Chalmers University of Technology

CFD with OpenSource software, assignment 3

Description and implementation of particle injection in OpenFOAM

Developed for OpenFOAM-1.7x

Author:
Anton Persson

Peer reviewed by:
Jelena Andric
Erwin Adi Hartono

November 5, 2010

Contents

1 Introduction and background **2**

2 Particle injection in dieselSpray **3**

2.1 Particles and injectors . . . 3
2.2 The case directory . . . 4
2.3 Data processing . . . 6
2.3.1 unitInjector . . . 6
2.3.2 hollowCone . . . 11
2.3.3 sprayInject . . . 14
2.4 Injection example . . . 15

3 A simple particle injector **19**

3.1 injectorSolidParticleFoam . . . 19
3.1.1 Directory and compilation settings . . . 19
3.1.2 injectorSolidParticleFoam.C . . . 20
3.1.3 injectorSolidParticleCloud.C . . . 21
3.1.4 createFields.H . . . 21
3.2 Example case . . . 22
3.3 Two injectors . . . 24
3.4 Evenly distributed injection . . . 25

Chapter 1

Introduction and background

This document describes the implementation of particle injection in the dieselSpray class, which is used in icoLagrangianFoam and dieselFoam. Also an implementation of a particle injector in the solidParticle class will be done.

In dieselSpray there are several different tools for introducing particles in the fluid flow that is being solved. These tools are properly called injectors. The injectors introduce particles in the domain, according to settings specified by the user. In the solidParticle class there is no injector tool. To be able to implement injection of particles in solidParticle, a deeper understanding of injectors is needed. This will be aquired by the description of the injection process that is made in this document.

Chapter 2

Particle injection in dieselSpray

2.1 Particles and injectors

There are some frequently used and useful expressions when treating particles modeled in a fluid flow. A term used in connection to computational descriptions is parcel. A *parcel* is really what is used in numerical simulations and is a representation of a gathering of real particles. This consruction is plainly made because that it is almost always too computational demanding to simulate all the real particles. To capture the behaviour of the real particles, some real case properties are defined, typically the massflow. In connection with dieselSpray the *spray* corresponds to the entire cloud containing the fuel injected. The solidParticle class contains no parcel treatment, only real particles.

In the dieselSpray class there are two objects that mainly determine how the injection of parcels will appear. These objects are the injector type and the injector model. The type determines the main characteristics of the injection based on specified physical properties as velocity, pressure, mass flow, injection region, number of parcels, mass of a parcel etc. Different injector types treat and process given information in different ways. For example, in the type commonRailInjector the velocity profile U, is calculated from

$$U = \sqrt{\frac{2(P_{inj} - P_{amb})}{\rho}}, \tag{2.1}$$

while in unitInjector the corresponding profile is based on

$$U = \frac{\dot{m}}{\rho C_D A} \tag{2.2}$$

In Eq. 2.1, P_{inj} is the pressure at the current injection position, P_{amb} the ambient pressure and ρ (also in Eq.2.2) the density of the injected parcel. In Eq.2.2, $\dot{m}$ is the mass flow of injected parcels, C_D the particle drag coefficient and A the area of the injection region.

The models in turn describes the shape of the injection region, i.e. spray appearance. An example is how in the Chomiak injector, the angle α, at which a parcel is injected is dependent on the parcel size, see Eq. 2.3. In Eq. 2.3, α_{max} is the maximum allowed injection angle, d the diameter of the current parcel and d_{min} and d_{max} the maximum and minimum diameter of all injected parcels.

$$\alpha = \frac{d - d_{max}}{d_{min} - d_{max}} \alpha_{max} \tag{2.3}$$

The result in this example is that the large parcels go straight and the smaller are injected at a larger spray angle.

To aquire the understanding of how the injectors work, it is here choosen to focus on the injector type unitInjector and the model hollowConeInjector.

2.2 The case directory

For understanding of the setup of a particle injector, let us take a closer look at the aachenBomb tutorial located in `$FOAM_TUTORIALS/dieselFoam/aachenBomb`. The directory structure is as follows

```
|-- 0
|   |-- epsilon
|   |-- ft
|   |-- fu
|   |-- k
|   |-- N2
|   |-- O2
|   |-- p
|   |-- spray
|   |-- T
|   |-- U
|   `-- Ydefault
|-- chemkin
|   |-- chem.inp
|   |-- chem.inp.1
|   |-- chem.inp_15
|   |-- chem.inp.full
|   `-- therm.dat
|-- constant
|   |-- chemistryProperties
|   |-- combustionProperties
|   |-- environmentalProperties
|   |-- injectorProperties
|   |-- polyMesh
|   |   |-- blockMeshDict
|   |   `-- boundary
|   |-- RASProperties
|   |-- sprayProperties
|   `-- thermophysicalProperties
`-- system
    |-- controlDict
    |-- fvSchemes
    `-- fvSolution
```

The files strongly connected to the injection of particles are `/constant/injectorProperties` and `/constant/sprayProperties`. A closer look in `constant/injectorProperties` shows the properties which can be defined for both the unitInjector and commonRail types. The type defined to be used in this case is though unitInjector, see below.

```
{
        injectorType        unitInjector;

        unitInjectorProps
        {
```

```
            position        (0 0.0995 0);
            direction       (0 -1 0);
            diameter        0.00019;
            Cd              0.9;
            mass            6e-06;
            nParcels        5000;

            X
            (
                1.0
            );

            massFlowRateProfile
            (
                (0 0.1272)
                (4.16667e-05 6.1634)
                (8.33333e-05 9.4778)
                (0.000125 9.5806)
                (0.000166667 9.4184)
                        .
                        .
                        .
                (0.00120833 5.1737)
                (0.00125 3.9213)
            );

            temperatureProfile
            (
                (0.0       320.0)
                (0.00125   320.0)
            );


        }
```

The `unitInjectorProps` section defines properties of the particles and is quite straight on. The parameter `position` determines the center of the (circular) region where parcels are injected and `d` the diameter of this disc (from now on called injection disc). The amount of the particle specie in kg to be injected is `mass`, `Cd` the drag coefficient and `nParcels` the total number of parcels to be injected. `X` defines the mass fraction of the particle specie. This corresponds to the first specie that is defined in `chemkin/chem.inp`. Next, `massFlowRateProfile` sets the mass flow at a certain time, $(t_i\ \dot{m})$. For `temperatureProfile` the same setup is used, i.e. $(t_i\ T)$. The temperature and mass flow properties together indicate that the last parcel will be injected at $t = 0.00125$.[1][4] Consequently the mass flow is varying during the injection while the temperature is constant. It is important that pre-calculations of the injection parameters have been done properly, at least to get exactly the desired output of the injection. For example the number of parcels, mass to be injected and the mass flow rate profile should harmonise.

In `constant/sprayProperties` which injector model that is used is specified at

```
injectorModel           hollowConeInjector;
```

The injection model specifies the size, velocity and direction of the parcels injected at the injection disc. Further down in the dictionary, coefficients for the hollow cone injector can be set:

```
hollowConeInjectorCoeffs
```

```
{
    dropletPDF
    {
        pdfType         RosinRammler;
        RosinRammlerPDF
        {
            minValue        1e-06;
            maxValue        0.00015;
            d               0.00015;
            n               3;
        }

        exponentialPDF
        {
            minValue        0.0001;
            maxValue        0.001;
            lambda          10000;
        }
    }

    innerConeAngle  ( 0 );
    outerConeAngle  ( 20 );
}
```

PDF is here short for Probability Density Function. As seen, Rosin Rammler is the one used here. Actually Rosin Rammler gives a distribution governed by a CDF, Cumulative Distribution Function. The CDF of Rosin Rammler is defined in Eq. 2.4 where F is the mass fraction of particles with diameter smaller than x, d the mean particle diameter and n a factor for setting the spread of particle size. The value of x varies in the range set by `minValue` and `maxValue` in the dictionary. Rosin Rammler distribution is commonly used for generating particle size distribution of grinding, milling and crushing operations.[2] Here it is used to create a size distribution of the simulated parcels.

$$F(x, n, d) = 1 - e^{-x/d^n} \tag{2.4}$$

When using the hollowcone injector, a spray shaped by a hollow cone will be created at the injection region. The inner and outer cone angle defines how this hollow cone should be created. If the inner cone angle, as in this case is set to zero, the cone will be solid. How this can look will be shown later.

