4.3 Drag coefficient

In Figures 8-15 the dependence of the drag coefficients for spherical and cylindrical particles on time and Reynolds numbers is shown. Generally, the results are in good agreement with theory.
In Figure 8, the dependence of the drag coefficient for sphere1 and cylinder1 is presented. It can be seen that the drag coefficient has lower values for the cylindrical particle than for the spherical one, since the Reynolds number for the cylindrical particle is higher. Besides, the drag coefficients for both spherical and cylindrical particles increase with time, which is again related to the particle Reynolds numbers that decrease with time, as has already been shown.

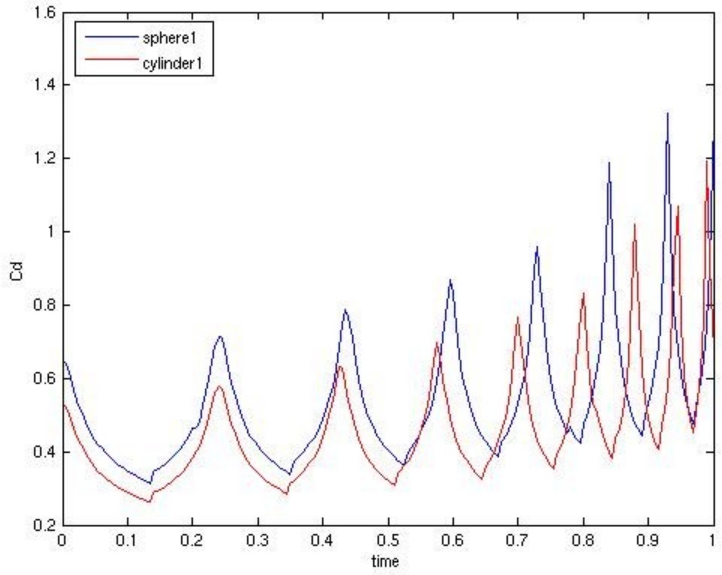

Figure8: Drag coefficient as a function of time for sphere1 and cylinder1

The similar conclusions can be drawn for sphere2 and cylinder2. The results are shown in Figures 9 and 10.
Figure 9 represents the results for the complete computational time of 1s.The drag coefficient for sphere2 cannot be clearly observed. It can be noticed that it experiences extremely high values after 0.8 s. This corresponds to the region of very low particle Reynolds numbers presented in Figure 7.This result is not physical. What can be seen in Figure 8 is rather a numerical problem that occurs when the particles are almost at rest and bouncing against the lower wall.
Figure 9 shows the results for t=0.8 s. The previously described trend for both drag coefficients can clearly be observed.

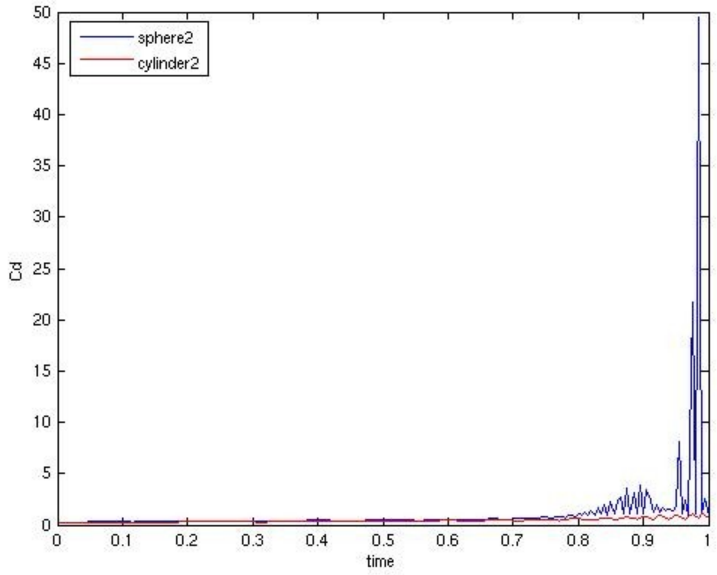

Figure9: Drag coefficient as a function of time for sphere2 and cylinder2 (computational time t=1s)

