Cite as: Hansson, J.: Introducing a hybrid rebound and sticking particle-wall interaction model. In Proceedings of CFD with OpenSource Software, 2023, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2023

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Introducing a hybrid rebound and sticking particle-wall interaction model

Developed for OpenFOAM-v2112

Author:
Johannes Hansson
Chalmers University of Technology
johanneh@chalmers.se

Peer reviewed by:
Henrik Ström
Wei Chen
Saeed Salehi

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 14, 2024

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How to perform Lagrangian particle tracking in OpenFOAM using the `icoUncoupledKinematicCloud` function object.
- How to use the standard particle-wall interaction models rebound (with constant coefficients for elasticity and restitution), stick and escape.

The theory of it:

- The basics of Lagrangian particle tracking.
- How inertia affects particle trajectories.
- What levels of particle-fluid coupling are available and when they can be used.
- The basics of elastic and inelastic collisions.
- How expected particle behavior, either rebound or stick, depends on particle- and local carrier flow properties, according to a chosen collision model.

How it is implemented:

- How the user-selected particle behavior, rebound or stick, is handled once a collision has been detected.

How to modify it:

- How to add new particle-wall collision models and types.
- How to implement a hybrid particle-wall interaction model that uses the predictions of a chosen collision model to predict rebound or stick dynamically.

Prerequisites

This tutorial is intended for OpenFOAM users that have some experience with using the software. Therefore, the report assumes that the reader is familiar with the following:

- Usage of Linux command line tools and bash scripting.
- Basic experience of running OpenFOAM tutorial cases and making simple modifications to the case dictionaries.
- Usage of ParaView, for post-processing of simulation results.
- Experience in compiling OpenFOAM code is beneficial, but not required.

Contents

Chapter 1

Theoretical background

1.1 Lagrangian particle tracking

When simulating particle-laden flows, the fluid and particles are often treated as two separate frames of reference. The carrier fluid is generally treated according to the Eulerian description, whereas the particles are treated according to the Lagrangian description. In this work we focus on the Lagrangian description of the particles and assume that the Eulerian description of the fluid is known to the reader. Additionally, we limit the discussion to inert, solid, particles. Bubbles and similar particles are outside the scope of this work.

In the Lagrangian description, particles are treated as discrete points that may or may not have an associated volume. Small particles, when compared to the spatial scales of the carrier fluid, are often modeled without any volume effects. This assumption makes simulations easier to implement and computationally cheaper. The assumption is, for small particles, associated with only negligible modeling errors. Comparatively large particles, on the other hand, need to be modeled with non-zero volume since the carrier fluid properties are not constant over the surface of the particle. A more complex and computationally expensive model is then needed to accurately model the behavior of larger particles. In this work we consider only comparatively small particles, so the zero-volume modeling approximation is used throughout this report.

The separate description of Eulerian fluid and Lagrangian particles also allows us to consider the effects of particle inertia on the particle tracks. For very light, neutrally buoyant, particles it is possible to use an Eulerian description of the particle transport mechanisms. The description of their collective motion is then given by the evolution of a particle concentration field. However, when we have particles with finite inertia or some degree of buoyancy, then we need a different description to capture their behavior.

In the Lagrangian model for the particles we instead solve Newton's equation of motion given a set of forces acting on each individual particle, instead of on a packet of a particle concentration field, as is the case in the Eulerian description. From Newton's second law of motion we know that

$$m\mathbf{a} = \sum_i \mathbf{F}_i, \tag{1.1}$$

where m is the particle mass, $\mathbf{a}$ is the acceleration vector and $\mathbf{F}_i$ is force vector of force number i. The forces are often divided into two categories depending on their origin, fluid-particle interaction forces and ancillary forces. The fluid-particle forces include, for example, drag and pressure gradient forces. Ancillary forces represent forces that do not directly come from the fluid, such as gravitational and electrostatic forces. Using this categorization, the force summation in Eq. (1.1) can then be expressed as

$$m\mathbf{a} = \sum_i \mathbf{F}_{\text{fluid}-\text{particle},i} + \sum_j \mathbf{F}_{\text{ancillary},j}. \tag{1.2}$$

From a modeling perspective, the fluid-particle forces are surface forces that must be integrated over the surface of the particle to quantify their contribution to the particle track. The ancillary forces

are, on the other hand, typically body forces that act over the entire particle volume. Expanding the summation signs in Eq. (1.2) with, for example, the forces mentioned above, we can rewrite Eq. (1.1) as

$$m\mathbf{a} = \mathbf{F}_{\text{drag}} + \mathbf{F}_{\text{lift}} + \mathbf{F}_{\text{pressure gradient}} + \mathbf{F}_{\text{gravity}} + \mathbf{F}_{\text{electrostatic}} + \dots \tag{1.3}$$

Do note, however, that it is important to evaluate which forces are relevant in each particular case. It could be that a particular system needs a description of, for example, the lift force, but not the electrostatic forces, and Eq. (1.3) would need to be modified accordingly.

As an example, we can describe the drag force on a sphere by the surrounding fluid by using the commonly used model given by [1, 2, 3]

$$\mathbf{F}_{\text{drag}} = -\frac{3}{4}\frac{\mu_{\text{f}} C_{\text{D}} \text{Re}_{\text{p}}}{\rho_{\text{p}} d_{\text{p}}^2}\frac{\mathbf{u}_{\text{rel}}}{|\mathbf{u}_{\text{rel}}|}, \tag{1.4}$$

where we have dynamic viscosity of the carrier fluid μ_{f}, particle drag coefficient C_{D}, particle Reynolds number Re_{p}, particle density ρ_{p}, particle diameter d_{p} and relative velocity $\mathbf{u}_{\text{rel}} = \mathbf{u}_{\text{p}} - \mathbf{u}_{\text{f}}$ between the particle $\mathbf{u}_{\text{p}}$ and the fluid $\mathbf{u}_{\text{f}}$. In the case of a sphere, the drag coefficient can be obtained using [1, 2, 3]

$$C_{\text{D}} = \begin{cases} \dfrac{24}{\text{Re}_{\text{p}}}\left(1 + \dfrac{1}{6}\text{Re}_{\text{p}}^{2/3}\right) & \text{if } \text{Re}_{\text{p}} \leq 1000 \\ 0.424 & \text{if } \text{Re}_{\text{p}} > 1000 \end{cases} \tag{1.5}$$

and the particle Reynolds number is given by [1, 2, 3]

$$\text{Re}_{\text{p}} = \frac{\rho_{\text{f}} |\mathbf{u}_{\text{rel}}| d_{\text{p}}}{\mu_{\text{f}}}, \tag{1.6}$$

where ρ_{f} is the fluid density. This is the drag model implemented in the commonly used `sphereDrag` force model in OpenFOAM.

1.2 Effect of inertia on particle trajectory

The typical choice of using the Lagrangian framework for modeling the particles is due to how particle inertia can be modeled. Heavy, high-density particles tend to follow trajectories that are not affected much by the surrounding fluid, almost like a bullet. On the other hand, lighter, neutrally buoyant particles tend to follow the surrounding flow very closely, which is the typical behavior of often used tracer particles. Light particles can be approximately modeled using an Eulerian framework, but heavier particles are not as easy to model in this description. On the other hand, by using the Lagrangian framework particle inertia is always included in the description, accurately describing both light and heavy particles.

The effects of inertia on particle behavior in the fluid is often described by the so-called Stokes number, which is defined by

$$\text{St} = \frac{t_0 u_0}{l_0}, \tag{1.7}$$

where t_0 is the relaxation time of the particle, u_0 is the fluid velocity and l_0 is a characteristic length scale, often of an obstacle. The relaxation time is given, under the assumption of Stokes flow, by

$$t_0 = \frac{\rho_{\text{p}} d_{\text{p}}^2}{18\mu_{\text{f}}}. \tag{1.8}$$

Stokes flow represents the flow cases where the particle Reynolds number is $\text{Re}_{\text{p}} << 1$. A low Stokes number represents a light particle that reacts quickly to changes in the fluid. In the equations above it is seen as a small relaxation time t_0. A high Stokes number, on the other hand, represents a heavy particle with a comparatively large relaxation time, which means the particle reacts slowly to changes in the fluid. The Stokes number can then be used as a measure of inertial effects on particles suspended in a fluid. Previous works have, for example, studied particle deposition on a cylinder as a function of Stokes number [4].

1.3 Fluid-particle coupling

In the typical case of Lagrangian particle tracking we have two or more phases, which means that we must consider how the different phases couple to each other. If we consider Lagrangian particles in an Eulerian carrier fluid, then we have three categories of interactions between the fluid and the particles.

The simplest type of interaction, one-way coupling, represents the case where the carrier fluid affects the particles, but the particles do not affect the fluid or each other. This description is well-suited for flows in which the particle volume fraction is very low, so called dilute flows. These flows have so few particles that their effect on turbulence is negligible, so the fluid is not affected by their presence. The particles are also so well-dispersed that they rarely collide with each other. This is the easiest type of interaction to model, from a computational perspective.

If the particle volume fraction is a bit higher, then we also need to consider the effect particles have on the carrier fluid. This is referred to as two-way coupling, where the fluid affects the particles and the particles affect the fluid. However, the particles are still so well-dispersed that they only rarely collide. This means that particle-particle interaction is not an important factor and can be ignored. Two-way coupling is slightly more complex to model than one-way coupling.

Finally, if the particle volume fraction is high enough we also need to model the effects of particle-particle collisions, in addition to the fluid-particle and particle-fluid couplings. This happens for dense flows where the particles make up a significant fraction of the total system volume. This type of coupling is called four-way coupling and represents a sharp increase in computational cost as opposed to one- and two-way coupling.

In this work we are mainly interested in particle-wall collisions, so to simplify our computational setup we only consider one-way coupling. This coupling method provides enough accuracy for the dilute flows considered. It is important to note that using four-way coupling would not change any results for this type of flow, except that the computational cost would go up.

1.4 Types of collisions

There are generally two types of collisions, elastic and inelastic collisions. Elastic ones are collisions in which both momentum and kinetic energy is conserved. In inelastic collisions, however, only momentum is conserved. Some degree of kinetic energy is lost as heat due to internal friction. The degree of inelasticity is typically measured using the coefficient of restitution e. This variable represents the change in speed just before and just after the collision, i.e. if we have a collision between a small particle and a solid wall then

$$|v_{\text{after}}| = e|v_{\text{before}}|, \tag{1.9}$$

where v_{before} and v_{after} represent the particle velocities just before and after the collision. A coefficient of restitution $e = 1$ represents a collision in which no kinetic energy is lost in the process. This is also known as a perfectly elastic collision. For $0 < e < 1$ we have an inelastic collision. This is the typical kind of collision found in real life. Some, but not all, of the velocity is lost in this interaction. Finally, we have $e = 0$ which represents a perfectly inelastic collision. In this case the two objects stick together after the collision. Worth noticing is that the coefficient of restitution is not necessarily constant for all collisions between the same two objects. The coefficient value could, for example, depend on the collisional velocity or angle.

In the above discussion, the collision is assumed to be head-on, i.e. with no sideways velocity component. For clarity we could then rewrite Eq. (1.9) as

$$|v_{\text{n,after}}| = e|v_{\text{n,before}}|, \tag{1.10}$$

where v_{n} represent normal velocities. In reality, such a component is typically present. We can then use e to denote the coefficient of restitution in the normal direction and μ as the friction coefficient in the tangential direction. μ then represents the fraction of kinetic energy in the tangential direction

that is lost in a collision. Similarly to Eq. (1.10) we get

$$|v_{\mathrm{t,after}}| = (1 - \mu)|v_{\mathrm{t,before}}|, \tag{1.11}$$

where v_{t} represent tangential velocities. A value of $\mu = 0$ represents a collision in which no tangential kinetic energy is lost.