2.3 Data processing

In this section a description of how the user provided data is processed by the unitInjector type and the hollowCone model. A short look into sprayInject will also be made.

2.3.1 unitInjector

The directory `$FOAM_SRC/lagrangian/dieselSpray/injector/unitInjector` includes the following files:

```
unitInjector.C unitInjector.dep unitInjector.H
```

The `.dep` file is created during the compilation of the `.C` file and includes all dependencies for the injector. Meaning which files are needed, how they are to be registered and where on the users machine they should be installed so that the unitInjector will work. The structure of a `.C` and `.H` file is used through out all of OpenFOAM. Running the`.H`-header file requires some other files to be included, which is done in the beginning of the file by:

```
#include "injectorType.H"
#include "vector.H"
```

Here `vector.H` is simply needed for being able to manage vectors in the code and `injectorType.H` includes some definitions needed for all injector types. Including of `.H` files are almost always needed and in them, as well as in this one there is a `ifndef` command that checks if the file has already been included by some other file. This function can save a lot of time when running large compilations. The main task done in the header file is that unitInjector is defined with its both private and public attributes. There are both, private and public attributes of data and member functions. Among the public attributes there also are constructors and a destructor. Worth noticing in the `.H` file is that a new type is defined:

```
typedef VectorSpace<Vector<scalar>, scalar, 2> pair;
```

This is done for the solver to be able to handle the type that for example `massFlowRateProfile` (see Sect. 2.2) is. With the declarations made in the `.H` file, there is no need of declaring them in the `.C` file again, since the header file is included in the `.C` file.

In `unitInjector.C` functions and constructors carries out calculations needed for the injection process. Included in this file is `unitInjector.H, addToRunTimeSelectionTable.H, Random.H` and `mathematicalConstants.H`. That `unitInjector.H` and some mathematical constants are needed is obvious. The `addToRunTimeSelectionTable.H` is used so that the unitInjector can be reached as a injectorType in injectorProperties dictionary while `Random.H` is for being able for creating random numbers. The constructors section begins with reading the properties defined in the file `injectorProperties` (see below).

```
    injectorType(t, dict),
    propsDict_(dict.subDict(typeName + "Props")),
    position_(propsDict_.lookup("position")),
    direction_(propsDict_.lookup("direction")),
    d_(readScalar(propsDict_.lookup("diameter"))),
    Cd_(readScalar(propsDict_.lookup("Cd"))),
    mass_(readScalar(propsDict_.lookup("mass"))),
    nParcels_(readLabel(propsDict_.lookup("nParcels"))),
    X_(propsDict_.lookup("X")),
    massFlowRateProfile_(propsDict_.lookup("massFlowRateProfile")),
    velocityProfile_(massFlowRateProfile_),
    injectionPressureProfile_(massFlowRateProfile_),
    CdProfile_(massFlowRateProfile_),
    TProfile_(propsDict_.lookup("temperatureProfile")),
    averageParcelMass_(mass_/nParcels_),
    pressureIndependentVelocity_(true)
```

As seen the properties `velocityProfile` and `injectionPressureProfile` is read in as `massFlowRateProfile`. This is due to that these properties are not defined in the injector properties dictionary, but still need to have values for the injection to function. They will also be corrected to values corresponding to their quantities later in the code. The average parcel mass is also calculated in this section and the property `pressureIndependentVelocity` is set as *true*. That this is set to true is directly coupled to the earlier mentioned fact that the pressure profile is not specified. Note also that a variable `CdProfile` is set equal to the mass flow rate profile. The reason for this will also be understood later.

After this `unitInjector.C` continues with controling that the user has set the start and end times of `massFlowRateProfile` and `temperatureProfile` to the same. Also the code makes sure that the time column in `massFlowRateProfile` is specified as time in s. These control and correct operations are shown in code below.

```
    // check if time entries for soi and eoi match
    if (mag(massFlowRateProfile_[0][0]-TProfile_[0][0]) > SMALL)
    {
        FatalErrorIn
        (
            "unitInjector::unitInjector(const time& t, const dictionary dict)"
        )<< "start-times do not match for TemperatureProfile and "
         << " massFlowRateProfile." << nl << exit (FatalError);
    }

    if
    (
        mag(massFlowRateProfile_[massFlowRateProfile_.size()-1][0]
      - TProfile_[TProfile_.size()-1][0])
      > SMALL
    )
    {
        FatalErrorIn
        (
            "unitInjector::unitInjector(const time& t, const dictionary dict)"
        )<< "end-times do not match for TemperatureProfile and "
         << "massFlowRateProfile." << nl << exit(FatalError);
    }

    // convert CA to real time
    forAll(massFlowRateProfile_, i)
    {
        massFlowRateProfile_[i][0] =
            t.userTimeToTime(massFlowRateProfile_[i][0]);
        velocityProfile_[i][0] = massFlowRateProfile_[i][0];
        injectionPressureProfile_[i][0] = massFlowRateProfile_[i][0];
    }

    forAll(TProfile_, i)
    {
        TProfile_[i][0] = t.userTimeToTime(TProfile_[i][0]);
    }
```

When this is done the mass flow rate profile is corrected, so that it is ensured that it matches the total mass to be injected, see Eq. 2.5. Also, a profile for the drag coefficient is created, containing the time entries of the mass flow rate profile and the value of C_d. Below Eq. 2.5, the code for this procedure is shown.

$$\dot{m}_{prof,new} = \dot{m}_{prof} \frac{m_{tot}}{\int_{t_0}^{t_{end}} \dot{m}_{prof}} \tag{2.5}$$

```
scalar integratedMFR = integrateTable(massFlowRateProfile_);

    forAll(massFlowRateProfile_, i)
    {
        // correct the massFlowRateProfile to match the injected mass
        massFlowRateProfile_[i][1] *= mass_/integratedMFR;

        CdProfile_[i][0] = massFlowRateProfile_[i][0];
        CdProfile_[i][1] = Cd_;
    }
```

After this, the injection direction vector is normalized by that it is divided it with its own magnitude (see below).

```
// Normalize the direction vector
direction_ /= mag(direction_);
```

To keep track of every parcel a coordinate system for each one needs to be defined, three directions perpendicular to each other are needed. Since the direction vector for them already is created, two vectors perpendicular to this is sufficient. This is created based on a randomly generated vector consisting of numbers between 0 and 1. The random vector, $\overline{A}_{rnd}$ gives through manipulation, a vector, $\overline{B}_{tangent}$ perpendicular to the direction vector, $\overline{dir}$. $\overline{B}_{tangent}$ is normalized and a third vector, perpendicular to both this and the direction vector, is simply created through a cross product. See steps in Eq. 2.6 below.

$$\begin{array}{lcl} \overline{A}_{rnd} & = & randomly\ generated\ vector\ in\ [0,1] \\ \overline{B}_{tangent} & = & \overline{A}_{rnd} - (\overline{A}_{rnd} \cdot \overline{dir}) * \overline{dir} \\ magV & = & magnitude(\overline{A}_{tangent}) \\ \overline{tan}_{Inj1} & = & \frac{\overline{A}_{tangent}}{magV} \\ \overline{tan}_{Inj2} & = & \overline{dir} \times \overline{tan}_{Inj1} \end{array} \tag{2.6}$$

In the code the section corresponding to Eq. 2.6 is

```
void Foam::unitInjector::setTangentialVectors()
{
    Random rndGen(label(0));
    scalar magV = 0.0;
    vector tangent;

    while (magV < SMALL)
    {
        vector testThis = rndGen.vector01();

        tangent = testThis - (testThis & direction_)*direction_;
        magV = mag(tangent);
    }

    tangentialInjectionVector1_ = tangent/magV;
    tangentialInjectionVector2_ = direction_ ^ tangentialInjectionVector1_;

}
```

The while loop is as seen to make sure that magV is SMALL, which it should be as its set to 0.0 in the beginning of the calculation.