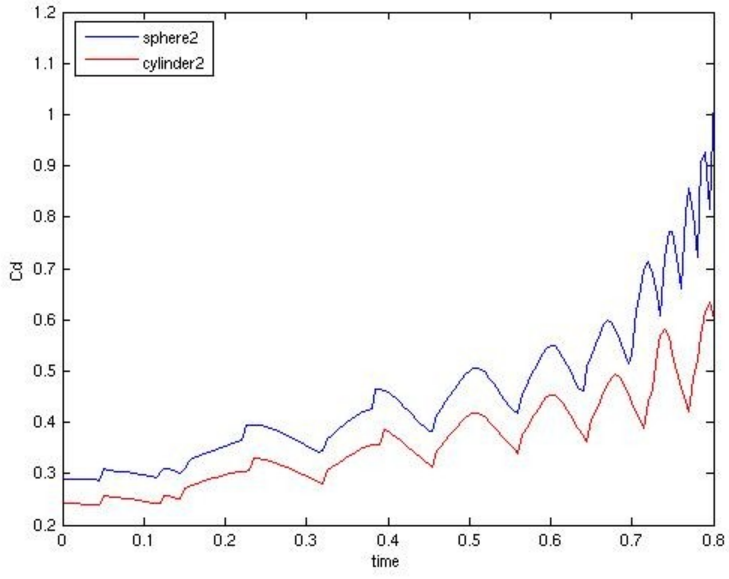

Figure10: Drag coefficient as a function of time for sphere2 and cylinder2(computational time t=0.8s)

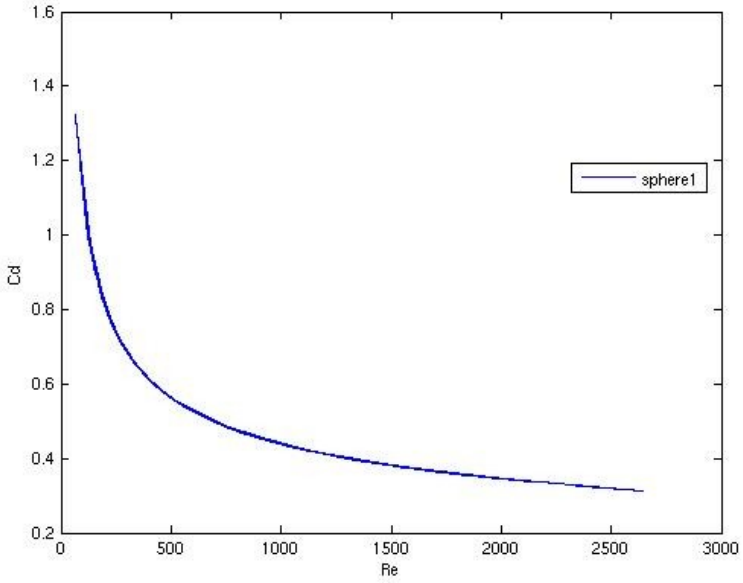

Figure11: Drag coefficient as a function of particle Reynolds number for sphere1

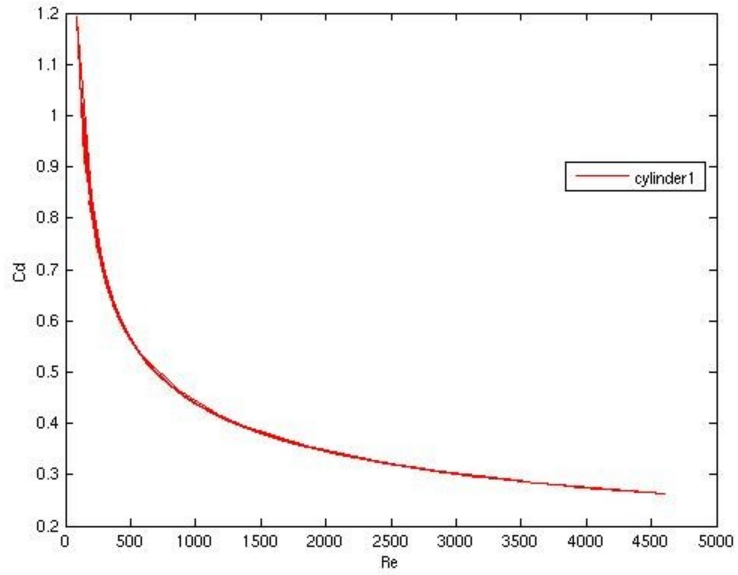

Figure12: Drag coefficient as a function of particle Reynolds number for cylinder1

Figure 13 shows the dependence of drag coefficient for sphere2 on the Reynolds number.
The same numerical problem observed while analyzing the dependence of drag coefficient for sphere2 on time is present again.