1.5 Details of the implemented collision model

The method outlined in this tutorial works for many types of instantaneous particle-wall collision models, but as an example we implement a collision model proposed by Eidevåg et al. [5]. This model is tuned for use with ice particles impacting on bluff bodies used in research for the automotive industry. The methodology presented in this report is, nevertheless, general in the sense that other models, such as for ash particles [6], can be implemented using a similar strategy.

Eidevåg et al.'s collision model consists of two major parts, an adhesion model and a resuspension criterion. The adhesion model calculates if a certain collision will lead to particle sticking or rebound with a certain coefficient of restitution. The resuspension criterion checks if the drag force exerted on a deposited particle is large enough to allow it to be resuspended into the flow. If the criterion predicts that the particle will be resuspended then the particle will not be allowed to stick and a perfectly elastic collision is instead triggered. This behavior may not be strictly correct since a real particle that is resuspended loses all information about original incidence angle and velocity, whereas this model keeps this information in the perfect collision. However, this behavior is used in the article by Eidevåg et al. [5], so we use it here as well for consistency.

A schematic illustration of the Eidevåg et al. collision model is presented in Figure 1.1. Here we see the main calculation steps in the model and which conditions are checked to decide which collision behavior the model should trigger. Further details about the model are presented in Sections 1.5.1 and 1.5.2.

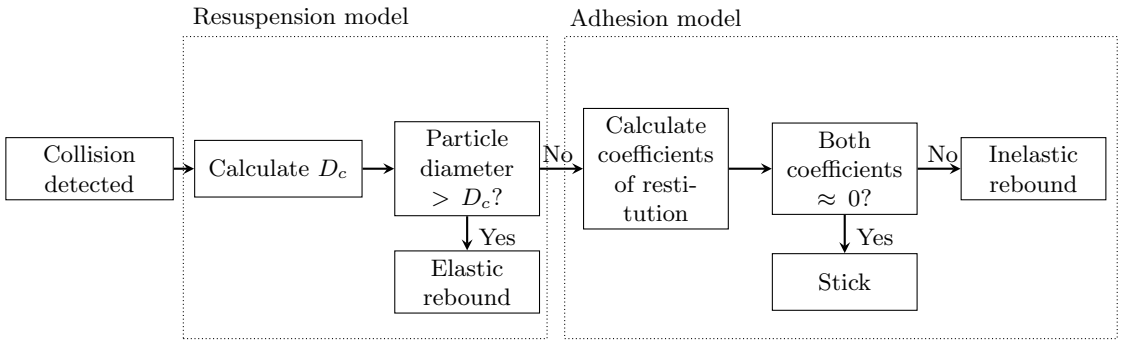

Figure 1.1: A schematic illustration of the Eidevåg et al. collision model. It is composed of two sub-models, the resuspension model and the adhesion model.

1.5.1 Adhesion model

When a particle approaches a wall, the particle experiences attractive van der Waals forces from the wall [5]. Once the collision has happened, the kinetic energy of the particle determines if the particle will stick to the surface or if it will be resuspended into the flow. If the kinetic energy of the particle is not enough to overcome the attractive forces, then it will stick to the surface. If not, then it will rebound. The forces that are active in the sticking process can be modeled using the Johnson-Kendall-Roberts (JKR) model [7, 8]. This model takes into account both the elastic response of the particle and wall, as well as the attractive van der Waals forces between the objects. Eidevåg et al. [5] use a simplified implementation of this model based around two critical velocities, the normal stick velocity $V_{\mathrm{c,n}}$ and the tangential stick velocity $V_{\mathrm{c,t}}$. If the particle impact velocity

V_i is smaller than either of these critical velocities, then the corresponding velocity component is eliminated. If the impact velocity is smaller than both velocities, then the particle sticks to the surface.

In general, the coefficients of restitution for a particle-wall collision are in this model given by

$$e_\mathrm{n} = e_\mathrm{qe}\sqrt{1 - \left(\frac{V_\mathrm{c,n}}{\max\left(V_\mathrm{i}, V_\mathrm{c,n}\right)}\right)^2} \tag{1.12}$$

for the wall-normal direction and

$$e_\mathrm{t} = 1 - \mu = e_\mathrm{qe}\sqrt{1 - \left(\frac{V_\mathrm{c,t}}{\max\left(V_\mathrm{i}, V_\mathrm{c,t}\right)}\right)^2} \tag{1.13}$$

for the tangential direction. The variable $e_\mathrm{qe} = \sqrt{1-\xi} = \sqrt{1-0.15} \approx 0.92$ represents the average coefficient of restitution for head-on collisions between spheres and a solid wall, both made of ice. These collisions were in the quasi-elastic regime, i.e. for collisions where $V_\mathrm{i} < V_\mathrm{c,n}$. ξ represents the fraction of kinetic energy that is lost by causing mechanical damage at the interface between the two colliding objects [9, 10].

The critical velocities above are derived from the JKR model. Here, the particle kinetic energy needed for a complete rebound in the normal direction, E_s, is defined by

$$E_\mathrm{s} = \frac{3K_1\pi a_0^2 W}{4(6)^{1/3}}, \tag{1.14}$$

where $K_1 \approx 0.9355$ is a numerical integration constant [5, 11]. The variable W represents the work of adhesion and a_0 is the equilibrium contact radius given by [5, 11]

$$a_0 = \left(\frac{9\pi W R^*}{2E^*}\right)^{\frac{1}{3}}, \tag{1.15}$$

in which E^* is the effective Young's modulus and R^* is the effective radius of contact, defined by

$$R^* = \frac{R_1 R_2}{R_1 + R_2}, \tag{1.16}$$

where R_1 and R_2 represent the radii of the two particles in a collision. In this case we assume that R_1 represents our colliding particle and R_2 represents the wall, which can be approximated with $R_2 \to \infty$. Using this assumption we get $R^* = R_1$. From these, Eidevåg et al. define the highest velocity for sticking in the normal direction as

$$V_\mathrm{s} = \sqrt{\frac{2E_\mathrm{s}}{m_\mathrm{p}}}, \tag{1.17}$$

with m_p representing the mass of the particle and E_s as defined in Eq. (1.14). For the tangential direction, Eidevåg proposes the equation

$$V_\mathrm{c,t} \approx 0.23\left(\frac{\Delta\gamma}{\gamma}\right)V_\mathrm{c,n}, \tag{1.18}$$

where $\frac{\Delta\gamma}{\gamma}$ is the adhesion hysteresis of rolling [5]. Numerical values for the parameters are chosen to represent material properties for ice-ice contacts, with the values $W = 0.218\,\mathrm{J/m^2}$ and $E^* = 5.4\,\mathrm{GPa}$ [5]. The numerical value $\frac{\Delta\gamma}{\gamma} \approx 1$ comes from micron-sized ice particles [5, 11].

1.5.2 Resuspension model

The adhesion model only takes properties of the particle and wall into account when deciding on whether to apply rebound or stick behavior for a certain collision. However, it is entirely possible that a particle may stick, but that the fluid will almost immediately make it roll and eventually detach from the surface, resuspending the particle into the flow. To also account for the effects of the fluid on depositing particles, a resuspension criterion is proposed by Eidevåg et al. [5]. The implementation of this resuspension criterion is based on a critical diameter D_c. Any colliding particle that has a diameter larger than this is not allowed to stick, irrespective of what the adhesion model predicts. The reason for this is that the fluid would immediately resuspend the particle due to the drag and lift forces from the fluid. These particles instead experience a perfectly elastic collision with the wall.

This critical diameter is derived from numerically solving the moment balance condition

$$F_{\mathrm{D},\parallel}\frac{d_\mathrm{p}}{2} + M_\parallel = 2.4F_{\mathrm{D},\parallel}\frac{d_\mathrm{p}}{2} \geq M_\mathrm{a}. \tag{1.19}$$

If the the inequality is true, then the particle is predicted to roll and eventually resuspend into the flow. In Eq. (1.19), $F_{\mathrm{D},\parallel}$ is the magnitude of the drag force on the particle parallel to the wall surface, the vector expression for which is given by

$$\mathbf{F}_{\mathrm{D},\parallel} = \frac{1}{2}\rho_\mathrm{f} C_{\mathrm{D},\parallel} A_\mathrm{p} |\mathbf{U}_\mathrm{b}| \mathbf{U}_\mathrm{b}, \tag{1.20}$$

where $C_{\mathrm{D},\parallel}$ represents the drag coefficient in the tangential direction for a particle that is deposited on a flat surface, A_p is the projected area of the particle and $\mathbf{U}_\mathrm{b}$ is the fluid velocity at a distance $\frac{d_p}{2}$ from the wall surface. In Eq. (1.19) we also have the torque from non-uniformity in the flow $M_\parallel$, approximated by the expression

$$M_\parallel = 1.4F_{\mathrm{D},\parallel}\frac{d_\mathrm{p}}{2}, \tag{1.21}$$

and the adhesive resistance against starting a roll, M_a, which can be expressed as

$$M_\mathrm{a} = \frac{1}{4}\pi W d_\mathrm{p} a_0 \frac{\Delta\gamma}{\gamma}. \tag{1.22}$$

Solving Eq. (1.19) requires us to know the value of U_b. Eidevåg et al. use the expression

$$u^+ = \frac{U_\mathrm{b}}{u_\tau}, \tag{1.23}$$

where u^+ is a normalized velocity and u_τ is the friction velocity. Here, u^+ is not known directly either so Eidevåg et al. [5] use the Reichardt profile [12, 13], given by

$$u^+ = \frac{1}{\kappa}\ln\left(1+\kappa y^+\right) + 7.8\left(1 - \exp\left(\frac{-y^+}{11}\right) - \frac{-y^+}{11}\exp\left(\frac{-y^+}{3}\right)\right), \tag{1.24}$$

to calculate u^+ given the known values of dimensionless wall distance y^+. In the equation, $\kappa \approx 0.41$ is the von Kármán constant. Eidevåg et al. [5] use the expression

$$y^+ = \frac{\rho_\mathrm{f} d_\mathrm{p} u_\tau}{2\mu_\mathrm{f}} \tag{1.25}$$

for this wall distance. Using Eq. (1.23), together with Eq. (1.24) and (1.25), we can calculate U_b from u_τ, which can be calculated from the wall shear stress according to [14]

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}}. \tag{1.26}$$

Wall shear stress is directly computable in the simulations, so we can now solve for the critical diameter in Eq. (1.19).

Eidevåg et al. solved Eq. (1.19) numerically for a set of u_τ values and observed that the solution for the critical diameter D_c, to a good approximation, can be approximated by the fitting function

$$D_\mathrm{c} = a u_\tau^b, \tag{1.27}$$

where $a = 77.7\,\mu\mathrm{m}$ and $b = -1.34$ are fitting parameters obtained by least squares fitting. When using the model, Eidevåg et al. replace u_τ with $\hat{u}_\tau = \overline{u}_\tau + 3\sigma_{u_\tau}$, where $\overline{u}_\tau$ is the average u_τ and σ_{u_τ} is the standard deviation in u_τ.

Chapter 2

Lagrangian particle tracking in OpenFOAM

This chapter describes how the existing OpenFOAM particle tracking implementation works. It starts with a description of the `icoUncoupledKinematicCloud` function object and the files needed for configuring the solver. Then, the existing particle-wall collision models are presented and discussed together with details of their implementation. Finally, an example of particle tracking solver output is presented and discussed.