For making sure that the mass fraction `X` sums up to one, this is checked. If it does not, this is fixed with the expression in Eq. 2.7 where a loop over `i` is done.

$$X(i) = \frac{X(i)}{sum(X)} \tag{2.7}$$

This equation is constructed so that it will work for injections with several species. The code of this is shown below.

```
// check molar fractions
    scalar Xsum = 0.0;
    forAll(X_, i)
    {
```

```
        Xsum += X_[i];
    }

    if (mag(Xsum - 1.0) > SMALL)
    {
        WarningIn("unitInjector::unitInjector(const time& t, Istream& is)")
            << "X does not sum to 1.0, correcting molar fractions."
            << nl << endl;
        forAll(X_, i)
        {
            X_[i] /= Xsum;
        }
    }
```

Also needed for the injection is how many parcels that should be injected every time step, this is done next. The number of parcels are calculated from dividing the mass to be injected by the average mass of a parcel. It is also rounded off to the closest integer, see Eq. 2.8.

$$n_{parcels} = integer\ of\ (\frac{m_{inj}}{m_{ave}} + 0.49) \tag{2.8}$$

The translation into code of Eq. 2.8 is

```
Foam::label Foam::unitInjector::nParcelsToInject
(
    const scalar time0,
    const scalar time1
) const
{
    scalar mInj = mass_*(fractionOfInjection(time1)-fractionOfInjection(time0));
    label nParcels = label(mInj/averageParcelMass_ + 0.49);
    return nParcels;
}
```

The position of the injection of each parcel, based on the injection disc diameter and the disc's center position is then defined for the solver. These positions are distributed within the injection disc according to Eq. 2.9 where S_{rnd} is a scalar with the same behaviour as the components of $\overline{A}_{rnd}$ in Eq. 2.6.

$$\begin{aligned} r_{inj} &= d * S_{rnd} \\ angle_{injPos} &= 2\pi * S_{rnd} \\ \overline{pos}_{parc} &= \overline{pos} + r_{inj} * (\overline{tan}_{Inj1} * cos(\alpha_{injAngle}) + \overline{tan}_{Inj2} * sin(\alpha_{injAngle})) \end{aligned} \tag{2.9}$$

The paranthesis in the last calculation of Eq. 2.9 forms a vector in the plane perpendicular to the direction vector, since both $\overline{tan}_{Inj1}$ and $\overline{tan}_{Inj2}$ are perpendicular to the injection direction. When adding the position vector to this vector, the position for the injected parcel starting from origo is obtained. If the case solved for is in two dimensions a conversion for this is also implemented in the code. The control of whether it is a two dimensional case or not is made in the file `spray.C` which is a central part of the dieselSpray class (located in `$FOAM_SRC/lagrangian/dieselSpray/spray`). The calculation for the three dimensional case, i.e. Eq. 2.9 can be seen in code format below.

```
Foam::vector Foam::unitInjector::position
(
    const label n,
    const scalar time,
    const bool twoD,
    const scalar angleOfWedge,
```

```
    const vector& axisOfSymmetry,
    const vector& axisOfWedge,
    const vector& axisOfWedgeNormal,
    Random& rndGen
) const
// otherwise, disc injection
        scalar iRadius = d_*rndGen.scalar01();
        scalar iAngle = 2.0*mathematicalConstant::pi*rndGen.scalar01();

        return
        (
            position_
          + iRadius
          * (
              tangentialInjectionVector1_*cos(iAngle)
            + tangentialInjectionVector2_*sin(iAngle)
          )
        );
```

As earlier mentioned the velocity and pressure profiles are corrected for the injection. This is done at the end of this file, with the expressions seen in Eq. 2.10.

$$\begin{array}{rcl} A & = & 0.25 * \pi d^2 \\ U & = & \frac{\dot{m}}{C_d \rho A} \\ p & = & p_{ref} + 0.5 \rho v^2 \end{array} \tag{2.10}$$

Eq. 2.10 is looped over the time positions specified in `massFlowRateProfile`, so that the profiles are specified at all times. Note that the calculation of the velocity profile can be recognised from the example stated in Eq. 2.2. In OpenFoam code this looks as

```
void Foam::unitInjector::correctProfiles
(
    const liquidMixture& fuel,
    const scalar referencePressure
)
{
    scalar A = 0.25*mathematicalConstant::pi*pow(d_, 2.0);
    scalar pDummy = 1.0e+5;

    forAll(velocityProfile_, i)
    {
        scalar time = velocityProfile_[i][0];
        scalar rho = fuel.rho(pDummy, T(time), X_);
        scalar v = massFlowRateProfile_[i][1]/(Cd_*rho*A);
        velocityProfile_[i][1] = v;
        injectionPressureProfile_[i][1] = referencePressure + 0.5*rho*v*v;
    }
}
```

2.3.2 hollowCone

In Section 2.2 it was mentioned that the spray properties dictionary defined which injector model that is to be used. Here hollowConeInjector is used and it is built up in the same way as the injector type, with `hollowCone.H` and `hollowCone.C`. It is located in
`$FOAM_SRC/lagrangian/dieselSpray/spraySubModels/injectorModel/hollowCone`.

In `hollowCone.H` it is first checked if the file already has been included and then files needed for the file itself are included. Needed files are: `injectorModel.H`, `scalarList.H` and `pdf.H`. Which are giving definitions for injector models in general, providing a type definition of a scalar list and allowing a number of PDF's to be used. With these files included, the class declaration with public and private attributes can be done.

The `.C` file includes except the `.H` file, `addToRunTimeSelectionTable` and `mathematicalConstants` which are needed for the same purposes as in `unitInjector.C`. The first thing done under constructors is that the data for `pdfType` (including its settings), `innerAngle` and `outerAngle` are obtained from the spray properties dictionary. It is also checked that the settings are done in a correct way. Then the reference pressure is set to the ambient pressure and the velocity profiles are corrected using the construction of Eq. 2.10 as in `unitInjector.C`. All this is seen below.

```
// Construct from components
hollowConeInjector::hollowConeInjector
(
    const dictionary& dict,
    spray& sm
)
:
    injectorModel(dict, sm),
    hollowConeDict_(dict.subDict(typeName + "Coeffs")),
    dropletPDF_
    (
        pdfs::pdf::New
        (
            hollowConeDict_.subDict("dropletPDF"),
            sm.rndGen()
        )
    ),
    innerAngle_(hollowConeDict_.lookup("innerConeAngle")),
    outerAngle_(hollowConeDict_.lookup("outerConeAngle"))
{

    if (sm.injectors().size() != innerAngle_.size())
    {
        FatalError << "hollowConeInjector::hollowConeInjector"
            << "(const dictionary& dict, spray& sm)\n"
            << "Wrong number of entries in innerAngle"
            << abort(FatalError);
    }

    if (sm.injectors().size() != outerAngle_.size())
    {
        FatalError << "hollowConeInjector::hollowConeInjector"
            << "(const dictionary& dict, spray& sm)\n"
            << "Wrong number of entries in outerAngle"
            << abort(FatalError);
    }

    scalar referencePressure = sm.ambientPressure();

    // correct velocityProfile
    forAll(sm.injectors(), i)
    {
```

```
        sm.injectors()[i].properties()->correctProfiles(sm.fuels(), referencePressure);
    }

}
```

After this, the injection direction is calculated based on the inner and outer angle. In the unitInjector section (2.3.1) it was described how the injection positions of the parcels where set using the calculation in 2.9. In the injector model, in this case hollowConeInjector, the initial direction of the injected particles are determined from a similar calculation, see Eq. 2.11.

$$\begin{aligned}
angle &= innerAngle + S_{rnd} * (outerAngle - innerAngle) \\
\alpha &= sin(angle\pi/360) \\
dir_{corr} &= cos(angle\pi/360) \\
\beta &= 2\pi * S_{rnd} \\
\hat{n} &= \alpha * (\overline{tan}_{Inj1} * cos(\beta) + \overline{tan}_{Inj2} * sin(\beta) \\
\overline{dirParc} &= dir_{corr} * \overline{dir} + \hat{n} \\
\overline{dirParc}_{norm} &= \frac{\overline{dirParc}}{magnitude(\overline{dirParc})}
\end{aligned} \tag{2.11}$$