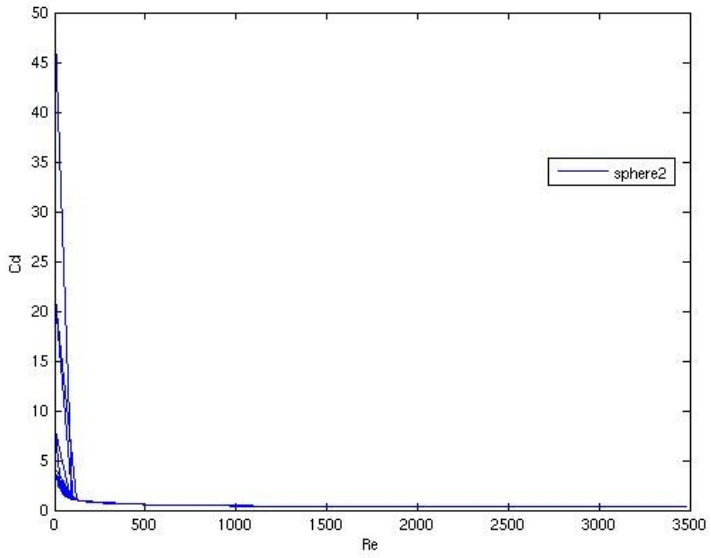

Figure13: Drag coefficient as a function of Reynolds number for sphere2

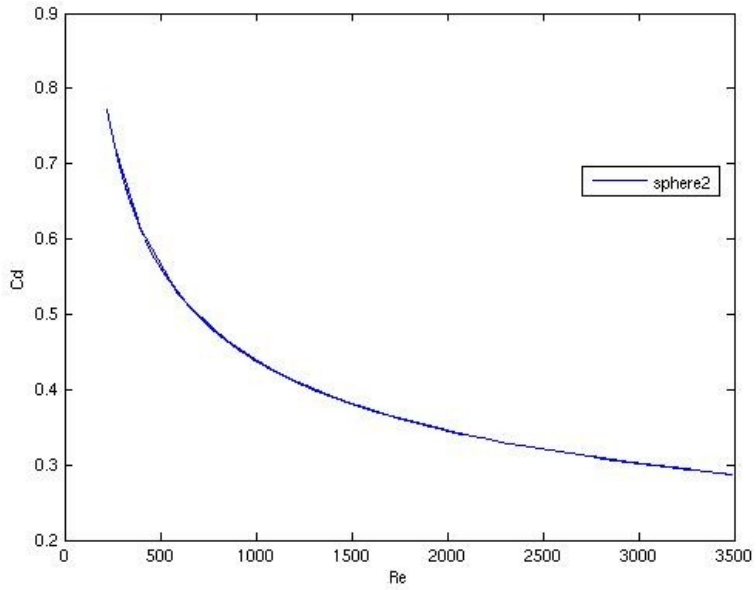

Figure14: Drag coefficient as a function of Reynolds number for sphere2 (computational time t=0.8s)

The same numerical problem is not present in Figure14 that shows the dependence of the drag coefficient on the Reynolds number for cylinder2.

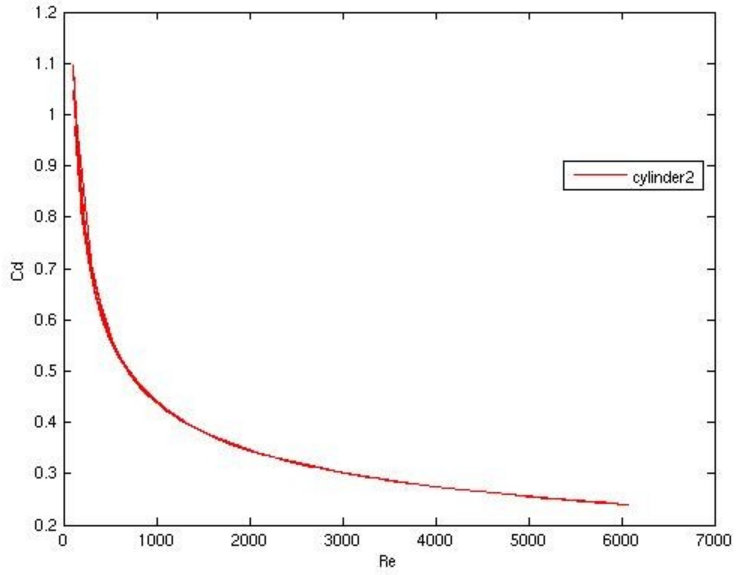

Figure15: Drag coefficient as a function of Reynolds number for cylinder2

References:

1. ERCOFTAC , The Best Practice Guidelines for Computational Fluid Dynamics of turbulent dispersed multiphase flows, 2008
2. Crowe, C.,Sommerfeld,M.,Tsuji, Y., Multiphase flows with droplets and particles, CRC Press, 1998
3. Loth, E., Drag of non-spherical solid particles of regular and irregular shape, Science Direct, 2007
4. Sasic, S., Van Wachem, B., Direct numerical simulation (DNS) of an individual fiber in an arbitrary flow field – an implicit immersed boundary method, Multiphase Science and Technology, Vol21, Issues 1-2, 2009
5. Lectures –PhD course in CFD with OpenSource software, Quarter2, 2009, Chalmers University of Technology
6. http://foam.sourceforge.net/doc/Doxygen/html/
7. http://www.cplusplus.com/doc/tutorial