2.1 Basic particle tracking setup

One of the most flexible ways of running incompressible, one-way coupled Lagrangian particle tracking is using the `icoUncoupledKinematicCloud` function object [15]. This function object can be coupled with any single-phase, incompressible, transient solver to track spherical particles at the same time as the field is being solved. This makes the tracking more useful than simple tracer particles that can be added in post-processing software such as ParaView since this simultaneous tracking can include the effects of particle inertia. It can also track the particles accurately without needing to save many flow field snapshots, thereby reducing storage space requirements.

To use this function object we need to do three things:

1. Declare the function object in the `system/controlDict` file
2. Declare particle tracking settings in the `constant/kinematicCloudProperties` file
3. Declare settings for gravity in the `constant/g` file

The first requirement is fulfilled by declaring a function object, in this case named `tracks`, in the functions subdictionary of the `controlDict` file:

`system/controlDict`

```
//controlDict contents...

functions
{
    tracks
    {
        type    icoUncoupledKinematicCloud;
        libs    (lagrangianFunctionObjects);
    }
}
```

For the second requirement we need to create a `kinematicCloudProperties` file similar to the excerpt presented below. A complete example of the `kinematicCloudProperties` file is presented in

appendix A. This dictionary specifies many aspects of the particle tracking, such as interpolation methods, number of particles to track, where they are injected, as well as particle diameters, densities, etc. In this report we are mainly concerned with how particle-wall collisions are implemented and used, so we will place particular focus on the line `patchInteractionModel localInteraction;` and the subdictionary `localInteractionCoeffs`. Note that ... represents a shorthand for content not shown.

constant/kinematicCloudProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      kinematicCloudProperties;
}

solution
{
    active          true;
    coupled         false;
    transient       yes;
    cellValueSourceCorrection off;
    //...

}

//...

subModels
{
    //...

    patchInteractionModel localInteraction;

    localInteractionCoeffs
    {
        patches
        (
            //...

            "(cylinder)"
            {
                type stick;
            }

        );
    }

    //...

}
```

Similarly, for the third requirement we define the file `constant/g` as

constant/g

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
```

```
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -2 0 0 0 0];
value           (0 0 -9.81);


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

This is all the required setup to add particle tracking to many of the transient CFD solvers in OpenFOAM. The rest of this report will deal with how to customize the particle tracking `intermediate` library so that it can handle hybrid particle-wall collision behaviors.

2.2 Particle-wall interaction models

At the time of writing, OpenFOAM only supports a small set of particle-wall interaction models [16]. The currently implemented models are

- Rebound
- Stick
- Escape

The rebound condition represents a generally inelastic collision between a particle and a wall. The degree of inelasticity is controlled by two parameters, `e` and `mu`, in the `localInteractionCoeffs` subdictionary of the `kinematicCloudProperties` file. `e` represents the normal restitution coefficient and `mu` represents the tangential friction, see Section 1.4. Perfectly elastic collisions can be realized by setting the normal restitution coefficient `e` to one and the tangential friction `mu` to zero. When using the rebound condition no particle will ever stick to the surface.

The stick condition represents unconditional sticking. If a particle impacts a surface, then it will always stick to the surface and stay there forever. The particle cannot be resuspended into the flow. This model does not need any extra parameters, in contrast to the rebound model.

Finally, the escape model just removes particles from the domain. It is often used at outlets or other similar interfaces from which the particles cannot return. Separate models can be assigned to each patch in the simulation domain, so it is possible to have, for example, an escape model at the outlet and a stick model on the surface of some submerged object.

2.2.1 Implementation details of the particle-wall interaction models

Once a collision has been detected, the chosen collision model for a given patch is used to determine how the particle will behave. The implementations of the particle-wall collision models are given in the file `$FOAM_SRC/lagrangian/intermediate/submodels/Kinematic/PatchInteractionModel/LocalInteraction/LocalInteraction.C`, as listed below. A full code listing of the collision models are given in appendix B. A shorter outline of the code structure is given below.

`LocalInteraction.C`

```
template<class CloudType>
bool Foam::LocalInteractionMix<CloudType>::correct
(
typename CloudType::parcelType& p,
const polyPatch& pp,
bool& keepParticle
)
{
```

```
    //initialization code...

    switch (it)
    {
        case PatchInteractionModel<CloudType>::itNone:
        {
            //...
        }
        case PatchInteractionModel<CloudType>::itEscape:
        {
            //...
        }
        case PatchInteractionModel<CloudType>::itStick:
        {
            //...
        }
        case PatchInteractionModel<CloudType>::itRebound:
        {
            //...
        }
        default:
        {
            //...
        }
    }
}
```

The listed function is responsible for choosing the collision model, based on user selection when setting up the OpenFOAM case, and then apply the desired behavior on the particles. Once a collision has been triggered, the `correct` function is run once for every particle that collides with a patch. Critically, it is the `swith` statement on line 240 that determines the impact behavior of this specific particle, based on the value of the `it` variable representing the selected collision model for this patch. The code execution path then jumps to one of the four main interaction cases, `itNone`, `itEscape`, `itStick` or `itRebound`. There is also a final, `default`, case that catches any programming errors that might result in a particle trying to use a wall interaction model that does not exist. `itNone` is a special wall interaction model that does nothing, often used for top and bottom wall patches in two-dimensional simulations. Such patches are only used for three-dimensional simulations and are not relevant for the two-dimensional version. The three remaining cases refer to the itemized models presented in Section 2.2. All interaction type variables (`itNone`, `itRebound`, `itStick`, `itEscape` and `itOther`) are defined as enum entries on line 74 of the `$FOAM_SRC/lagrangian/intermediate/submodels/Kinematic/PatchInteractionModel/PatchInteractionModel/PatchInteractionModel.H` file, see below. The `itOther` type is a special interaction type that only triggers an error message about an unknown interaction type.

PatchInteractionModel.H

```
63 template<class CloudType>
64 class PatchInteractionModel
65 :
66     public CloudSubModelBase<CloudType>,
67     public functionObjects::writeFile
68 {
69 public:
70
71     // Public enumerations
72
73         // Interaction types
74         enum interactionType
75         {
76             itNone,
77             itRebound,
78             itStick,
79             itEscape,
80             itOther
81         };
```

```
82
83          static wordList interactionTypeNames_;
```

Going back to the different cases in `LocalInteraction.C`, on line 246 in the listing below we have the definition of the escape model. When this model is run, it removes the current particle from the domain, records the particle escape in the `nEscape` counter and adds the particle mass to the `massEscape_` summation variable for escaped mass.

LocalInteraction.C

```
246 case PatchInteractionModel<CloudType>::itEscape:
247 {
248     keepParticle = false;
249     p.active(false);
250     U = Zero;
251
252     const scalar dm = p.mass()*p.nParticle();
253
254     nEscape_[patchi][idx]++;
255     massEscape_[patchi][idx] += dm;
256
257     if (writeFields_)
258     {
259         const label pI = pp.index();
260         const label fI = pp.whichFace(p.face());
261         massEscape().boundaryFieldRef()[pI][fI] += dm;
262     }
263     break;
264 }
```

On line 265 below we have the stick condition, that does essentially the same as the escape condition, but without actually removing the particle from the domain. The particle will therefore remain in place for as long as the simulation is running. Note that the variable `keepParticle` is `true` when the particle is sticking, `false` when it is escaping. Also note that we here use the counters `nStick` and `massStick`.

LocalInteraction.C

```
265 case PatchInteractionModel<CloudType>::itStick:
266 {
267     keepParticle = true;
268     p.active(false);
269     U = Zero;
270
271     const scalar dm = p.mass()*p.nParticle();
272
273     nStick_[patchi][idx]++;
274     massStick_[patchi][idx] += dm;
275
276     if (writeFields_)
277     {
278         const label pI = pp.index();
279         const label fI = pp.whichFace(p.face());
280         massStick().boundaryFieldRef()[pI][fI] += dm;
281     }
282     break;
283 }
```

The last particle-wall interaction model, rebound, is defined on line 284 below. On lines 309 and 310 we define the wall-normal and tangential directions. These directions are then used on lines 314 and 317 to modify the normal and tangential particle velocities, as described by the coefficients `e` and `mu`. Velocities are computed relative to the wall or patch velocity, so that everything is transformed to this new system by subtracting the patch velocity on line 295, and then adding it back on line 320. The if statement on lines 297 to 307 is used to avoid particles getting stuck on patch faces if the relative velocity between particle and wall is very low [17].

LocalInteraction.C

```
284 case PatchInteractionModel<CloudType>::itRebound:
285 {
286     keepParticle = true;
287     p.active(true);
288 
289     vector nw;
290     vector Up;
291 
292     this->owner().patchData(p, pp, nw, Up);
293 
294     // Calculate motion relative to patch velocity
295     U -= Up;
296 
297     if (mag(Up) > 0 && mag(U) < this->Urmax())
298     {
299         WarningInFunction
300         << "Particle U the same as patch "
301         << "    The particle has been removed" << nl << endl;
302 
303         keepParticle = false;
304         p.active(false);
305         U = Zero;
306         break;
307     }
308 
309     scalar Un = U & nw;
310     vector Ut = U - Un*nw;
311 
312     if (Un > 0)
313     {
314         U -= (1.0 + patchData_[patchi].e())*Un*nw;
315     }
316 
317     U -= patchData_[patchi].mu()*Ut;
318 
319     // Return velocity to global space
320     U += Up;
321 
322     break;
323 }
```

The final case on line 324, the default case, just throws an error and prints an error message indicating that we have tried to use a wall interaction model that has not been implemented. It should not be possible to reach this case unless there is a bug in the implementation.

LocalInteraction.C

```
324 default:
325 {
326     FatalErrorInFunction
327     << "Unknown interaction type "
328     << patchData_[patchi].interactionTypeName()
329     << "(" << it << ") for patch "
330     << patchData_[patchi].patchName()
331     << ". Valid selections are:" << this->interactionTypeNames_
332     << endl << abort(FatalError);
333 }
```

2.3 Running the particle tracking

Once we have set up our case and defined the particle properties and behavior we are interested in, starting the particle tracking is as simple as just running our regular transient solver for the fluid, for example `pisoFoam`.

In the example case that accompanies this report we have a simulation set-up for flow around a cylinder at Reynolds number Re = 200 000. This case also features particle tracking of small spheres using the `icoUncoupledKinematicCloud` function object, as described in Section 2.1.

Below is an excerpt of the log output when `pisoFoam` is run on this case. It would look similar even if another solver was used, for example `pimpleFoam`. Apart from the usual `pisoFoam` output with results from the linear solvers, we are also presented with some particle tracking statistics. We can see counters for number of particles, mass, momentum, etc. Of special interest for particle deposition studies we have the per-patch statistics of the count and mass of escaped and stuck particles. These statistics indicate, at a glance, if the system is behaving as expected.

Another important thing we can learn from this output is that it acts as a check that the particle tracking is indeed running. The fluid solver is perfectly capable of continuing to run, even if something is wrong with the particle tracking function object, so the fact that we can see the output from the particle tracking means that it is running.