In Eq. 2.11 *angle* is a random generated angle with a value between the inner and outer angle. A direction vector ($\overline{dirParc}$) and a normal vector ($\hat{n}$) to this is created based on the original direction, the tangentialvectors ($\overline{dir}$, $\overline{tan}_{Inj1}$ and $\overline{tan}_{Inj2}$) and *angle*. The tangentialvectors are obtained using Eq. 2.6. The result is a vector starting from origo and defining the direction of the injected parcel. As in `unitInjector.C`, there is a option that calculates the direction if the case is two-dimensional. In code, the calculation made in Eq. 2.11 looks as

```
vector hollowConeInjector::direction
(
    const label n,
    const label hole,
    const scalar time,
    const scalar d
) const
{
    scalar angle = innerAngle_[n] + rndGen_.scalar01()*(outerAngle_[n]-innerAngle_[n]);
    scalar alpha = sin(angle*mathematicalConstant::pi/360.0);
    scalar dcorr = cos(angle*mathematicalConstant::pi/360.0);
    scalar beta = 2.0*mathematicalConstant::pi*rndGen_.scalar01();

    // randomly distributed vector normal to the injection vector
    vector normal = vector::zero;
 normal = alpha*
        (
            injectors_[n].properties()->tan1(hole)*cos(beta) +
            injectors_[n].properties()->tan2(hole)*sin(beta)
        );

    // set the direction of injection by adding the normal vector
    vector dir = dcorr*injectors_[n].properties()->direction(hole, time) + normal;
    dir /= mag(dir);

    return dir;

}
```

The velocity profile calculated by the injector type is then reconstructed. If the injector type has the `pressureIndependentVelocity` property set to *true* the velocity will simply be fetched (see code below), which is the case for the unitInjector type.

```
scalar hollowConeInjector::velocity
(
    const label i,
    const scalar time
) const
{
    if (it.pressureIndependentVelocity())
    {
        return it.getTableValue(it.velocityProfile(), time);
    }
```

The `getTableValue` function, as the name suggest, gets the value of the current position in velocity profile and `it` is set to call the injector properties created by the injector type. If the hollowCone injector is used together with a injector where this property is *false*, a calculation based on the pressure profile will be carried out to construct a velocity profile, see below (continuation of `hollowConeInjector::velocity`).

```
else
    {
        scalar Pref = sm_.ambientPressure();
        scalar Pinj = it.getTableValue(it.injectionPressureProfile(), time);
        scalar rho = sm_.fuels().rho(Pinj, it.T(time), it.X());
        scalar dp = max(0.0, Pinj - Pref);
        return sqrt(2.0*dp/rho);
    }
}
```

At the end of `hollowCone.C` the average velocity is calculated by taking the mean of the velocity over the injection time. See Eq. 2.12 and following code.

$$U_{mean} = \frac{sum(U)}{no.\ of\ timesteps\ of\ injection} \tag{2.12}$$

```
scalar hollowConeInjector::averageVelocity
(
    const label i
) const
{
    const injectorType& it = sm_.injectors()[i].properties();
    scalar dt = it.teoi() - it.tsoi();
    return it.integrateTable(it.velocityProfile())/dt;
}
```

The velocity profiles are here summed up by `integrateTable`.

2.3.3 sprayInject

When the injector type and injector model have defined all necessary parameters for the injection they just need to be initialized in the code. Of course, this is coupled to many computations during a simulation. But to get a better understanding of how all calculations made in the two previous sections are used, a segment of the file `sprayInject.C` (located in `$FOAM_SRC/lagrangian/dieselSpray/spray` is shown below.

```
forAll(injectors_, i)
    {
      if (mass > 0)
         for(label j=0; j<Np; j++)
            {
            for(label n=0; n<nHoles; n++)
                {
                 if (injectorCell >= 0)
                    {
                    } // if (injectorCell....
                } // for(label n=0...
            } // for(label j=0....
        } // if (mass>0)...
    } // forAll(injectors)...
```

This piece of code shows how the injection is done. The *if statements* above are for making sure that injection really should take place and the *for loops* are there so that the injection of parcels is made as many times as desired. The most outer loop repeats as many time as there are injectors (one in the case examined here), the `Np` loop is for the number of parcels each time step (recognized from earlier) and `nHoles` is for the number of holes used in the current injector type (one for unitInjector).

With the now obtained knowledge, a picture of the injection process can be made. A try to visualize the injection created by unitInjector and hollowCone (with $innerAngle = 0$, so solid) is done in figure 2.1

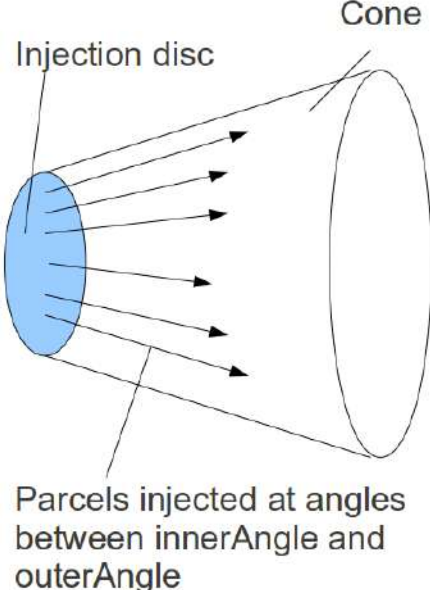

Figure 2.1: Fundamental picture of injection using unitInjector and hollowCone. Size distribution governed by Rosin Rammler, $innerAngle = 0$.

2.4 Injection example

To be able to further visualize how the injection process is carried out, snapshots of the aachenBomb tutorial (no ignition) simulation will be shown here. In the following figures, the standard run where $outerAngle = 20$ is showed together with a case where $outerAngle = 60$. In both cases $innerAngle = 0$. By studying the difference between them a interpretation of how the hollowCone

injector shapes the spray should be aquired. In the figures, the plane to the left is placed at the position of the injection disc and the clips are made to easy illustrate how the spray is spread in the domain. The negative y-direction and thus the direction of gravity is perpendicular to the plane created at the injection disc and pointing in the same direction as the top of the spray.

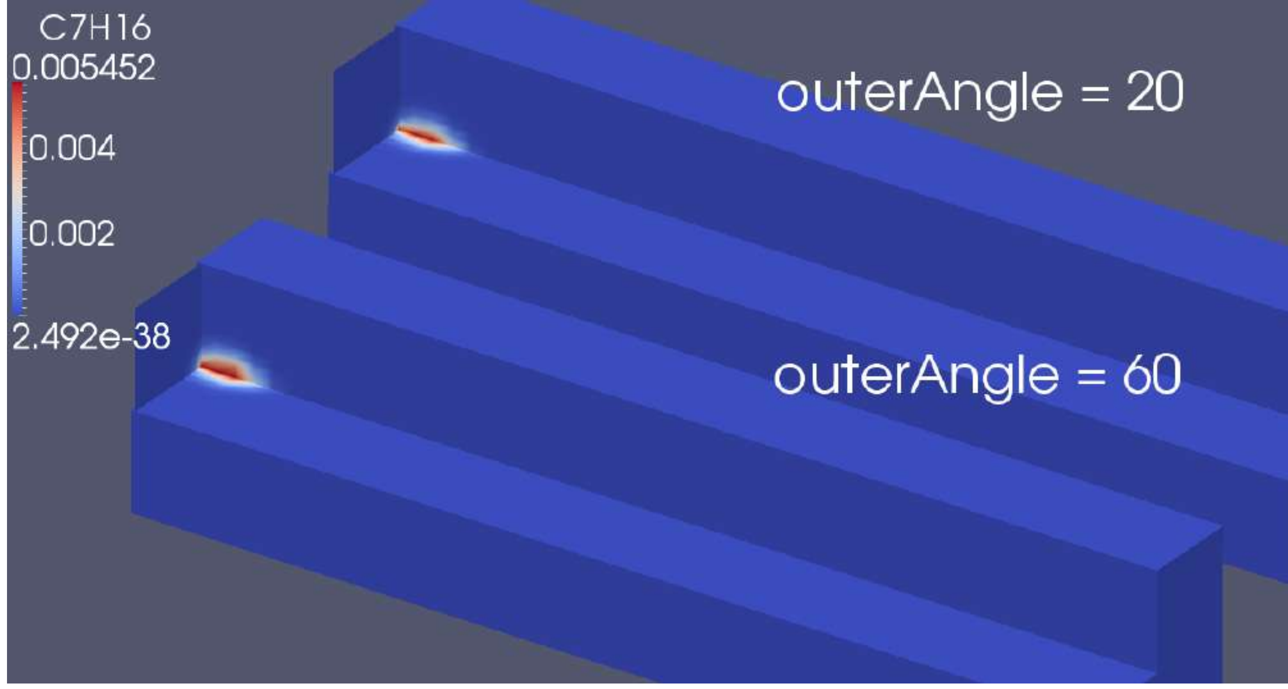

Figure 2.2: $Time = 0.0001$. Injection just initialised, sprays almost similar. But a closer look gives that the 60° case spray is a bit wider.

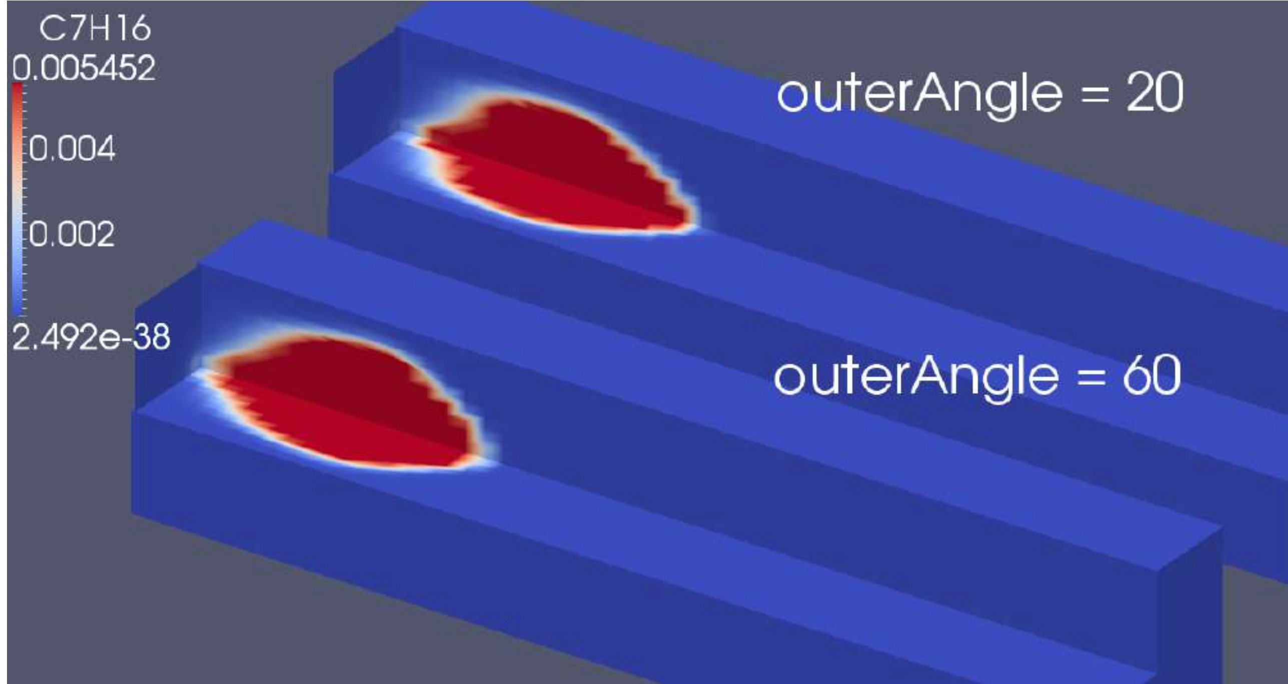

Figure 2.3: *Time* = 0.00065. The difference between the cases grow larger. A cone shape can be identified, starting from the injection disc

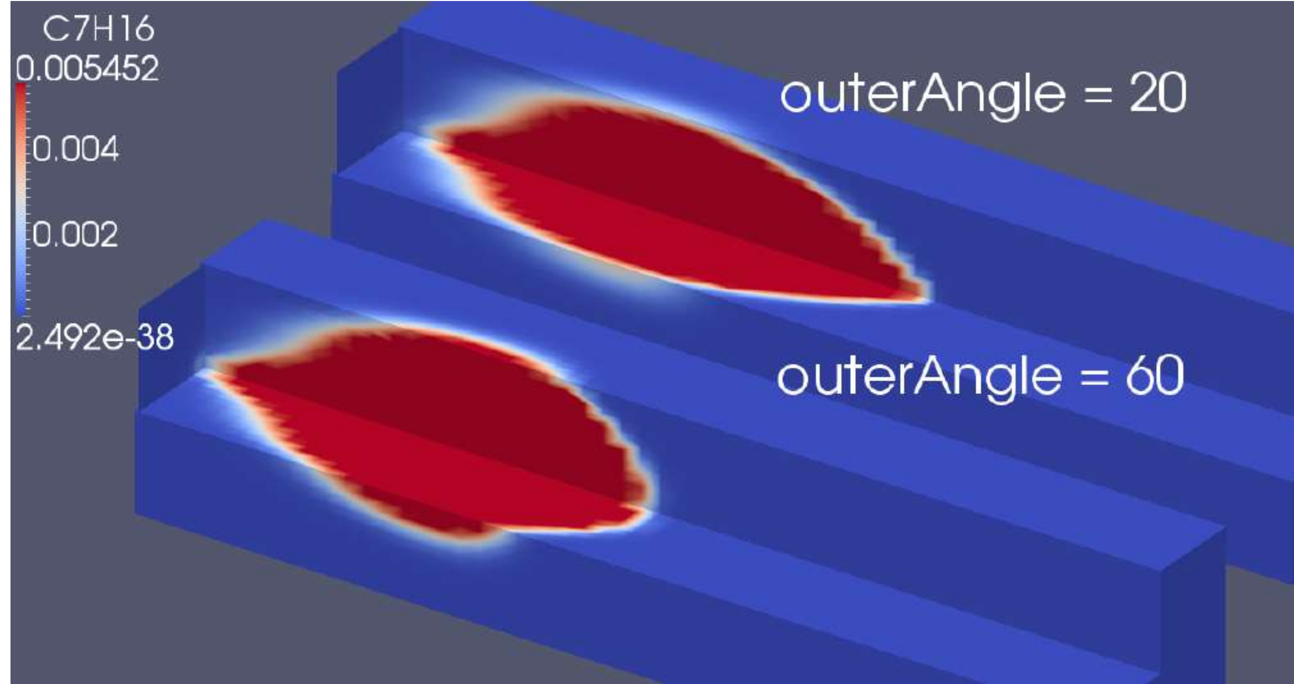

Figure 2.4: *Time* = 0.00165. The difference continues to grow. The 60° case has clearly reached the outer bondaries of the domain.