```
Time = 0.01

Courant Number mean: 0.0202404 max: 0.809068
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual = 4.54416e-08, No Iterations 11
smoothSolver:  Solving for Uy, Initial residual = 1, Final residual = 3.83003e-08, No Iterations 10
GAMG:  Solving for p, Initial residual = 1, Final residual = 0.0341452, No Iterations 4
time step continuity errors : sum local = 4.59568e-07, global = -6.63446e-08, cumulative = -6.63446e-08
GAMG:  Solving for p, Initial residual = 0.134128, Final residual = 0.00416861, No Iterations 2
time step continuity errors : sum local = 1.49621e-07, global = -3.34946e-08, cumulative = -9.98391e-08
GAMG:  Solving for p, Initial residual = 0.0280719, Final residual = 6.94414e-07, No Iterations 12
time step continuity errors : sum local = 2.80701e-11, global = -4.81325e-12, cumulative = -9.9844e-08
smoothSolver:  Solving for omega, Initial residual = 0.00259155, Final residual = 3.70316e-08, No
     Iterations 4
smoothSolver:  Solving for k, Initial residual = 1, Final residual = 4.6953e-08, No Iterations 6
bounding k, min: 0 max: 1.18753e-15 average: 1.00278e-15
ExecutionTime = 0.06 s  ClockTime = 0 s


Solving2-D cloud kinematicCloud

Cloud: kinematicCloud injector: injection
Added 10 new parcels

Cloud: kinematicCloud
Current number of parcels       = 10
Current mass in system          = 5.17243e-09
Linear momentum                 = (5.1724e-09 -3.37811e-14 0)
|Linear momentum|                = 5.1724e-09
Linear kinetic energy           = 2.58619e-09
Average particle per parcel     = 1
Injector injection:
- parcels added             = 10
- mass introduced           = 5.17243e-09
Parcel fate: system (number, mass)
- escape                    = 0, 0
Parcel fate: patch (sides) (number, mass)
- escape                    = 0, 0
- stick                     = 0, 0
Parcel fate: patch (inlet|outlet|injectionPatch) (number, mass)
- escape                    = 0, 0
- stick                     = 0, 0
Parcel fate: patch (cylinder) (number, mass)
- escape                    = 0, 0
- stick                     = 0, 0
Rotational kinetic energy       = 0
```

2.4 Testing the stick collision model

To get a feel for how the default collision model `stick` works, we run the tutorial case with this model. The positions of particles are presented in Figure 2.1, where red particles represent particles that have attached themselves to the surface and blue particles represents particles that are free-floating. From the figure we can clearly see that we get a section of deposition on the front of the cylinder, as well as a some scattered deposition on the back side. Unexpectedly, we also get a few particles that have not attached themselves to the surface, but are nonetheless stuck right at the

surface of the wall. The reason for this is not known, but it is suspected that this phenomenon is due to the coarse mesh used in this report. An almost identical situation is reached with a pure rebound condition, where no particles should deposit on the surface. Further work is needed to properly resolve this question.

In Figure 2.2 we see a histogram of the deposited particles as a function of angle into the flow. An angle of 0° represents the direction straight into the flow direction, i.e. at the flow stagnation point. In this graph it is a bit easier to quantify deposition density compared to Figure 2.1. From the histogram we see that most of the deposition is close to the stagnation point, with some deposition on the back side of the cylinder, as expected from Figure 2.1.

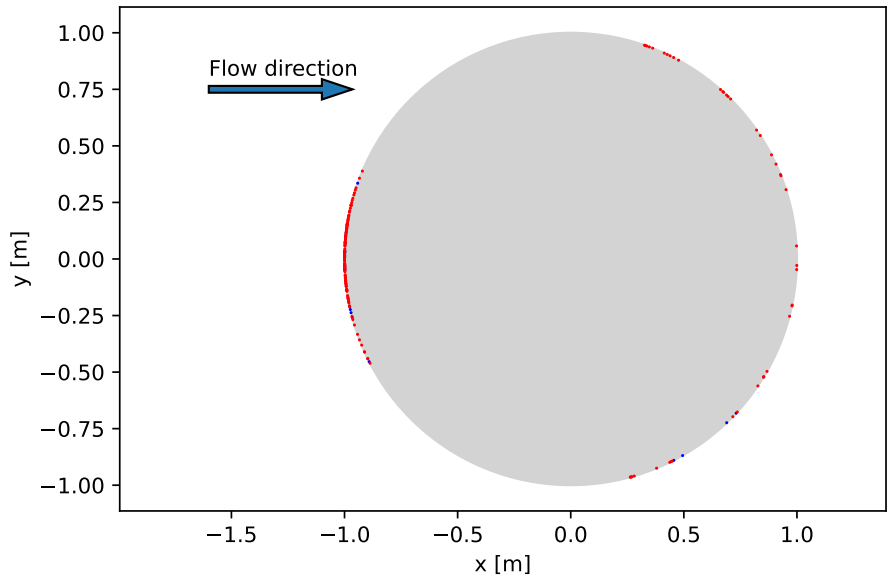

Figure 2.1: Particle deposition on a cylinder. Red represents particles that have attached to the surface, blue particles that are free-floating.

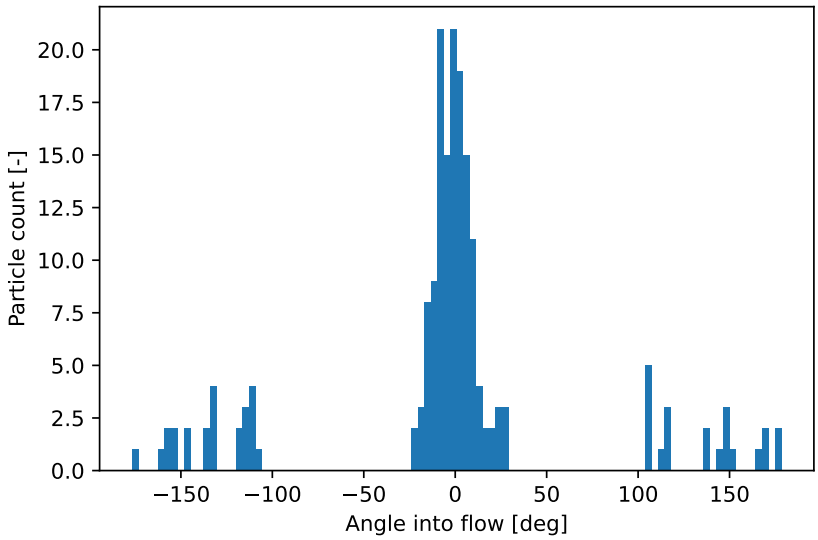

Figure 2.2: Number of particles that have deposited on a cylinder, as a function of angle into the flow.

Chapter 3

Implementing a new collision model

The two particle-wall collision models implemented in OpenFOAM, rebound and stick, work well for many theoretical cases. However, the two models are a bit too limited when it comes to real-world collisional behavior. It is entirely possible to see particles that under some conditions rebound from a surface, and stick under slightly different conditions. The rebound behavior could even have non-constant coefficients of restitution, making the case even more complex. This is, for example, seen in the case of snow particles [5]. Also particles of ash or dust show similar behavior [6].

In this report we will, as an example, implement a snow-wall collisional model by Eidevåg et al. [5] that is used in snow deposition studies in the car industry. This model describes both the rebound and sticking behavior of the snow particles, as a function of properties of the particles themselves and local flow conditions in the carrier fluid. It serves as a good illustration of the principles used in this OpenFOAM programming tutorial since it combines two of the behaviors already existing in the code into a new collisional model. By knowing how to implement this model we can easily generalize the procedure to implement any similarly behaving model.

In order to simplify the implementation of the new model, we will break down the process into a set of smaller steps. First, we will add a new particle-wall interaction model to the set of previously existing models. Then, we will add a new type of collision to this model. Once we have checked that this new model can be used by the OpenFOAM solvers, we start modifying the actual collision model itself, first by implementing an ad hoc collision model that switches between rebound and stick behavior, and finally by properly implementing the snow-wall collision model described by Eidevåg et al. [5]. An advantage with implementing everything in steps is that we continuously test our code and understanding throughout the process. We are, in some sense, always testing the smallest possible problem set.

The following sections present a step-by-step guide for implementing a new collision model from scratch. A full implementation of the final model is provided in the files accompanying this report. These accompanying files also include the test case with flow around a cylinder presented in Section 2.3.

3.1 Setting up the source code

The first step the development process is to implement a new `PatchInteractionModel` based on the existing model `LocalInteraction`. This is not strictly necessary since it is possible to just add a new collision type to the existing interaction model, but since we are going to do some important changes to how the code works it is good practice to create a new, separate, model for the new implementation. By doing this we also keep the ability of going back to the original collision model without having to recompile any code.

In this tutorial we are mainly going to be working with the `intermediate` library. It contains

much of the needed code and features for Lagrangian particle tracking. The first step is to copy the source code of this library to the user directory. To do this we use the commands

```
cd $WM_PROJECT_USER_DIR
cp -r $FOAM_SRC/lagrangian/intermediate .
```

In order to use the modifications we are about to introduce to the code, we need to compile the user-local version of the library. Thus, we first need to change the compilation target from the system directory `FOAM_LIBBIN` to `FOAM_USER_LIBBIN`.

```
cd $WM_PROJECT_USER_DIR/intermediate/Make
sed -i 's/FOAM_LIBBIN/FOAM_USER_LIBBIN/' files
```

We then compile the intermediate library using `wmake`.

```
cd $WM_PROJECT_USER_DIR/intermediate
wmake
```

This process creates the library file `$FOAM_USER_LIBBIN/liblagrangianIntermediate.so` which is dynamically loaded by OpenFOAM at runtime. It is possible to change the name of the library file from `liblagrangianIntermediate.so` to something else, but here we choose to keep the same name as the system library so that the default behavior is overridden by our custom library, without needing any further modifications. Note that the `.so` file ending represents a shared library, and that the suffix is automatically added by the compiler.

In order to start using the modified `intermediate` library we also need to recompile the library containing the Lagrangian function objects. It is located at `$FOAM_SRC/functionObjects/lagrangian`. Similarly as for the `intermediate` library, we copy the source code to the user directory, make a change to the compilation options and then compile.

```
cd $WM_PROJECT_USER_DIR
cp -r $FOAM_SRC/functionObjects/lagrangian .
cd $WM_PROJECT_USER_DIR/lagrangian/Make
sed -i 's/FOAM_LIBBIN/FOAM_USER_LIBBIN/' files
sed -i 's|-I$(LIB_SRC)/lagrangian/intermediate/lnInclude|-I$(WM_PROJECT_USER_DIR)/intermediate/
    lnInclude|' options
cd $WM_PROJECT_USER_DIR/lagrangian
wmake
```

We need to make these changes so that the Lagrangian function object library links to our new version of the intermediate library instead of the original, system-wide, version. Once the compilation is finished we can use the Lagrangian particle tracking function object with our newly compiled `intermediate` library. All further work in this tutorial will be done in the `intermediate` library and we will no longer need to touch the Lagrangian function object code.

3.2 Creating a new particle-wall collision model

The particle-wall collision model that we aim to extend, `LocalInteraction`, is found in `$WM_PROJECT_USER_DIR/intermediate/submodels/Kinematic/PatchInteractionModel/LocalInteraction/LocalInteraction.C` However, since we need to do rather significant modifications to the functionality of this file, we probably want to create a new user-selectable `patchInteractionModel` that contains our modifications. Therefore, we will start by making a copy of the `LocalInteraction` directory. The new name is arbitrary, but let us call it `LocalInteractionMix` to indicate that we are mixing the original rebound and stick models.

```
cd $WM_PROJECT_USER_DIR/intermediate/submodels/Kinematic/PatchInteractionModel
mkdir LocalInteractionMix
cp LocalInteraction/LocalInteraction.H LocalInteractionMix/LocalInteractionMix.H
cp LocalInteraction/LocalInteraction.C LocalInteractionMix/LocalInteractionMix.C
```

Now we need to replace all references to `LocalInteraction` with references to `LocalInteractionMix`.