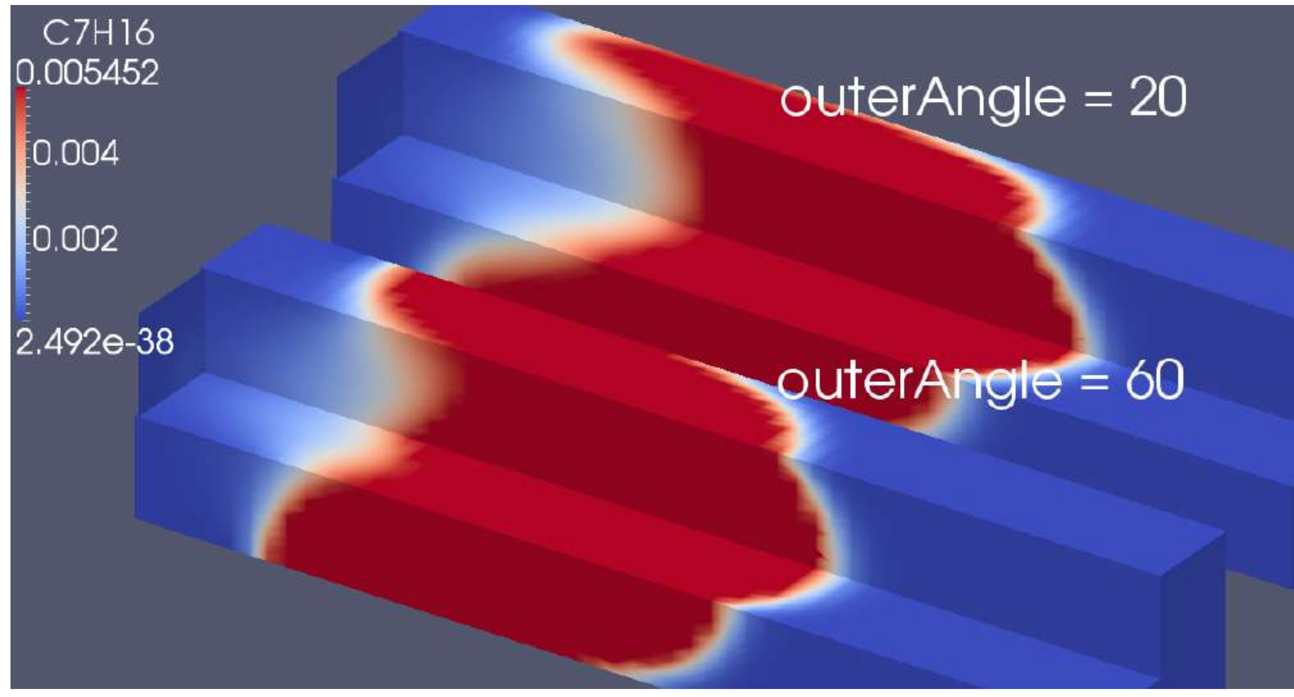

Figure 2.5: $Time = 0.01$. The last time step, both sprays have distributed evenly in the domains

Chapter 3

A simple particle injector

The solidParticle class offers simulation or at least computations for demo purposes of solid particles. Here, a particle injector will be added in the class and a case will be solved with a modified version of the simpleFoam solver.

3.1 injectorSolidParticleFoam

The modification will be compiled as a new class, `injectorSolidParticleFoam`. The first step is to set up a new directory for compilation.

3.1.1 Directory and compilation settings

First of all, create a directory for the new class.

```
mkdir $WM_PROJECT_USER_DIR/applications/solvers/injectorSolidParticleFoam
cd $WM_PROJECT_USER_DIR/applications/solvers/injectorSolidParticleFoam
```

Now copy the needed files to the new directory.

```
cp -r $FOAM_SRC/lagrangian/solidParticle/* .
cp $FOAM_SOLVERS/incompressible/simpleFoam/* .
```

Rename the files corresponding to the new class name (the indentation below is so that it is possible to easily copy the commands in all pdf-viewers).

```
 mv solidParticle.C injectorSolidParticle.C
mv solidParticle.H injectorSolidParticle.H
 mv solidParticleI.H injectorSolidParticleI.H
mv solidParticleIO.C injectorSolidParticleIO.C
 mv solidParticleCloud.C injectorSolidParticleCloud.C
mv solidParticleCloud.H injectorSolidParticleCloud.H
 mv solidParticleCloudI.H injectorSolidParticleCloudI.H
mv simpleFoam.C injectorSolidParticleFoam.C
```

Then make sure that the code within the files also agrees with the new names.

```
sed -i s/solid/injectorSolid/g injectorSolidParticle.C
 sed -i s/solid/injectorSolid/g injectorSolidParticle.H
sed -i s/solid/injectorSolid/g injectorSolidParticleI.H
 sed -i s/solid/injectorSolid/g injectorSolidParticleIO.C
sed -i s/solid/injectorSolid/g injectorSolidParticleCloud.C
 sed -i s/solid/injectorSolid/g injectorSolidParticleCloud.H
sed -i s/solid/injectorSolid/g injectorSolidParticleCloudI.H
 sed -i s/simpleFoam/injectorSolidParticleFoam/g injectorSolidParticleFoam.C
```

The last row above is not neccesary, but it is done so that it is easy to see in the beginning of the file that the solver is `injectorSolidParticleFoam` and not `simpleFoam`. For correct compiling, the files in `/Make` needs to be changed. Edit `Make/files` with whatever text editor you like. Make sure it looks like:

```
injectorSolidParticle.C
injectorSolidParticleIO.C
injectorSolidParticleCloud.C
injectorSolidParticleFoam.C

EXE = $(FOAM_USER_APPBIN)/injectorSolidParticleFoam
```

Also change `Make/options` to

```
EXE_INC = \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/incompressible/RAS/RASModel \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/finiteVolume/lnInclude  \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude

EXE_LIBS = \
    -lincompressibleRASModels \
    -lincompressibleTransportModels \
    -lfiniteVolume  \
    -llagrangian \
    -lincompressibleTurbulenceModel
```

The settings in `files` are needed to define which files that are to be compiled and which command the new class will be called with. In `options`, models for solving both the flow and the particle tracks are included. The new class is now setup for the compilation. But before doing that, changes to implement the particle injector are needed.

3.1.2 injectorSolidParticleFoam.C

In the beginning of this file, among the include statements, make sure to include

```
#include "injectorSolidParticleCloud.H"
```

This is done so that the solver can call functions of `injectorSolidParticleCloud.H`, which will be modified in the following section. Also, after `#include "initContinuityErrs.H"`, state

```
#include "readGravitationalAcceleration.H"
```

This is done since the gravitational acceleration, g needs to be read when solving the particle tracks. Directly before `Info<< "\nStarting time loop\n" << endl;` insert

```
injectorSolidParticleCloud particles(mesh);
```

so that the solver knows that particles are being included in the flow. To update the movement of particles, just before `runTime.write();`, add

```
particles.move(g);
```

The first three of the above statements need to be included before this last statement. This is due to that these make sure that everything needed for solving the particle tracks is included. The calculation of all particle positions is then done at `particles.move(g);`.

Save and close the file. The solver should now work as desired, but the injection of the particles has not yet been defined.

3.1.3 injectorSolidParticleCloud.C

For specification of the injection process, the following code is needed. This can be included anywhere inside the Member function `void Foam::solidParticleCloud::move(const dimensionedVector& g)`, where the updating of particle positions is done. So to be sure to get it right, put it at the top, just after the first curly bracket.[3]

```
// Injector 1
//Set injection position (z=0 if 2d)
scalar posy=0.015;
scalar posz=0;
scalar posx=-0.0203;
vector pos = vector(posx,posy,posz);
//Set initial velocity vector
vector vel=vector(0,0,0);
//Particle diameter
scalar d = 1e-3;
// Find cell at specified injection position and add particle here
label cellI=mesh_.findCell(pos);
if(cellI>=0) {
injectorSolidParticle* ptr= new injectorSolidParticle(*this,pos,cellI,d,vel);
Cloud<injectorSolidParticle>::addParticle(ptr);
}
```

As can be seen, properties like diameter, initial velocity and initial position of the injected particles are specified here. The label `cellI` finds a cell located at the injection position. Then, in the *if statement*, a particle with the specified diameter and velocity is added in this cell, at the injection position. Save and close.

3.1.4 createFields.H

The solver injectorSolidParticleFoam will need to read the density of the fluid the particles are introduced in. To be able to do this the following should be added at the top of `createFields.H`.

```
 Info<< "\nReading transportProperties\n" << endl;
    IOdictionary transportProperties
    (
        IOobject
        (
            "transportProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE,
            false
        )
    );

dimensionedScalar rhotP(transportProperties.lookup("rho"));

volScalarField rho
                (
                    IOobject
                    (
                        "rho",
                        runTime.timeName(),
```

```
                    mesh,
                    IOobject::NO_READ,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("rho", rhotP.dimensions(), rhotP.value())
            )
;
```

This is the final modification before compilation, save and close the file. Make sure to be in the directory of the `.C` files that are to be compiled and do

```
wclean
wmake
```

3.2 Example case

The new class will now be implemented on the pitzDaily case. This case is appropriate since it is possible to, in simple way demonstrate how the particles are influenced by the fluid flow. The domain of the pitzDaily case can be seen in figure 3.1. [5]

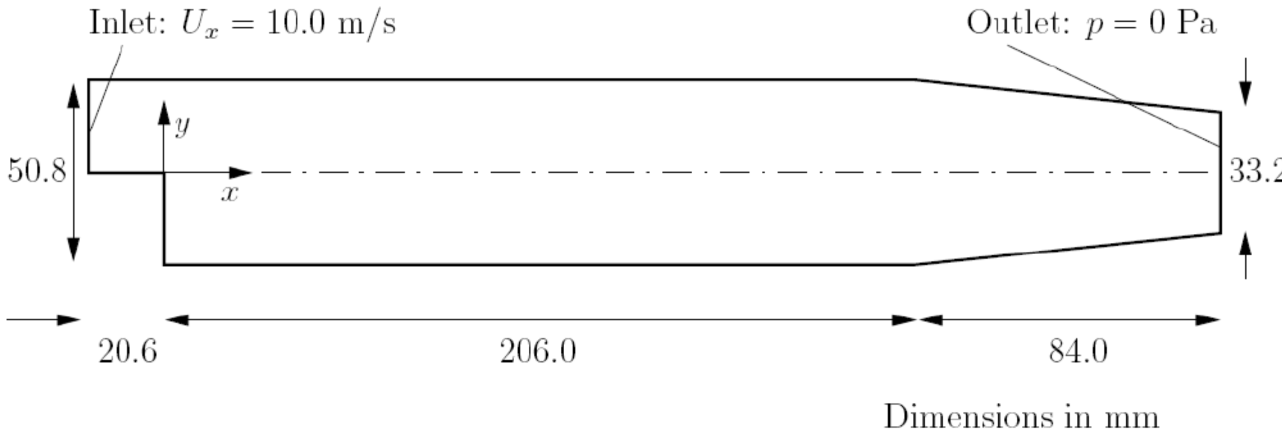

Figure 3.1: Geometry of the pitzDaily tutorial case.

Start by copying and renaming the case.

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily ./pitzDailyInject
cd pitzDailyInject
```

In this example, the flow will be solved to steady state before the particles are introduced into the flow. This technique is commonly used in CFD-simulations, even though there normally are more sophisticated models for particle-particle, particle-fluid and particle-wall interaction. So, to solve the flow do

```
blockMesh
simpleFoam
```

This will take a couple of minutes. Meanwhile, locate the attached folder `files` which include boundary conditions and physical properties for particles in accordance with the solidParticle class. Since the particle injector will be used here, the conditions are set up so that no additional particles are injected at the simulation start. When `simpleFoam` has solved the flow, the case directory have several time directorys, with 1000 as the latest. Do the following.

- Copy the folder `lagrangian` to `./1000`
- Copy the file `particleProperties` to `./constant`
- Copy the file `g` to `.constant`.

Density is in `particleProperties` set to $1.2\,kg/m^3$ and in `g` the gravity to $9.81\,m/s^2$. Then edit `system/controlDict` and change the following parameters as

```
startTime       1000;
endTime         1000.15;
deltaT          0.003;
writeInterval   1;
timePrecision   8;
```

This means that the simulation only will run for additionaly 0.15 seconds and data for every time step of $0.003\,s$ will be saved. The time precision is changed, so that all numbers of these time steps will be included when being saved.

Finally, for setting the fluids density, edit `/constant/transportProperties` and after `nu nu [ 0 2 -1 0 0 0 0 ] 1e-05;` insert

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

This density value is what is read in the code inserted in `createFields.H` in Section 3.1.4. Necessary settings for the simulation are now set. Run the simulation, convert it and postprocess by

```
injectorSolidParticleFoam
foamToVTK
paraview
```

In paraview click "Open" and open `pitzDailyInject.vtk` in the `/VTK` directory. Click "Apply" and choose to display the domain with velocity, U and as "Surface". See below.

Again, click "Open", browse into `/lagrangian/defaultCloud` and open `defaultCloud.vtk`. Click "Apply", go to the last time step with and rescale data to range with . By doing this, it is ensured that the particle data can be reached in paraview since they are introduced in the flow at after about twenty time steps. On `defaultCloud` insert a "Glyph" filter by marking it and clicking . Choose "d" for "Scalars", "Sphere" for "Glyph Type", "Scale Mode" as "off", enable edit of "Set Scale Factor" and set it to 0.005. Click "Apply" and . The parcels are injected as specified and their path through the domain can be seen in figure 3.2.

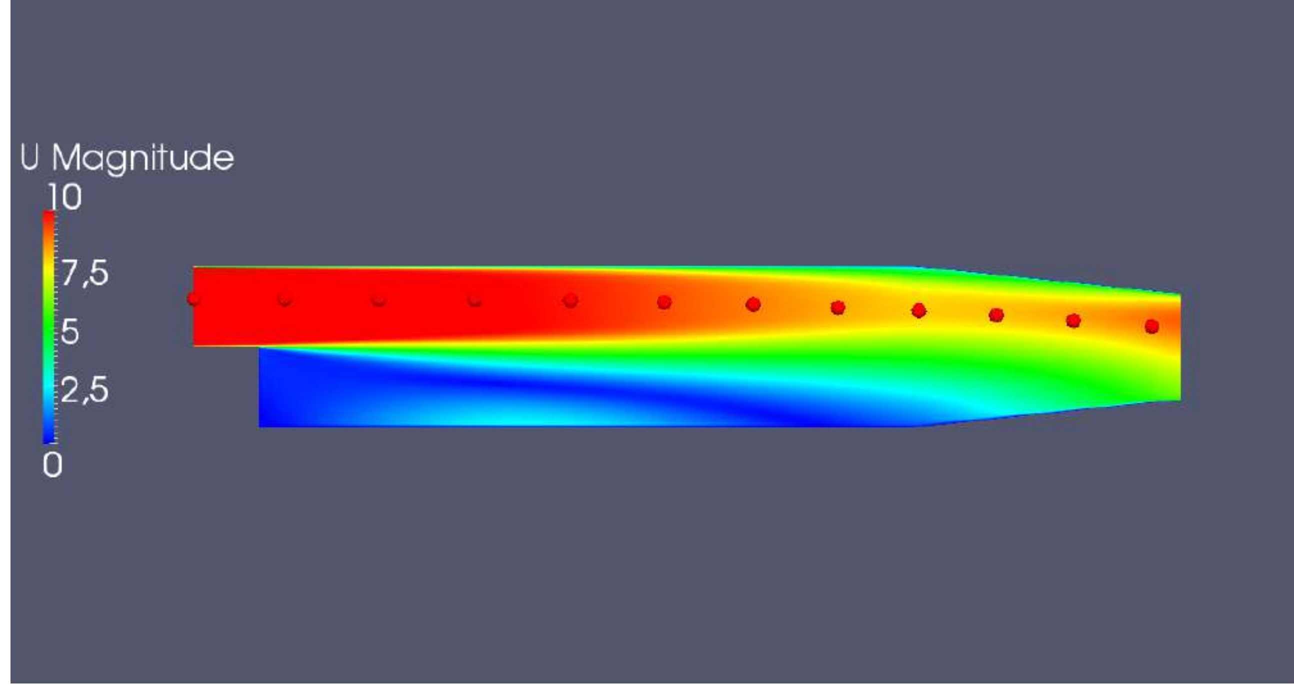

Figure 3.2: Solid particles in the case domain after the last time step.

3.3 Two injectors

With a slight modification of the setup in previous section, a second injector can be implemented on the pitzDaily case. Go back to the injectorSolidParticleFoam class directory.

```
cd $WM_PROJECT_USER_DIR/applications/solvers/injectorSolidParticleFoam
```

Edit the file `injectorSolidParticleCloud.C` and after the previous specification of the first injector, add

```
// Injector 2
//Set injection position (z=0 if 2d)
scalar posx2=0.15;
scalar posz2=0;
scalar posy2=-0.0243;
vector pos2 = vector(posx2,posy2,posz2);
//Set initial velocity vector
vector vel2=vector(0,0,0);
//Particle diameter
scalar d2 = 1e-3;
// Find cell at specified injection position and add particle here
label cellI2=mesh_.findCell(pos);
if(cellI2>=0) {
injectorSolidParticle* ptr= new injectorSolidParticle(*this,pos2,cellI2,d2,vel2);
Cloud<injectorSolidParticle>::addParticle(ptr);
}
```

This injector will be placed at the lower wall of the domain. Save and close, clean and compile (`wclean` and `wmake`) again.

Go to the case directory again.

```
cd $FOAM_RUN/pitzDailyInject
```

Previous knowledge tells that in order to be able to follow the entire track of the particles injected at the second injector a longer runtime is needed. Edit `.system/controlDict` and change the end time to

```
endTime          1000.42;
```

Before rerunning the case, delete the previous particle track results.