```
cd $WM_PROJECT_USER_DIR/\
intermediate/submodels/Kinematic/PatchInteractionModel/LocalInteractionMix
sed -i 's/LocalInteraction/LocalInteractionMix/g' LocalInteractionMix.H
sed -i 's/localInteraction/localInteractionMix/g' LocalInteractionMix.H
sed -i 's/LocalInteraction/LocalInteractionMix/g' LocalInteractionMix.C
```

In order to be able to use this new class we need to compile it into our OpenFOAM user directory. We then go to the `$WM_PROJECT_USER_DIR/intermediate/Make` directory and edit the `files` file so that the compilation output directory is changed from `$(FOAM_LIBBIN)/liblagrangianIntermediate` to `$(FOAM_USER_LIBBIN)/liblagrangianIntermediate`.

```
cd $WM_PROJECT_USER_DIR/intermediate/Make/
sed -i 's/FOAM_LIBBIN/FOAM_USER_LIBBIN/g' files
```

We must now make the OpenFOAM run-time model selection mechanism aware of our new collision model. We do this by modifying `$WM_PROJECT_USER_DIR/intermediate/parcels/include/makeParcelPatchInteractionModels.H`. First we need to add our newly created header file `LocalInteractionMix.H`. Then, we need to add the macro call `makePatchInteractionModelType(LocalInteractionMix, CloudType)`.

```
cd $WM_PROJECT_USER_DIR/intermediate/parcels/include/
sed -i '/"LocalInteraction.H"/{p;s|"LocalInteraction.H"|"LocalInteractionMix.H"|;}'\
makeParcelPatchInteractionModels.H
sed -i '/LocalInteraction,/{p;s|LocalInteraction,|LocalInteractionMix,|;}'\
makeParcelPatchInteractionModels.H
```

We are now ready to compile our modified library. We then go to the root directory of the library and run the compiler.

```
cd $WM_PROJECT_USER_DIR/intermediate/
wmake
```

To make compilation faster, it is possible to replace the `wmake` command with a parallelized version. For example, to run the compilation on eight threads, use `wmake -j 8`. The resulting compiled code is the same in both cases, but the second version of the command is significantly faster on multi-core computers.

The new particle-wall collision model is now ready to use. Do, however, note that we have so far only added a new model, with a new name, that behaves exactly like the original version. To use the new collision model we simply need to change two lines in the `kinematicCloudProperties` file of our case. First, we need to specify that we want to use the new collision model by changing `patchInteractionModel localInteraction;` to `patchInteractionModel localInteractionMix;`. Then, we need to modify the name of the sub-dictionary specifying particle-wall interaction type for the different patches in the domain. To do so, simply replace `localInteractionCoeffs` with `localInteractionMixCoeffs`. The final particle-wall collision settings in the `kinematicCloudProperties` file then become

kinematicCloudProperties

```
patchInteractionModel localInteractionMix;

localInteractionMixCoeffs
{
    patches
    (
    "(sides)"
    {
        type rebound;
        e    1;  // normal restitution coefficient
        mu   0;  // tangential friction
    }

    "(inlet|outlet|injectionPatch)"
    {
```

```
16          type escape;
17      }
18
19      "(cylinder)"
20      {
21          type stick;
22      }
23
24      );
25  }
```

3.3 Modifying the new collision model

The new collision model we just created is identical in behavior to the original `LocalInteraction` model. In order to implement the new collision behavior we need to create a new type of interaction within `LocalInteractionMix`. In this report we will target a mixed rebound-stick criterion that can behave in one of two ways. For certain collisional conditions, the particle should rebound. It should then behave as if the `rebound` collision type had been selected. For other collisional conditions, the model should behave as the `stick` type. The selection between these two types is based on the collisional model described in Section 1.5.

We begin by creating a new type of particle-wall collision type within `LocalInteractionMix`. Let us call this new type `reboundStickMix`. We then add a new enum entry, `itReboundStickMix`, to `PatchInteractionModel.H`, and a string entry, `"reboundStickMix"`, to `PatchInteractionModel.C`. The two files should then look like the two listings below.

PatchInteractionModel.H

```
63  template<class CloudType>
64  class PatchInteractionModel
65  :
66      public CloudSubModelBase<CloudType>,
67      public functionObjects::writeFile
68  {
69  public:
70
71      // Public enumerations
72
73          // Interaction types
74          enum interactionType
75          {
76              itNone,
77              itRebound,
78              itStick,
79              itReboundStickMix,
80              itEscape,
81              itOther
82          };
```

PatchInteractionModel.C

```
36  template<class CloudType>
37  Foam::wordList Foam::PatchInteractionModel<CloudType>::interactionTypeNames_
38  {
39      "rebound", "stick", "reboundStickMix", "escape"
40  };
```

Additionally, we must add a new case to the switch statement, i.e.

```
case itReboundStickMix:
{
    it = "reboundStickMix";
    break;
}
```

This produces the complete switch statement below.

PatchInteractionModel.C

```
59 template<class CloudType>
60 Foam::word Foam::PatchInteractionModel<CloudType>::interactionTypeToWord
61 (
62     const interactionType& itEnum
63 )
64 {
65     word it = "other";
66
67     switch (itEnum)
68     {
69         case itNone:
70         {
71             it = "none";
72             break;
73         }
74         case itRebound:
75         {
76             it = "rebound";
77             break;
78         }
79         case itStick:
80         {
81             it = "stick";
82             break;
83         }
84         case itReboundStickMix:
85         {
86             it = "reboundStickMix";
87             break;
88         }
89         case itEscape:
90         {
91             it = "escape";
92             break;
93         }
94         default:
95         {
96         }
97     }
98
99     return it;
100 }
```

Next, we add a new if statement to translate from string to enum type.

```
else if (itWord == "reboundStickMix")
{
    return itReboundStickMix;
}
```

The complete function then looks like below.

PatchInteractionModel.C

```
103 template<class CloudType>
104 typename Foam::PatchInteractionModel<CloudType>::interactionType
105 Foam::PatchInteractionModel<CloudType>::wordToInteractionType
106 (
107     const word& itWord
108 )
109 {
110     if (itWord == "none")
111     {
112         return itNone;
113     }
```

```
114     if (itWord == "rebound")
115     {
116         return itRebound;
117     }
118     else if (itWord == "stick")
119     {
120         return itStick;
121     }
122     else if (itWord == "reboundStickMix")
123     {
124         return itReboundStickMix;
125     }
126     else if (itWord == "escape")
127     {
128         return itEscape;
129     }
130     else
131     {
132         return itOther;
133     }
134 }
```

Lastly, we add a new case to the switch statement in `LocalInteractionMix.C`, using ... as a shorthand for the contents of each switch case, giving us the following structure.

`LocalInteractionMix.C`

```
switch (it)
{
    case PatchInteractionModel<CloudType>::itNone:
    {
        //...
    }
    case PatchInteractionModel<CloudType>::itEscape:
    {
        //...
    }
    case PatchInteractionModel<CloudType>::itStick:
    {
        //...
    }
    case PatchInteractionModel<CloudType>::itRebound:
    {
        //...
    }
    case PatchInteractionModel<CloudType>::itReboundStickMix:
    {

    }
    default:
    {
        //...
    }
}
```

To prepare for using this newly defined model, we need to change line 21 in the file `kinematicCloudProperties` such that

`kinematicCloudProperties`

```
19 "(cylinder)"
20 {
21     type reboundStickMix;
22 }
```

Note that we have not yet added any code to the `itReboundStickMix` case, so running it right now will not trigger the desired behavior.

3.4 Testing the new collision type

Before we proceed any further, it is a good idea to test that our collision model works, even if it does not do anything useful yet. We would therefore like to run a very simple check is to see if we can trigger both types of behavior, rebound and stick. For the case of flow around a cylinder introduced in Section 2.3, we can use an ad hoc condition based on the sign of the particle position in the y direction. Here we use

$$\begin{cases} \text{stick} & \text{if } y_\text{p} \geq 0 \\ \text{rebound} & \text{if } y_\text{p} < 0 \end{cases},$$

where y_p represents the particle position in the y direction at the time of impact. This would mean that particles only stick to the upper half of the front side of the cylinder. Particles that impact the lower half would rebound. A condition like this has no physical meaning, but it is very simple to verify in our coding. We will in Section 3.5 replace this ad hoc condition with a proper collision model.

In the existing stick and rebound switch cases we already have all the code needed to describe the behavior of a rebounding or sticking particle. To test our new model we then just need a selection condition, essentially an `if` statement, that alternates between the two. We then call the already existing rebound or stick code to handle that specific collision.

As a first test of our new mixed model we can then use the condition `if (p.position()[1]<0)` to alternate between the two types of behavior. All we would need to do is to add the if statement and then copy the existing code from the rebound and stick cases. Schematically it would look like the code listing below.

LocalInteractionMix.C

```
case PatchInteractionModel<CloudType>::itReboundStickMix:
{
    if (p.position()[1]<0)//proof-of-concept, rebound if particle y velocity is negative, stick
     otherwise
    //very ad hoc, used just to show the concept
    {
        //rebound code ...
    }
    else
    {
        //stick code ...
    }
```

A complete listing of this ad hoc condition would then look like below.

LocalInteractionMix.C

```
case PatchInteractionModel<CloudType>::itReboundStickMix:
{
    if (p.position()[1]<0)//proof-of-concept, rebound if particle y velocity is negative, stick
     otherwise
    //very ad hoc, used just to show the concept
    {
        keepParticle = true;
        p.active(true);

        vector nw;
        vector Up;

        this->owner().patchData(p, pp, nw, Up);

        // Calculate motion relative to patch velocity
        U -= Up;

        if (mag(Up) > 0 && mag(U) < this->Urmax())
        {
            WarningInFunction
```

```
            << "Particle U the same as patch "
            << "    The particle has been removed" << nl << endl;

            keepParticle = false;
            p.active(false);
            U = Zero;
            break;
        }

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

        if (Un > 0)
        {
            U -= (1.0 + patchData_[patchi].e())*Un*nw;
        }

        U -= patchData_[patchi].mu()*Ut;

        // Return velocity to global space
        U += Up;

        break;
    }
    else
    {
        keepParticle = true;
        p.active(false);
        U = Zero;

        const scalar dm = p.mass()*p.nParticle();

        nStick_[patchi][idx]++;
        massStick_[patchi][idx] += dm;

        if (writeFields_)
        {
            const label pI = pp.index();
            const label fI = pp.whichFace(p.face());
            massStick().boundaryFieldRef()[pI][fI] += dm;
        }
        break;
    }
}
```

Now, we can compile the intermediate library with `wmake` (see Section 3.2) and run the cylinder case again. Once the case has completed we can analyze the results, which should look something like Figure 3.1. Again, red represents particles that have attached to the cylinder surface, blue that they are free-floating. We note that the behavior in Figure 3.1 is exactly the behavior we wanted to see, particles stick only to the upper part of the the cylinder. From this we conclude that our new collision model works as expected and we are ready to implement more realistic models.

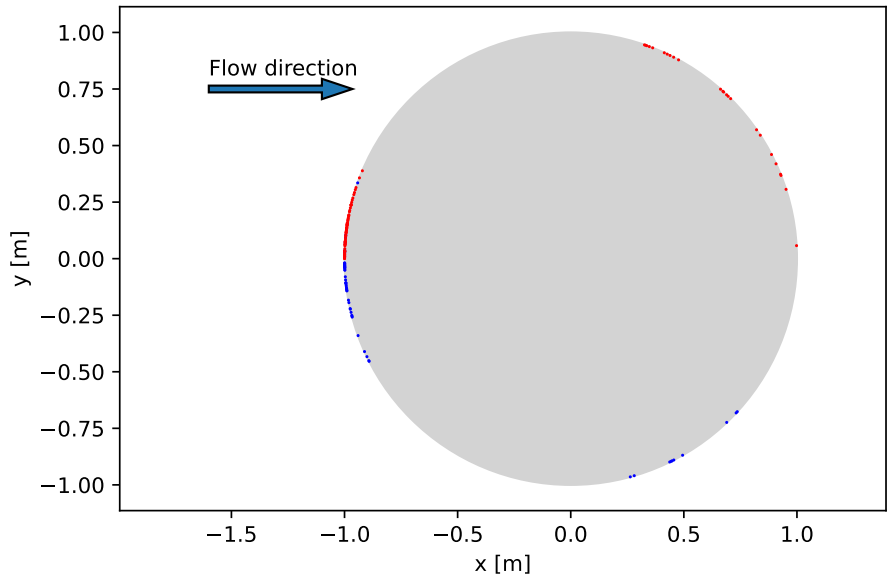

Figure 3.1: Proof-of-concept visualization of the ad hoc rebound and sticking criterion. Red indicates particles that have stuck to the cylinder surface, blue that particles are floating freely.

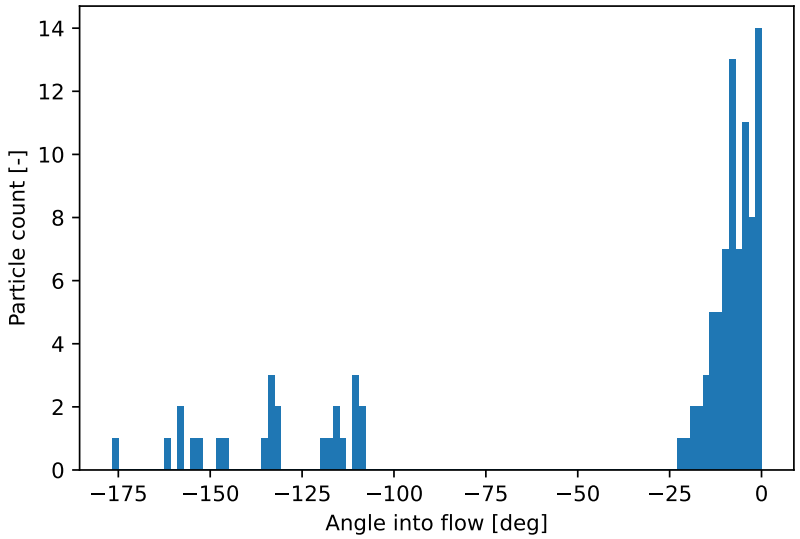

Figure 3.2: Number of particles that have deposited on a cylinder, as a function of angle into the flow for the ad-hoc collision model. Particles have only deposited in the clockwise direction counted from the stagnation point, yielding negative values for the angle.

3.5 Implementing the snow collision model

Now that we know that our implemented code changes are understood by the OpenFOAM solvers, we can start implementing the proper particle-wall collision model as presented in Section 1.5. The high-level behavior of this model is going to be quite similar to the ad hoc model, i.e. we have an `if` statement that checks a certain condition and chooses the appropriate collisional behavior, rebound or stick, depending on the condition. In the case of the Eidevåg et al. model, we have two conditions we need to check, one for the resuspension criterion and one for the sticking mechanism. If the resuspension criterion determines that the particle will be resuspended, then we perform a perfectly elastic collision. If not, then we check the adhesion criterion. If this condition predicts that the particle will stick, then we call the existing stick code. If the particle is predicted not to stick, then we run the rebound code, with coefficients of restitution given by the model. This can be implemented as in the code excerpt below.

LocalInteractionMix.C

```
//Resuspension criterion.
//If particle diameter is >= D_c, then we always get rebound.
//Else, follow adhesion model
if (p.d() >= D_c)
{
    //Particle will resuspend
    //Always rebound with e_n=e_t=1
    e_n = 1;
    e_t = 1;
    rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
}
else
{
    //check adhesion model
    if (e_n < SMALL && e_t < SMALL)
    {
        //stick
        stick(p, pp, keepParticle, patchi, U, idx);
    }
    else
    {
        //rebound, variable e_n, e_t
        rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
    }
}
```

The critical diameter D_c is calculated as presented in Section 1.5.2 and in the original publications by Eidevåg et al. [5, 11]. A listing of the necessary code to implement this calculation is presented below. This code first defines the two model parameters a and b. Then, the averaged `wallShearStress` fields are interpolated to the particle position. Finally, Eq. (1.27) is used to calculate D_c.

LocalInteractionMix.C

```
//resuspension criterion
scalar a = 77.7e-6;
scalar b = -1.34;

const volVectorField& wallShearStressMeanField =
    mesh_.lookupObject<volVectorField>("wallShearStressMean");
const volSymmTensorField& wallShearStressPrime2MeanField =
    mesh_.lookupObject<volSymmTensorField>("wallShearStressPrime2Mean");

autoPtr<interpolation<vector>> interpolatorMean =
    interpolation<vector>::New("cellPoint", wallShearStressMeanField);
auto wallShearStressMeanPointValue =
    interpolatorMean->interpolate(p.position(), p.cell());

autoPtr<interpolation<symmTensor>> interpolatorPrime2Mean =
    interpolation<symmTensor>::New("cellPoint", wallShearStressPrime2MeanField);
```

```
auto wallShearStressPrime2MeanPointValue =
    interpolatorPrime2Mean->interpolate(p.position(), p.cell());

scalar D_c = a*pow(sqrt(mag(wallShearStressMeanPointValue) +
    3*sqrt(wallShearStressPrime2MeanPointValue.xx() +
    wallShearStressPrime2MeanPointValue.yy() +
    wallShearStressPrime2MeanPointValue.zz())), b);
```

For the adhesion model, the following code is used. It first defines the numerical values for the material properties of ice-ice collisions. Then, it uses the equations presented in Section 1.5.1 to arrive at the coefficients of restitution e_{n} and e_{t}.

LocalInteractionMix.C

```
//adhesion model
scalar W = 0.218; //J/m^2, work of adhesion
scalar E_star = 5.4e9; //Pa, effective Young's modulus
scalar Dgamma_gamma = 1; //[-], adhestion hysteresis of rolling
scalar e_qe = sqrt(1-0.15);
scalar R_star = p.d()/2; //m, effective radius of contact, i.e. particle radius when colliding with
     wall, see Eidevåg 2020

scalar K_1 = 0.9355; //integration constant, see Eidevåg 2020
scalar m_p = p.mass()*p.nParticle(); //kg, mass of particle
scalar a_0 = cbrt(9*mathematical::pi*W*pow(R_star,2)/(2*E_star));
scalar E_s = 3*K_1*mathematical::pi*pow(a_0,2)*W/(4*cbrt(6.0));
scalar V_s = sqrt(2*E_s/m_p);

scalar V_cn = V_s; //Critical velocity in normal direction
scalar V_ct = 0.23*Dgamma_gamma*V_cn; //Critical velocity in tangential direction

vector nw; //wall normal vector
vector Up; //particle velocity
this->owner().patchData(p, pp, nw, Up);
vector Ur = U - Up; //particle velocity relative to wall
vector Un = (Ur & nw)*nw; //velocity component along normal direction
vector Ut = Ur - Un; //velocity component along tangential direction
scalar V_in = mag(Un);
scalar V_it = mag(Ut);

//wall-normal and tangential coefficients of restitution
scalar e_n = e_qe * sqrt(1 - pow((V_cn/(max(V_in, V_cn))),2));
scalar e_t = e_qe * sqrt(1 - pow((V_ct/(max(V_it, V_ct))),2));
```

In order to avoid code duplication, it is beneficial to convert the original rebound and stick switch cases into proper functions that can be called from any switch case. In this way we can use the same particle behavior in several different collision models. The new functions are presented below.

LocalInteractionMix.C

```
template<class CloudType>
void Foam::LocalInteractionMix<CloudType>::stick
(
typename CloudType::parcelType& p,
const polyPatch& pp,
bool& keepParticle,
const label patchi,
vector& U,
const label idx
)
{
    keepParticle = true;
    p.active(false);
    U = Zero;

    const scalar dm = p.mass()*p.nParticle();

    nStick_[patchi][idx]++;
```

```
    massStick_[patchi][idx] += dm;

    if (writeFields_)
    {
        const label pI = pp.index();
        const label fI = pp.whichFace(p.face());
        massStick().boundaryFieldRef()[pI][fI] += dm;
    }
}
```

LocalInteractionMix.C

```
template<class CloudType>
void Foam::LocalInteractionMix<CloudType>::rebound
(
typename CloudType::parcelType& p,
const polyPatch& pp,
bool& keepParticle,
const label patchi,
vector& U,
const label idx,
const scalar e_n,
const scalar e_t
)
{
    keepParticle = true;
    p.active(true);

    vector nw;
    vector Up;

    this->owner().patchData(p, pp, nw, Up);

    // Calculate motion relative to patch velocity
    U -= Up;

    if (mag(Up) > 0 && mag(U) < this->Urmax())
    {
        WarningInFunction
        << "Particle U the same as patch "
        << "    The particle has been removed" << nl << endl;

        keepParticle = false;
        p.active(false);
        U = Zero;
        return;
    }

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

    if (Un > 0)
    {
        U -= (1.0 + e_n)*Un*nw;
    }

    U -= (1-e_t)*Ut;

    // Return velocity to global space
    U += Up;
}
```

This way, the original rebound and stick cases can be simplified to a simple function call, as seen below.

LocalInteractionMix.C

```
case PatchInteractionModel<CloudType>::itStick:
{
    stick(p, pp, keepParticle, patchi, U, idx);
    break;
}
case PatchInteractionModel<CloudType>::itRebound:
{
    scalar e_n = patchData_[patchi].e();
    scalar e_t = 1-patchData_[patchi].mu();
    rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
    break;
}
```

Additionally, these functions need to be declared in the header file `LocalInteractionMix.H`.

LocalInteractionMix.H

```
template<class CloudType>
class LocalInteractionMix
:
public PatchInteractionModel<CloudType>
{
    // Private data
    //...

    virtual void stick
    (
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const label patchi,
    vector& U,
    const label idx
    );

    virtual void rebound
    (
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const label patchi,
    vector& U,
    const label idx,
    const scalar e_n,
    const scalar e_t
    );

//...
```

Finally, we must declare the the `mesh_` variable in the header file, and also initialize it in the class constructor. The declaration is seen in the listing below.

LocalInteractionMix.H

```
template<class CloudType>
class LocalInteractionMix
:
public PatchInteractionModel<CloudType>
{
    // Private data

    // Reference to fluid mesh
    const polyMesh& mesh_;

//...
```

In the constructor we define the initialization as on line 9 in the following code excerpt.

LocalInteractionMix.C

```
54 template<class CloudType>
55 Foam::LocalInteractionMix<CloudType>::LocalInteractionMix
56 (
57 const dictionary& dict,
58 CloudType& cloud
59 )
60 :
61 PatchInteractionModel<CloudType>(dict, cloud, typeName),
62 mesh_(cloud.mesh()), //this line was added
63 patchData_(cloud.mesh(), this->coeffDict()),
64 nEscape_(patchData_.size()),
65 massEscape_(nEscape_.size()),
66 nStick_(nEscape_.size()),
67 massStick_(nEscape_.size()),
68 writeFields_(this->coeffDict().getOrDefault("writeFields", false)),
69 injIdToIndex_(),
70 massEscapePtr_(nullptr),
71 massStickPtr_(nullptr)
72 {
73     //...
```

The copy constructor is handled in an equivalent way.

3.6 Testing the snow collision model

Returning again to the example case of flow around a cylinder from Section 2.3, we now use the `localInteractionMix` model representing our new collision model. A plot of the particle deposition results using this model is presented in Figure 3.3. A clear distinction can be seen between this figure and Figure 3.1. In Figure 3.3, far fewer particles actually stick to the cylinder. This observation becomes even clearer if we compare the corresponding histograms in Figures 3.4 and 2.2. It is then clear that this collision model has a pronounced effect on particle deposition rates, with an approximately 60 % reduction in deposition rates.

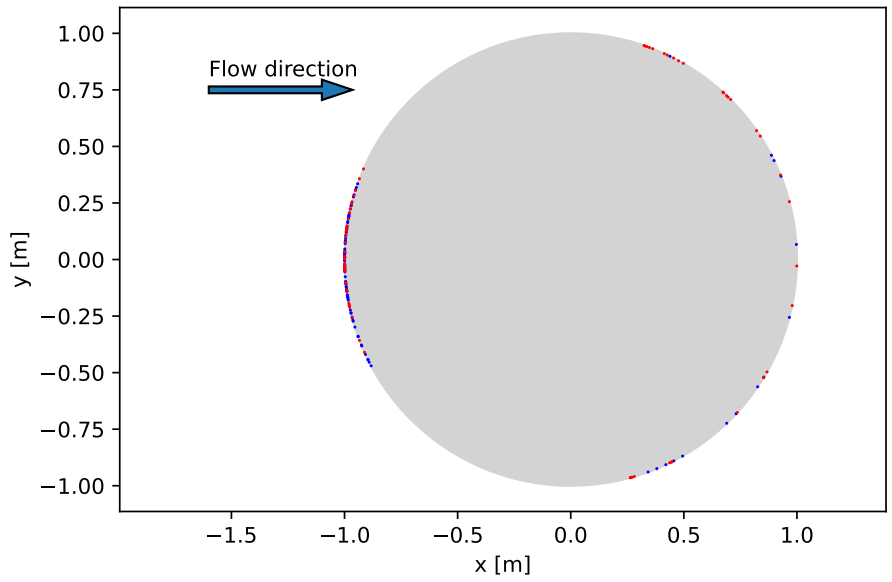

Figure 3.3: Proof-of-concept visualization of the ad hoc rebound and sticking criterion. Red indicates particles that have stuck to the cylinder surface, blue that particles are floating freely.

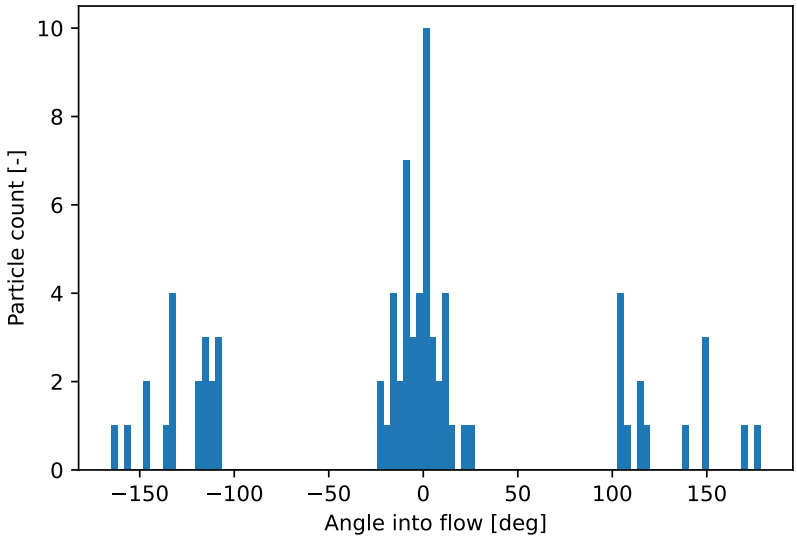

Figure 3.4: Number of particles that have deposited on a cylinder, as a function of angle into the flow for the ad-hoc collision model. Around 60 % fewer particles have deposited compared to the `stick` case in Figure 2.2.

An important thing to keep in mind is that this report focuses on the methods and implementation strategies needed to implement new collision models into OpenFOAM. Further work is needed to validate the fluid flow and particle deposition results. A good follow-up study would be to validate the implemented collision model with the original results presented by Eidevåg et al. [5].

Bibliography

[1] J. Vandenkerckhove, "Technical comments," *ARS Journal*, vol. 31, no. 10, pp. 1466–1469, 1961, publisher: American Institute of Aeronautics and Astronautics. [Online]. Available: https://arc.aiaa.org/doi/10.2514/8.5825

[2] A. A. Amsden, P. J. O'Rourke, and T. D. Butler, "KIVA-II: A computer program for chemically reactive flows with sprays," 1989, technical report, Los Alamos National Lab. [Online]. Available: https://www.osti.gov/biblio/6228444

[3] (2023) OpenFOAM: API guide: SphereDragForce< CloudType > class template reference. [Online]. Available: https://www.openfoam.com/documentation/guides/latest/api/classFoam_1_1SphereDragForce.html

[4] N. E. L. Haugen and S. Kragset, "Particle impaction on a cylinder in a cross-flow as function of stokes and reynolds numbers," *Journal of Fluid Mechanics*, vol. 661, pp. 239–261, 2010, publisher: Cambridge University Press. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/particle-impaction-on-a-cylinder-in-a-crossflow-as-function-of-stokes-and-reynolds-numbers/30E0BF549C92BC497017023646DEE250

[5] T. Eidevåg, M. Eng, D. Kallin, J. Casselgren, Y. Bharadhwaj, T. S. Bangalore Narahari, and A. Rasmuson, "Snow contamination of simplified automotive bluff bodies: A comparison between wind tunnel experiments and numerical modeling," *SAE International Journal of Advances and Current Practices in Mobility*, pp. 2022–01–0901, 2022. [Online]. Available: https://www.sae.org/content/2022-01-0901/

[6] U. Kleinhans, C. Wieland, F. J. Frandsen, and H. Spliethoff, "Ash formation and deposition in coal and biomass fired combustion systems: Progress and challenges in the field of ash particle sticking and rebound behavior," *Progress in Energy and Combustion Science*, vol. 68, pp. 65–168, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360128517300795

[7] J. Hærvig, U. Kleinhans, C. Wieland, H. Spliethoff, A. L. Jensen, K. Sørensen, and T. J. Condra, "On the adhesive JKR contact and rolling models for reduced particle stiffness discrete element simulations," *Powder Technology*, vol. 319, pp. 472–482, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0032591017305430

[8] J. S. Marshall, "Discrete-element modeling of particulate aerosol flows," *Journal of Computational Physics*, vol. 228, no. 5, pp. 1541–1561, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002199910800572X

[9] M. Higa, M. Arakawa, and N. Maeno, "Size dependence of restitution coefficients of ice in relation to collision strength," *Icarus*, vol. 133, no. 2, pp. 310–320, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019103598959383

[10] T. Eidevåg, E. S. Thomson, S. Sollén, J. Casselgren, and A. Rasmuson, "Collisional damping of spherical ice particles," *Powder Technology*, vol. 383, pp. 318–327, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0032591021000413

[11] T. Eidevåg, P. Abrahamsson, M. Eng, and A. Rasmuson, "Modeling of dry snow adhesion during normal impact with surfaces," *Powder Technology*, vol. 361, pp. 1081–1092, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0032591019309143

[12] H. Reichardt, "Vollständige darstellung der turbulenten geschwindigkeitsverteilung in glatten leitungen," vol. 31, no. 7, pp. 208–219, 1951. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19510310704

[13] Reichardt profile – CFD-wiki, the free CFD reference. [Online]. Available: https://www.cfd-online.com/Wiki/Reichardt_profile

[14] F. M. White, *Fluid mechanics*, 8th ed. McGraw-Hill Education, 2016.

[15] (2023) OpenFOAM: API guide: icoUncoupledKinematicCloud class reference. [Online]. Available: https://www.openfoam.com/documentation/guides/latest/api/classFoam_1_1functionObjects_1_1icoUncoupledKinematicCloud.html

[16] (2023) OpenFOAM: API guide: PatchInteractionModel< CloudType > class template reference. [Online]. Available: https://www.openfoam.com/documentation/guides/latest/api/classFoam_1_1PatchInteractionModel.html

[17] (2020) ENH: Adding check for wall interaction when particle is stuck on moving (9207140e) · commits · development / openfoam · GitLab. [Online]. Available: https://develop.openfoam.com/Development/openfoam/-/commit/9207140e377364fd73ebd2cf2006bb7837335ea2

Study questions

1. Why do we need a combined rebound and stick particle-wall collision model?
2. What are the benefits and limitations of one-way coupled particle tracking?
3. Are the effects of wall roughness in particle impacts included in the implementation presented in this report?
4. What is the purpose of the resuspension model?

Appendix A

The kinematicCloudProperties file

kinematicCloudProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      kinematicCloudProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solution
{
    active          true;
    coupled         false;
    transient       yes;
    cellValueSourceCorrection off;
    //maxCo           0.3;

    interpolationSchemes
    {
        rho             cell;
        U               cellPoint;
        mu              cell;
    }

    averagingMethod basic;

    integrationSchemes
    {
        U               Euler;
    }

}

constantProperties
{
    parcelTypeId 1;

    rhoMin          1e-15;
    minParcelMass   1e-15;
```

```
48
49      rho0            917;
50      youngsModulus   1e8;
51      poissonsRatio   0.35;
52
53  }
54
55  subModels
56  {
57      particleForces
58      {
59          sphereDrag;
60      }
61
62      injectionModels
63      {
64
65          injection
66          {
67              type            patchInjection;
68              parcelBasisType fixed;
69              //patch           inlet;
70              patch           injectionPatch;
71              U0              (1 0 0);
72              nParticle       1;
73              parcelsPerSecond 1000;
74
75              sizeDistribution
76              {
77
78                  type        binned;
79                  binnedDistribution
80                  {
81                      distribution
82                      (
83                          (0.000030 0.2)
84                          (0.000050 0.2)
85                          (0.000075 0.2)
86                          (0.000100 0.2)
87                          (0.000150 0.2)
88                      );
89                  }
90              }
91              flowRateProfile constant 1;
92              massTotal       0;
93              SOI             0;
94              duration        30;
95          }
96
97      }
98
99      patchInteractionModel localInteractionMix;
100
101     localInteractionMixCoeffs
102     {
103         patches
104         (
105             "(sides)"
106             {
107                 type rebound;
108                 e    1;  // normal restitution coefficient
109                 mu   0;  // tangential friction
110             }
111
112             "(inlet|outlet|injectionPatch)"
113             {
114                 type escape;
115             }
```

```
116
117             "(cylinder)"
118             {
119                 type reboundStickMix;
120             }
121
122         );
123     }
124
125     dispersionModel none;
126
127     stochasticCollisionModel none;
128
129     surfaceFilmModel none;
130
131     collisionModel none;
132
133 }
134
135
136 cloudFunctions
137 {
138 }
139
140
141 // ************************************************************************* //
```

Appendix B

The `LocalInteraction.C` file

LocalInteraction.C

```
212 template<class CloudType>
213 bool Foam::LocalInteraction<CloudType>::correct
214 (
215     typename CloudType::parcelType& p,
216     const polyPatch& pp,
217     bool& keepParticle
218 )
219 {
220     const label patchi = patchData_.applyToPatch(pp.index());
221 
222     if (patchi >= 0)
223     {
224         vector& U = p.U();
225 
226         // Location for storing the stats.
227         const label idx =
228         (
229             injIdToIndex_.size()
230           ? injIdToIndex_.lookup(p.typeId(), 0)
231           : 0
232         );
233 
234         typename PatchInteractionModel<CloudType>::interactionType it =
235             this->wordToInteractionType
236             (
237                 patchData_[patchi].interactionTypeName()
238             );
239 
240         switch (it)
241         {
242             case PatchInteractionModel<CloudType>::itNone:
243             {
244                 return false;
245             }
246             case PatchInteractionModel<CloudType>::itEscape:
247             {
248                 keepParticle = false;
249                 p.active(false);
250                 U = Zero;
251 
252                 const scalar dm = p.mass()*p.nParticle();
253 
254                 nEscape_[patchi][idx]++;
255                 massEscape_[patchi][idx] += dm;
256 
257                 if (writeFields_)
258                 {
```

```
259                     const label pI = pp.index();
260                     const label fI = pp.whichFace(p.face());
261                     massEscape().boundaryFieldRef()[pI][fI] += dm;
262                 }
263                 break;
264             }
265             case PatchInteractionModel<CloudType>::itStick:
266             {
267                 keepParticle = true;
268                 p.active(false);
269                 U = Zero;
270
271                 const scalar dm = p.mass()*p.nParticle();
272
273                 nStick_[patchi][idx]++;
274                 massStick_[patchi][idx] += dm;
275
276                 if (writeFields_)
277                 {
278                     const label pI = pp.index();
279                     const label fI = pp.whichFace(p.face());
280                     massStick().boundaryFieldRef()[pI][fI] += dm;
281                 }
282                 break;
283             }
284             case PatchInteractionModel<CloudType>::itRebound:
285             {
286                 keepParticle = true;
287                 p.active(true);
288
289                 vector nw;
290                 vector Up;
291
292                 this->owner().patchData(p, pp, nw, Up);
293
294                 // Calculate motion relative to patch velocity
295                 U -= Up;
296
297                 if (mag(Up) > 0 && mag(U) < this->Urmax())
298                 {
299                     WarningInFunction
300                         << "Particle U the same as patch "
301                         << "    The particle has been removed" << nl << endl;
302
303                     keepParticle = false;
304                     p.active(false);
305                     U = Zero;
306                     break;
307                 }
308
309                 scalar Un = U & nw;
310                 vector Ut = U - Un*nw;
311
312                 if (Un > 0)
313                 {
314                     U -= (1.0 + patchData_[patchi].e())*Un*nw;
315                 }
316
317                 U -= patchData_[patchi].mu()*Ut;
318
319                 // Return velocity to global space
320                 U += Up;
321
322                 break;
323             }
324             default:
325             {
326                 FatalErrorInFunction
```

```
327                     << "Unknown interaction type "
328                     << patchData_[patchi].interactionTypeName()
329                     << "(" << it << ") for patch "
330                     << patchData_[patchi].patchName()
331                     << ". Valid selections are:" << this->interactionTypeNames_
332                     << endl << abort(FatalError);
333             }
334         }
335
336         return true;
337     }
338
339     return false;
340 }
```

Appendix C

The `LocalInteractionMix.C` file

LocalInteractionMix.C

```
214 template<class CloudType>
215 void Foam::LocalInteractionMix<CloudType>::stick
216 (
217     typename CloudType::parcelType& p,
218     const polyPatch& pp,
219     bool& keepParticle,
220     const label patchi,
221     vector& U,
222     const label idx
223 )
224 {
225     keepParticle = true;
226     p.active(false);
227     U = Zero;
228
229     const scalar dm = p.mass()*p.nParticle();
230
231     nStick_[patchi][idx]++;
232     massStick_[patchi][idx] += dm;
233
234     if (writeFields_)
235     {
236         const label pI = pp.index();
237         const label fI = pp.whichFace(p.face());
238         massStick().boundaryFieldRef()[pI][fI] += dm;
239     }
240 }
241
242 template<class CloudType>
243 void Foam::LocalInteractionMix<CloudType>::rebound
244 (
245     typename CloudType::parcelType& p,
246     const polyPatch& pp,
247     bool& keepParticle,
248     const label patchi,
249     vector& U,
250     const label idx,
251     const scalar e_n,
252     const scalar e_t
253 )
254 {
255     keepParticle = true;
256     p.active(true);
257
258     vector nw;
259     vector Up;
260
```

```
261     this->owner().patchData(p, pp, nw, Up);
262
263     // Calculate motion relative to patch velocity
264     U -= Up;
265
266     if (mag(Up) > 0 && mag(U) < this->Urmax())
267     {
268         WarningInFunction
269             << "Particle U the same as patch "
270             << "    The particle has been removed" << nl << endl;
271
272         keepParticle = false;
273         p.active(false);
274         U = Zero;
275         return;
276     }
277
278     scalar Un = U & nw;
279     vector Ut = U - Un*nw;
280
281     if (Un > 0)
282     {
283         U -= (1.0 + e_n)*Un*nw;
284     }
285
286     U -= (1-e_t)*Ut;
287
288     // Return velocity to global space
289     U += Up;
290 }
291
292
293 template<class CloudType>
294 bool Foam::LocalInteractionMix<CloudType>::correct
295 (
296     typename CloudType::parcelType& p,
297     const polyPatch& pp,
298     bool& keepParticle
299 )
300 {
301     const label patchi = patchData_.applyToPatch(pp.index());
302
303     if (patchi >= 0)
304     {
305         vector& U = p.U();
306
307         // Location for storing the stats.
308         const label idx =
309         (
310             injIdToIndex_.size()
311           ? injIdToIndex_.lookup(p.typeId(), 0)
312           : 0
313         );
314
315         typename PatchInteractionModel<CloudType>::interactionType it =
316             this->wordToInteractionType
317             (
318                 patchData_[patchi].interactionTypeName()
319             );
320
321         switch (it)
322         {
323             case PatchInteractionModel<CloudType>::itNone:
324             {
325                 return false;
326             }
327             case PatchInteractionModel<CloudType>::itEscape:
328             {
```

```
329                 keepParticle = false;
330                 p.active(false);
331                 U = Zero;
332
333                 const scalar dm = p.mass()*p.nParticle();
334
335                 nEscape_[patchi][idx]++;
336                 massEscape_[patchi][idx] += dm;
337
338                 if (writeFields_)
339                 {
340                     const label pI = pp.index();
341                     const label fI = pp.whichFace(p.face());
342                     massEscape().boundaryFieldRef()[pI][fI] += dm;
343                 }
344                 break;
345             }
346             case PatchInteractionModel<CloudType>::itStick:
347             {
348                 stick(p, pp, keepParticle, patchi, U, idx);
349                 break;
350             }
351             case PatchInteractionModel<CloudType>::itRebound:
352             {
353                 scalar e_n = patchData_[patchi].e();
354                 scalar e_t = 1-patchData_[patchi].mu();
355                 rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
356                 break;
357             }
358             case PatchInteractionModel<CloudType>::itReboundStickMix:
359             {
360                 //adhesion model
361                 scalar W = 0.218; //J/m^2, work of adhesion
362                 scalar E_star = 5.4e9; //Pa, effective Young's modulus
363                 scalar Dgamma_gamma = 1; //[-], adhestion hysteresis of rolling
364                 scalar e_qe = sqrt(1-0.15);
365                 scalar R_star = p.d()/2; //m, effective radius of contact, i.e. particle radius when
    colliding with wall, see Eidevåg 2020
366
367                 scalar K_1 = 0.9355; //integration constant, see Eidevåg 2020
368                 scalar m_p = p.mass()*p.nParticle(); //kg, mass of particle
369                 scalar a_0 = cbrt(9*mathematical::pi*W*pow(R_star,2)/(2*E_star));
370                 scalar E_s = 3*K_1*mathematical::pi*pow(a_0,2)*W/(4*cbrt(6.0));
371                 scalar V_s = sqrt(2*E_s/m_p);
372
373                 scalar V_cn = V_s; //Critical velocity in normal direction
374                 scalar V_ct = 0.23*Dgamma_gamma*V_cn; //Critical velocity in tangential direction
375
376                 vector nw; //wall normal vector
377                 vector Up; //particle velocity
378                 this->owner().patchData(p, pp, nw, Up);
379                 vector Ur = U - Up; //particle velocity relative to wall
380                 vector Un = (Ur & nw)*nw; //velocity component along normal direction
381                 vector Ut = Ur - Un; //velocity component along tangential direction
382                 scalar V_in = mag(Un);
383                 scalar V_it = mag(Ut);
384
385                 //wall-normal and tangential coefficients of restitution
386                 scalar e_n = e_qe * sqrt(1 - pow((V_cn/(max(V_in, V_cn))),2));
387                 scalar e_t = e_qe * sqrt(1 - pow((V_ct/(max(V_it, V_ct))),2));
388
389                 //resuspension criterion
390                 scalar a = 77.7e-6;
391                 scalar b = -1.34;
392
393                 const volVectorField& wallShearStressMeanField =
394                     mesh_.lookupObject<volVectorField>("wallShearStressMean");
395                 const volSymmTensorField& wallShearStressPrime2MeanField =
```

```
396                     mesh_.lookupObject<volSymmTensorField>("wallShearStressPrime2Mean");
397
398                 autoPtr<interpolation<vector>> interpolatorMean =
399                     interpolation<vector>::New("cellPoint", wallShearStressMeanField);
400                 auto wallShearStressMeanPointValue =
401                     interpolatorMean->interpolate(p.position(), p.cell());
402
403                 autoPtr<interpolation<symmTensor>> interpolatorPrime2Mean =
404                     interpolation<symmTensor>::New("cellPoint", wallShearStressPrime2MeanField);
405                 auto wallShearStressPrime2MeanPointValue =
406                     interpolatorPrime2Mean->interpolate(p.position(), p.cell());
407
408                 scalar D_c = a*pow(sqrt(mag(wallShearStressMeanPointValue) +
409                     3*sqrt(wallShearStressPrime2MeanPointValue.xx() +
410                     wallShearStressPrime2MeanPointValue.yy() +
411                     wallShearStressPrime2MeanPointValue.zz())), b);
412
413                 //Resuspension criterion.
414                 //If particle diameter is >= D_c, then we always get rebound.
415                 //Else, follow adhesion model
416                 if (p.d() >= D_c)
417                 {
418                     //Particle will resuspend
419                     //Always rebound with e_n=e_t=1
420                     e_n = 1;
421                     e_t = 1;
422                     rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
423                 }
424                 else
425                 {
426                     //check adhesion model
427                     if (e_n < SMALL && e_t < SMALL)
428                     {
429                         //stick
430                         stick(p, pp, keepParticle, patchi, U, idx);
431                     }
432                     else
433                     {
434                         //rebound, variable e_n, e_t
435                         rebound(p,pp,keepParticle,patchi,U,idx,e_n,e_t);
436                     }
437                 }
438
439                 break;
440
441             }
442             default:
443             {
444                 FatalErrorInFunction
445                     << "Unknown interaction type "
446                     << patchData_[patchi].interactionTypeName()
447                     << "(" << it << ") for patch "
448                     << patchData_[patchi].patchName()
449                     << ". Valid selections are:" << this->interactionTypeNames_
450                     << endl << abort(FatalError);
451             }
452         }
453
454         return true;
455     }
456
457     return false;
458 }
```