```
rm -rf 1000.*
rm -rf VTK
```

Solve with the recompiled solver, convert and postprocess as in Sect. 3.2. The result shows that the second injector works and that its particles are injected into the recirculation region of the domain. Towards the end of the simulation, it can though be seen that the particles tears away from this region and flow out of the domain. It can also be seen that the behaviour of the particles is not totally steady. Reasons for this could be the nature of the flow and/or how the particle behaviour is modeled. To analyze this further is however outside the scope of this tutorial. The result after the last time step can be seen in figure 3.3

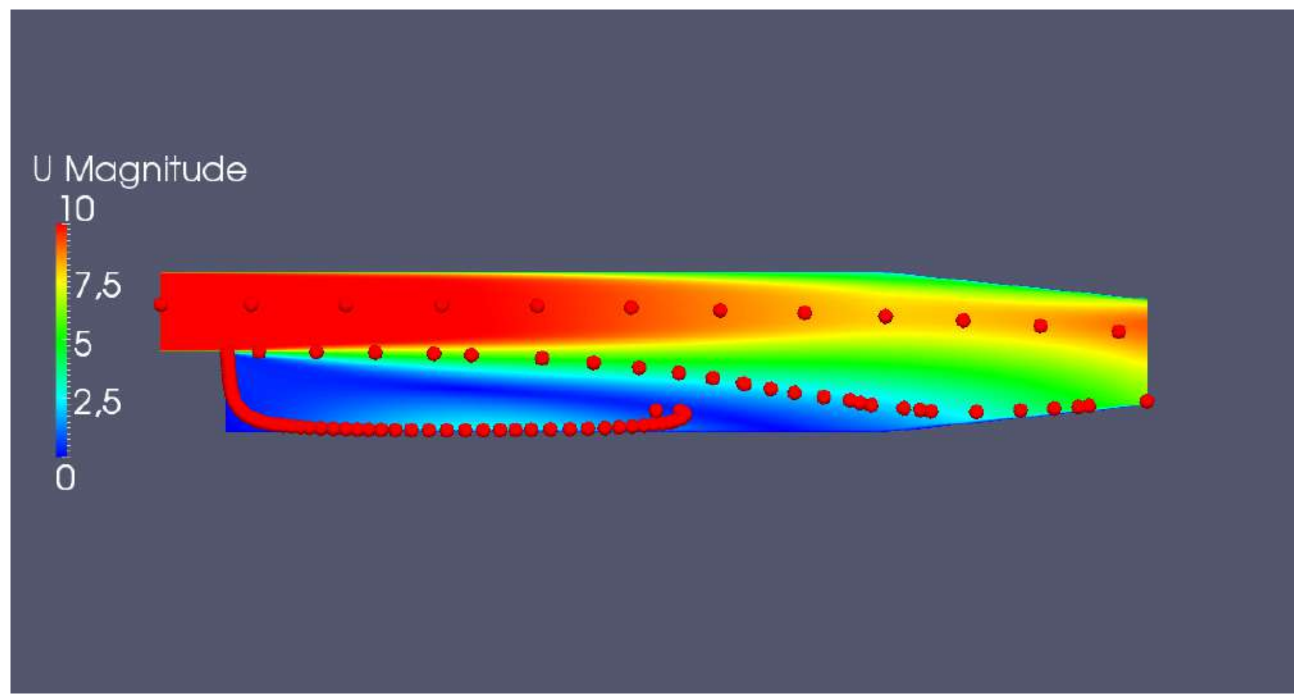

Figure 3.3: Solid particles in the case domain after the last time step. Two injectors used

3.4 Evenly distributed injection

The injector implemented in previous sections can easily be modified so that a collection of particles with a constant spacing is introduced. Go back to the directory of `injectorSolidParticle`.

```
cd $WM_PROJECT_USER_DIR/applications/solvers/injectorSolidParticleFoam
```

Up to now there has only been one particle injected per time step and injector. So, to inject several particles per time step it should be sufficient to repeat the injection process as many times as the desired number of particles is. This can be achieved by a for loop. Open `injectorSolidParticleCloud.C` and scroll down to the section where the injector is defined. Remove entirely the second injector and before `//Injector 1` insert

```
for (label i=0; i<=25; i++) {
```

To end the loop, don't forget to put a } directly after the end of the "if" statement at the end of the injector. With this setup, 25 particles will be injected per time step. But to make them spread over a certain area, an additionaly modification is needed. Change the y-pos to

```
scalar posy=-0.0248+0.002*i;
```

Also set the x-pos as

```
scalar posx=0.1;
```

Save and close. The injector will now be placed in the widest area of the domain and inject 25 particles with a spacing of 0.002 m. Compile the class, clear previous results, run and postprocess the case as earlier. Results are seen in figure 3.4

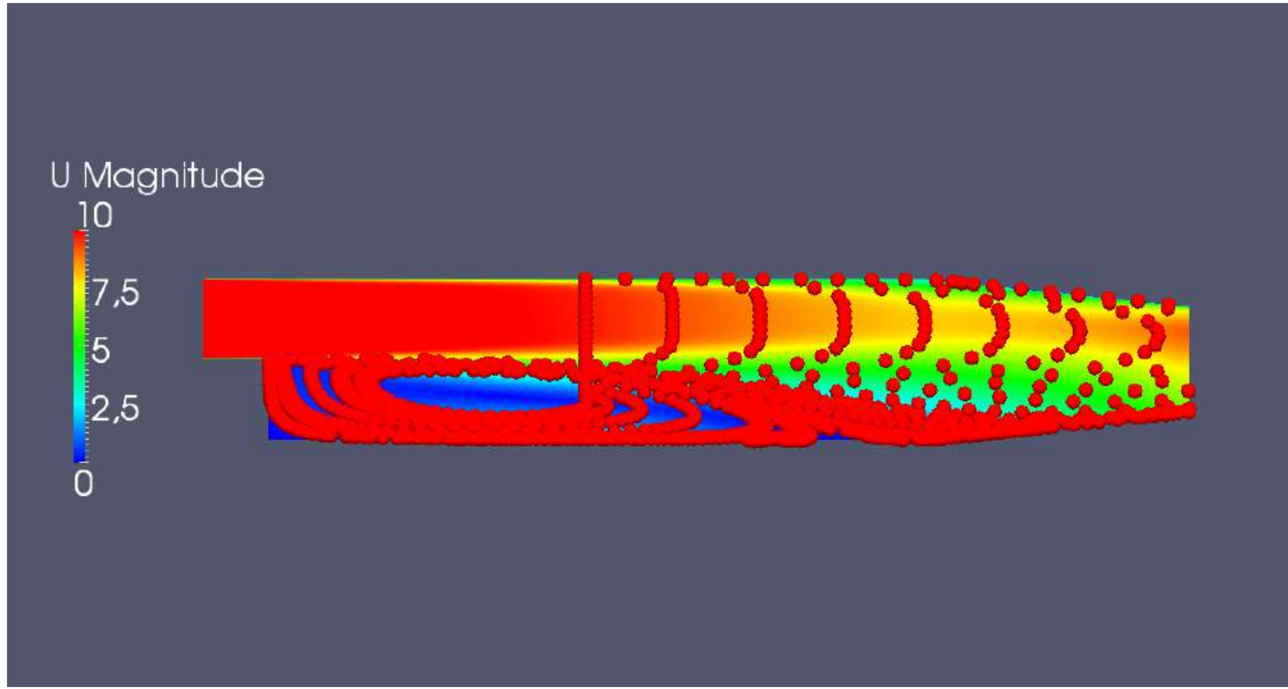

Figure 3.4: Solid particles in the case domain after the last time step. Evenly distributed injector used

Bibliography

[1] Per Carlsson *Tutorial dieselFoam*. Gothenburg, Sweden 2009.

[2] http://en.wikipedia.org/wiki/Weibull_distribution

[3] Aurélia Vallier *Tutorial icoLagrangianFoam/solidParticle* Gothenburg, Sweden 2010

[4] http://openfoamwiki.net/index.php?title=Special:Search&ns0=1&redirs=1&search=injector&limit=100&offset=0

[5] foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf