LABORATORY OF CRYOSPHERIC SCIENCES (CRYOS)

OpenFOAM Tutorial Project

Implementation of an Eulerian-Lagrangian snow transport model in OpenFOAM: *snowBedFoam 1.0*

Developed for OpenFOAM version 2.3.0.

Author:
Océane HAMES

Supervisors:
Mahdi JAFARI
Michael LEHNING

Thursday 22nd July, 2021

Contents

1 Introduction

The present document describes the implementation of an aeolian snow transport model within the open source computational fluid dynamics (CFD) software OpenFOAM. It was developed in the context of a master thesis at the Ecole Polytechnique Fédérale de Lausanne (EPFL) and the WSL Institute for Snow and Avalanche Research SLF, Switzerland. Two submodels are added to the original OpenFOAM Lagrangian library to simulate the transport of snow particles by the wind, in particular for medium- (saltation) and small-sized (suspension) particles. Herein, we first describe the theoretical framework for snow transport processes and their related mathematical expressions. Then, we present the OpenFOAM scripts embedding the different Lagrangian submodels for snow movement along with the files defining the new solver (*snowBedFoam*). This tutorial aims to make the modelling of snow transport more accessible to the OpenFOAM community.

2 Theoretical background

The current knowledge of snow transport processes which contributed to the build-up of our OpenFOAM model is described here. The text was inspired by the candidacy report of Brito Melo (2019), which outlines the main literature findings on the aeolian transport of particles.

2.1 Snow transport: general aspects

The early work of Bagnold (1941) constitutes a reference for the aeolian transport of sand. Still, his findings stay relevant to other particles, among which snow and its various interactions with wind. Snow aeolian transport occurs at a wide range of elevations, from regions close to the ground to high altitudes. Three main modes of transport are distinguished based on their underlying physical processes: 1) *saltation*, which describes the motion of particles close to the surface. In this mode, grains follow ballistic trajectories and return to the snow bed, possibly rebounding and/or ejecting other particles; ; 2) *suspension*, which relates to the transport of particles that are sufficiently light to be lifted higher up by turbulent eddies; 3) *creep* (or *reptation*), which is the rolling of heavier particles along the surface due to impacting grains or aerodynamic forces. Its contribution usually stays negligible compared to the other processes (Vionnet et al., 2013).

The saltation of grains along the surface accounts for about 75% of all particle movement by wind (Bagnold, 1941). In our OpenFOAM model, both the suspension and saltation modes are represented but saltation stays predominant. Most saltating particles are confined to a thin layer close to the surface (∼10 cm), which we refer to as the *saltation layer*. When aloft, saltating particles are accelerated by the fluid flow: their kinetic energy is partly dissipated as friction losses, partly sustained to start a new ballistic trajectory and partly transferred to eject other grains from the snowbed surface. A total of three saltation modes are commonly identified (Figure 1): aerodynamic entrainment, rebound and ejection. *Aerodynamic entrainment* (or lift) occurs when particles initially at the surface are lifted up by aerodynamic forces only. *Rebound* happens when particles bounce to a new ballistic trajectory after hitting the ground. *Ejection* (or splash) occurs when particles laying in the ground are set in motion due to the impact of saltating particles (Doorschot and Lehning, 2002). These transport modes are in fact modes of saltation initiation, which have a great impact on the ballistic trajectory of the particle. Different authors contributed to the physical understanding of these saltation modes and developed parametrizations of the wind-particle-bed interaction. We particularly refer to Comola and Lehning (2017) whose findings were implemented in the snow transport solver described herein.

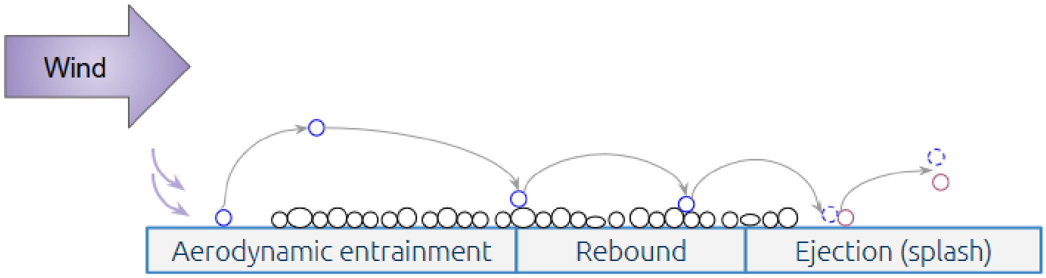

Figure 1: The three main particle saltation modes: aerodynamic entrainment, rebound and ejection. Adapted from Brito Melo (2019).

Based on several wind tunnel experiments with sand of uniform grain size that he conducted, Bagnold (1941) could establish the concept of *fluid threshold* which is the wind speed necessary for grains to start saltating when initially at rest. This threshold value varies in direct proportion to the predominant grain size of the sand surface. Thus, saltation starts when the shear stress exceeds the fluid threshold: this is an important concept for the build-up of our snow movement model. All the related mathematical expressions are detailed in the next subsection.

2.2 Governing equations for snow surface-flow interaction

The underlying principles and equations of the OpenFOAM snow transport model are described here. They are similar to those of the Large Eddy Simulation-Lagrangian Stochastic Model (LES-LSM) developed within the CRYOS laboratory at EPFL (CRYOS, 2021). We refer to the work of Comola and Lehning (2017) and Sharma et al. (2018) for more details. The three saltation modes - aerodynamic lift, rebounding and ejection of grains - are represented in the model under a mathematical form and implemented in the scripts in such manner.

2.2.1 Aerodynamic entrainment

Grains lying on the snow bed can be entrained into the saltation layer when the fluid surface shear stress $\tau_{f,surf}$ is large enough to lift them up, namely when it exceeds the fluid threshold value τ_{th} defined as (Bagnold, 1941)

$$\tau_{th} = A^2 g \langle d_p \rangle (\rho_p - \rho_f) \tag{1}$$

where $\langle d_p \rangle$ is the mean particle diameter, ρ_p and ρ_f are the particle and fluid densities, respectively and A is an empirical constant taken equal to 0.2 for snow as determined by Clifton et al. (2006) through wind-tunnel experiments. g refers to the gravitational acceleration and is assumed to be equal to 9.81 m/s^2.

Two different formulations for surface shear stress were implemented in the OpenFOAM aerodynamic lift submodel. The first one is is obtained by applying the logarithmic law of the wall (LOW),

$$\tau_{f,surf}^{LOW} = \rho_f \left(\frac{\kappa |\mathbf{U}_t|}{ln(z/z_0)} \right)^2 \tag{2}$$

where $\mathbf{U}_t$ is the tangential velocity vector, z the height of the first grid cell center, z_0 the aerodynamic roughness length and κ = 0.41 the von Kármán constant. The second expression is based on the vertical velocity gradient and the total kinematic viscosity:

$$\tau_{f,surf}^{TKV} = \rho_f \left(\left. \frac{\partial u}{\partial z} \right|_{z=0} (\nu_t + \nu) \right) \tag{3}$$

where $\frac{\partial u}{\partial z}|_{z=0}$ is the vertical velocity gradient and ν, ν_t the viscous and turbulent kinematic viscosity, respectively. This method has the advantage to be universal and independent of the wall function employed in the simulations. In each grid cell, the number of particles aerodynamically entrained by the fluid at each timestep, N_{ae}, linearly increases with the excess shear stress according to the formulation of Anderson and Haff (1991):

$$N_{ae} = \frac{C_e}{8\pi \langle d_p \rangle^2}(\tau_{f,surf} - \tau_{th})\Delta x \Delta y \Delta t \tag{4}$$

where C_e is an empirical parameter set to 1.5 (Doorschot and Lehning, 2002), Δx and Δy are the grid dimensions in the streamwise/spanwise directions and Δt is the simulation timestep. Once that N_{ae} is determined, particles are launched at height $h_{init} = 4\langle d_p \rangle$ and the particle diameter, initial velocity magnitude and ejection angle are all sampled from statistical distributions according to Clifton and Lehning (2008). More details can be found in their work.

2.2.2 Rebound and splash entrainment

Depending on its path, a snow particle present in the fluid might hit the surface upon which it can not only rebound -defined as *rebound* entrainment- but also eject other particles from the bed to the overlying fluid, defined as *splash* entrainment. The probability P_r that the snow particle rebounds when impacting the bed is given by Anderson and Haff (1991) as follows

$$P_r = P_m(1 - e^{-\gamma v_i}) \tag{5}$$

where P_m is the maximum probability equal to 0.9 for snow (Groot Zwaaftink et al., 2013), γ is an empirical constant equal to 2, and v_i is the velocity magnitude of the impacting particle. When rebounding, the particle is assumed to have a velocity magnitude of $v_r = 0.5v_i$ (Doorschot and Lehning, 2002) and the rebound angle is determined from a statistical distribution according to Kok and Rennó (2009).

Concerning the splash entrainment, the number of particles ejected from the bed N_{splash} is defined as the minimum between N_E and N_M whose expressions are (Comola and Lehning, 2017):

$$N_E = \frac{(1 - P_r\epsilon_r - \epsilon_{fr})d_i^3 v_i^2}{2\langle v \rangle^2(\langle d \rangle + \frac{\sigma_d^2}{\langle d \rangle})^3 \left(1 + r_E\sqrt{5[1 + (\frac{\sigma_d}{\langle d \rangle})^2]^9 - 5}\right) + 2\frac{\phi}{\rho_p}} \tag{6}$$

$$N_M = \frac{(1 - P_r\mu_r - \mu_{fr})d_i^3 v_i cos\alpha_i}{\langle v \rangle^2(\langle d \rangle + \frac{\sigma_d^2}{\langle d \rangle})^3 \left(\langle cos\alpha \rangle \langle cos\beta \rangle r_M\sqrt{[1 + (\frac{\sigma_d}{\langle d \rangle})^2]^9 - 1}\right)} \tag{7}$$

N_M and N_E are the number of ejections predicted by the momentum and energy balance, respectively. In Eq.6, ϵ_{fr} and ϵ_r are the fractions of impact energy lost to the bed and kept by the rebounding particle, respectively. μ_{fr} and μ_r are their equivalent for momentum in Eq.7. $\langle d\rangle$ and σ_d are the mean and standard deviation of the ejecta's diameter, $\langle v\rangle$ its mean velocity and α and β the horizontal and vertical ejection angles. ϕ is the cohesive bond exerted on a particle by its neighboring particles. r_M and r_E are correlation coefficients linking mass and velocity. More details about the derivation of these expressions can be found in the work of Comola and Lehning (2017). Similarly to the aerodynamic entrainment, the characteristics of the splashed particles are randomly sampled from statistical distributions. Overall, details about the equations of the surface-flow interaction can be found in the Supplementary Materials of the work from Sharma et al. (2018).

3 Implementation of a snow transport model

From this section on, several fonts are employed: the OpenFOAM font refers to the solver and function names of the software; the `command` font is employed when referring to a terminal command and directory/file names.

The existing OpenFOAM solver DPMFoam is employed for the implementation of this new snow transport model. It employs the lagrangian library of the software which compiles a variety of Lagrangian particle tracking (LPT) libraries. DPMFoam is a multiphase flow solver that handles the coupled Eulerian–Lagrangian phases and involves a finite number of particles spread in a continuous phase. The motion of individual particles is obtained directly by solving Newton's second law of motion, which corresponds to the so-called discrete particle method (DPM). Particles are aggregated in clouds and treated as one big computational parcel, where the effect of the volume fraction of particles on the continuous phase is included within the Eulerian continuum equations. Details on the numerical approach employed in DPMFoam are given in Fernandes et al. (2018), along with validation results.

Overall, the following steps must be performed for the build-up of a snow transport model:

1. The implementation of the aerodynamic entrainment equations (Sect.2.2.1) using the stochasticCollision submodel as a base;
2. The implementation of the rebound and splash entrainment modes (Sect.2.2.2) based on the patch-interaction submodel localInteraction;

3. The integration of a source term in the Eulerian momentum equations in the form of a large-scale pressure gradient;
4. The set-up of an initial velocity profile following the logarithmic law to reach faster convergence. Addition of an extra term for random noise is also made possible in order to mimic the effect of turbulence.
5. The addition of so-called volScalarField objects for the visualization of particle deposition and entrainment as well as surface friction velocity at the surface.

The main implementation steps mentioned above are described in details in the following sections. The first step consists in copying into the user folder (`$WM_PROJECT_USER_DIR`) the intermediate and distributionModels directories from the lagrangian library located in the OpenFOAM source libraries directory (`$FOAM_SRC`) through the following command:

```
cp -r $FOAM_SRC/lagrangian/intermediate \
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/
```

Comparably for the distributionModels library:

```
cp -r $FOAM_SRC/lagrangian/distributionModels \
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/distributionModelsTriple/
```

It is important that the folders copied in the user folder are named differently than the original ones to avoid compiling issues: here the names `lagrangianCRYOS`, `intermediateCRYOS` and `distributionModelsTriple` can be replaced by any other meaningful terms. Once that the folders are set, the name changes need to be taken into account for the compilation of the modified libraries. At the last line of the file `Make/files` in the respective folders, replace the `$(FOAM_LIBBIN)` expression by `$(FOAM_USER_LIBBIN)` and add the name of the new library:

```
LIB = $(FOAM_USER_LIBBIN)/liblagrangianIntermediateCRYOS
LIB = $(FOAM_USER_LIBBIN)/libdistributionModelTriple
```

Similarly, the DPMFoam solver folder must be copied from the OpenFOAM applications directory (`$FOAM_APP`) via the command:

```
cp -r $FOAM_APP/solvers/lagrangian/DPMFoam \
$WM_PROJECT_USER_DIR/applications/solvers/snowDPMFoam
```

Rename the `DPMFoam.C` file to `snowBedFoam.C`. In order to be able to compile the new application, the files must be modified in the `Make` directories. Change `snowDPMFoam/Make/files` to

```
DPMNewFoam.C

EXE = $(FOAM_USER_APPBIN)/snowBedFoam
```

and `snowDPMFoam/DPMTurbulenceModels/Make/files` to

```
DPMTurbulenceModels.C

LIB = $(FOAM_USER_LIBBIN)/libDPMTurbulenceModelsNew
```

The implementation stages described hereafter must be carried out in the user folder: by principle, the original OpenFOAM files in `$FOAM_SRC` should never be modified to ensure the correct operation of the software.

3.1 Particle statistical distribution models

Before the implementation of the snow transport equations, a new class must be created to integrate the different types of statistical distribution used for the sampling of the dimension, ejection angle and velocity of the particles (described in details in section S1.4. of Sharma et al. (2018), Supplementary Materials). This was achieved by using the OpenFOAM class distributionModels as a template. A total of three different statistical distributions are considered, namely: exponential, log-normal and normal. The subsequent step-by-step approach must be followed for their implementation:

1. In the `distributionModelsTriple` directory that was just created in the user folder, delete all the subdirectories except for the ones named `distributionModel` and `exponential`. The `Make` folder should also remain as it is essential for compilation;
2. Replace the term "distributionModel(s)" by "distributionModel(s)Triple" and "exponential" by "normalLogNormalExponential" in all the files and folders containing these instances;
3. In the file `distributionModelTriple.C`, delete the content of the Foam::distributionModels::distributionModel::check() Protected Member Function and replace the existing Member Functions by normalSample(), logNormalSample() and exponentialSample(). These functions should also be defined in `distributionModelTriple.H`, while the ones that were replaced should be removed. No changes should be brought to `distributionModelTripleNew.C`, except for the name as specified in the first step.

4. In the `normalLogNormalExponential.H` file, delete the lines found under the Private Data and Member Functions sections and add the definition of the normalSample, logNormalSample and exponentialSample functions;

5. In the `normalLogNormalExponential.C` file, add the mathematical expressions related to the three statistical distribution Member Functions used for the sampling of particle properties. In the Constructors section, keep only the distributionModelTriple (p) variable.

Figures 2 - 3 and 4 - 5 show the OpenFOAM `*.H` and `*.C` scripts for the new normalLogNormalExponential class, respectively. The equations corresponding to the statistical distributions are in Figure 5. As a final step, add the following line in the `Make/options` file of the `intermediateCRYOS` folder, after the expression `EXE_INC = \` :

```
-I../distributionModelsTriple/lnInclude \
```

In the same file, add the following after the line containing `LIB_LIBS = \`:

```
-L$(FOAM_USER_LIBBIN) \
-ldistributionModelTriple
```

These steps are needed for the correct compilation of the new class.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::normalLogNormalExponential

Description
    normalLogNormalExponential distribution model

SourceFiles
    normalLogNormalExponential.C

\*---------------------------------------------------------------------------*/

#ifndef normalLogNormalExponential_H
#define normalLogNormalExponential_H

#include "distributionModelTriple.H"

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

namespace Foam
{
namespace distributionModelsTriple
{

/*---------------------------------------------------------------------------*\
                        Class normalLogNormalExponential Declaration
\*---------------------------------------------------------------------------*/

class normalLogNormalExponential
:
    public distributionModelTriple
{

public:

```

Figure 2: `normalLogNormalExponential.H`, lines 1 to 57.

```
    //- Runtime type information
    TypeName("normalLogNormalExponential");


    // Constructors

        //- Construct from components
        normalLogNormalExponential(const dictionary& dict, cachedRandom& rndGen);

        //- Construct copy
        normalLogNormalExponential(const normalLogNormalExponential& p);

        //- Construct and return a clone
        virtual autoPtr<distributionModelTriple> clone() const
        {
            return autoPtr<distributionModelTriple>(new normalLogNormalExponential(*this));
        }


    //- Destructor
    virtual ~normalLogNormalExponential();


    // Member Functions

        //- Sample the normal distribution model
        virtual scalar normalSample(scalar mean_, scalar std_) const;

        //- Sample the lognormal distribution model
        virtual scalar logNormalSample(scalar mean_, scalar std_) const;

        //- Sample the exponential distribution model
        virtual scalar exponentialSample(scalar mean_, scalar std_) const;


};


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

} // End namespace distributionModelsTriple
} // End namespace Foam

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

#endif

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

Figure 3: `normalLogNormalExponential.H`, lines 58 to 105.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "normalLogNormalExponential.H"
#include "addToRunTimeSelectionTable.H"
#include "mathematicalConstants.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
{
    namespace distributionModelsTriple
    {
        defineTypeNameAndDebug(normalLogNormalExponential, 0);
        addToRunTimeSelectionTable(distributionModelTriple, normalLogNormalExponential, dictionary);
    }
}

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::distributionModelsTriple::normalLogNormalExponential::normalLogNormalExponential
(
    const dictionary& dict,
    cachedRandom& rndGen
)
:
    distributionModelTriple(typeName, dict, rndGen)
{
    check();
}


Foam::distributionModelsTriple::normalLogNormalExponential::normalLogNormalExponenti
```

Figure 4: `normalLogNormalExponential.C`, lines 1 to 55.

```
al(const normalLogNormalExponential& p)
:
    distributionModelTriple(p)
{}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

Foam::distributionModelsTriple::normalLogNormalExponential::~normalLogNormalExponential()
{}


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

Foam::scalar Foam::distributionModelsTriple::normalLogNormalExponential::normalSample(scalar mean_, scalar std_) const
{
    scalar rand1=rndGen_.sample01<scalar>();
    scalar rand2=rndGen_.sample01<scalar>();
    rand1=min(rand1+ROOTVSMALL,1.0);
    scalar val=mean_+std_*sqrt(-2.0*log(rand1))*cos(2.0*constant::mathematical::pi*rand2);
    return val;
}

Foam::scalar Foam::distributionModelsTriple::normalLogNormalExponential::logNormalSample(scalar mean_, scalar std_) const
{
    scalar s2 = log(1.0+pow(std_/mean_,2.0));
    scalar m = log(mean_) - 0.5*s2;
    scalar rand1=rndGen_.sample01<scalar>();
    scalar rand2=rndGen_.sample01<scalar>();
    rand1=min(rand1+ROOTVSMALL,1.0);
    scalar val=m+sqrt(s2)*sqrt(-2.0*log(rand1))*cos(2.0*constant::mathematical::pi*rand2);
    val=exp(val);
    return val;
}
Foam::scalar Foam::distributionModelsTriple::normalLogNormalExponential::exponentialSample(scalar mean_, scalar std_) const
{
    scalar rand1=rndGen_.sample01<scalar>();
    rand1=min(rand1,1.0-ROOTVSMALL);
    scalar val = -mean_*log(1.0-rand1);
    return val;
}
// ************************************************************************* //
```

Figure 5: `normalLogNormalExponential.C`, lines 56 to 96.

3.2 Submodel 1: aerodynamic entrainment

3.2.1 Copying the model template

The first step in the implementation of the aerodynamic lift submodel is to copy the `stochasticCollision` template directory found at:

```
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/submodels/...
Kinematic/StochasticCollision
```

Once this is done, follow the steps:

1. Rename the copied directory `BedAerodynamicLiftInjectionModel`;
2. Rename the `StochasticCollisionModel` subfolder by `BedAerodynamicLiftInjectionModel`. Do the same for all the files located inside. This will be the template of the class that was newly created;
3. Rename the `NoStochasticCollision` folder as well as all the files it contains by `NoBedAerodynamicLiftInjection`;
4. Inside all the files that were renamed in steps 2 and 3, replace the instances of the term "StochasticCollision" by the term "BedAerodynamicLiftInjection".
5. Create a copy of the new `BedAerodynamicLiftInjectionModel` subfolder and rename it `LogLawShearStress`. Inside the latter, replace every instance of the word "StochasticCollisionModel" by "LogLawShearStress" in the `*.C` and `*.H` files. It is in these scripts that the mathematical base for the aerodynamic lift model will be implemented;

At this stage, in the folder located at the path

```
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/submodels/...
Kinematic/BedAerodynamicLiftInjectionModel
```

You should have the list of directories shown in Figure 6.

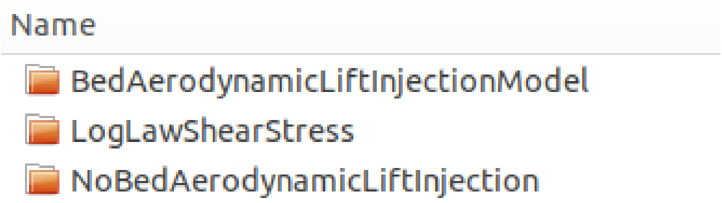
Name
BedAerodynamicLiftInjectionModel
LogLawShearStress
NoBedAerodynamicLiftInjection

Figure 6: Content of the `BedAerodynamicLiftInjectionModel` directory.

3.2.2 Description of the functions

The actual implementation of the aerodynamic entrainment equations can be performed now. Within the files of the three classes that were just created (LogLawShearStress - NoBedAerodynamicLiftinjection - BedAerodynamicLiftInjectionModel), replace the last term of the function

Foam::ClassName<CloudType>::collide(const scalar dt)

by bedAeroLiftInject(). This new function constitutes the base for the implementation of the aerodynamic entrainment equations within the LogLawShearStress class. No other change should be brought to BedAerodynamicLiftInjectionModel and NoBedAerodynamicLiftInjection. In the `LogLawShearStress` directory, simultaneously open the `LogLawShearStress.H` and `LogLawShearStress.C` files. In the new bedAeroLiftInject() function, erase all the lines that were related to the original collide function. Table 1 summarizes the Protected Member Functions involved in the LogLawShearStress model and their utility.

Function	**Utility**
bedAeroLiftInject	Main routine of the script. For every cell, the surface shear stress is computed and the number of lifted particles determined accordingly.
normalInject	Accounts for the vertical shift of the lifted particles and adds them in the domain (see Sharma et al. (2018), section S1.4.).

Table 1: Functions implemented in the aerodynamic entrainment model and their utility.

Linking the mathematical expressions from Section 2 to their corresponding segments of the code, the shear stress threshold (Eq.1) is implemented within the bedAeroLiftInject function at line *131* (Fig.12). The shear stress found at the surface can be computed in two ways (to be specified by the user in the `kinematicCloudProperties` file of the case directory), either by applying the logarithmic law (Eq.2 - line *124*, Fig.12) or via the modelled turbulent kinematic viscosity (Eq.3 - line *128*, Fig.12). The difference between the threshold and actual surface shear stress is used to determine the number of aerodynamically lifted particles N_{ae}, at line *135* (Eq.4 - Fig.12). From line *141* on, the code is related to the random sampling of the particle properties as well as to the injection of particles through the call of the respective functions (Table 1).

3.2.3 OpenFOAM scripts

The scripts related to our BedAerodynamicLiftInjectionModel Lagrangian submodel and more especially to the LogLawShearStress class are given below. Figures 7 to 9 present extracts from the `LogLawShearStress.H` file which contain data types and function definitions. The commented lines briefly describe the variables that are employed in the model. Figures 10 to 16 constitute the parts of the `LogLawShearStress.C` script where the equations for aerodynamic lift were included.

```
1   /*---------------------------------------------------------------------------*\
2     =========                 |
3     \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
4      \\    /   O peration     |
5       \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
6        \\/     M anipulation  |
7   -------------------------------------------------------------------------------
8   License
9       This file is part of OpenFOAM.
10
11      OpenFOAM is free software: you can redistribute it and/or modify it
12      under the terms of the GNU General Public License as published by
13      the Free Software Foundation, either version 3 of the License, or
14      (at your option) any later version.
15
16      OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
17      ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
18      FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
19      for more details.
20
21      You should have received a copy of the GNU General Public License
22      along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.
23
24  Class
25      Foam::LogLawShearStress
26
27  Description
28      Aerodynamic entrainment of bed particles
29
30
31  \*---------------------------------------------------------------------------*/
32
33  #ifndef LogLawShearStress_H
34  #define LogLawShearStress_H
35
36  #include "BedAerodynamicLiftInjectionModel.H"
37
38  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
39
40  namespace Foam
41  {
42  /*---------------------------------------------------------------------------*\
43                        Class LogLawShearStress Declaration
44  \*---------------------------------------------------------------------------*/
45
46  template<class CloudType>
47  class LogLawShearStress
48  :
49      public BedAerodynamicLiftInjectionModel<CloudType>
50  {
51      // Private Data
52
53      //- Mean particle diameter
54          scalar dm_;
55
56          //- Minimum number of particles per parcel
57          scalar pppMin_;
```

Figure 7: `LogLawShearStress.H`, lines 1 to 57.

```

        //- Std deviation of particle diameter
        scalar ds_;

        //- Std deviation of particle diameter
        scalar d_max_;

        //- Std deviation of particle diameter
        scalar d_min_;

        //- Aerodynamic roughness length
        scalar z0_;

        //Start of Activation (launch of saltation)
        scalar SOA_;

        //A constant for shear stress threshold computation
        scalar Acst_;

        //- Number of parcels aerodynamically lifted
        volScalarField nAeroLift_;

        //- Patch name
        const word patchName_;

        //- Flag to compute surface shear stress with log-law
        Switch tauLogLaw_;

        //- Patch ID
        const label patchId_;


protected:

    // Protected Data

        //- Convenience typedef to the cloud's parcel type
        typedef typename CloudType::parcelType parcelType;

        //- Parcel size distribution model
        const autoPtr<distributionModelsTriple::distributionModelTriple> sizeDistributionTriple_;

    // Protected Member Functions

        //- Main aerodynamic entrainment routine
        virtual void bedAeroLiftInject();
        void normalInject(const vector& U_NewP, const vector& coorf, const vector& coorfr, const scalar& d_g, const scalar& nParticle);

        //- Aerodynamically lifted parcel type label - id assigned to identify parcel for
        //  post-processing.
        label aeroLiftParcelType_;

public:

```

Figure 8: `LogLawShearStress.H`, lines 58 to 111.

```
    //- Runtime type information
    TypeName("logLawShearStress");


    // Constructors

        //- Construct from dictionary
        LogLawShearStress
        (
            const dictionary& dict,
            CloudType& cloud,
            const word& modelName = typeName
        );

        //- Construct copy
        LogLawShearStress(LogLawShearStress<CloudType>& cm);

        //- Construct and return a clone
        virtual autoPtr<BedAerodynamicLiftInjectionModel<CloudType> > clone() //const
        {
            return autoPtr<BedAerodynamicLiftInjectionModel<CloudType> >
            (
                new LogLawShearStress<CloudType>(*this)
            );
        }


    //- Destructor
    virtual ~LogLawShearStress();

    // Member Functions
};


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

} // End namespace Foam

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

#ifdef NoRepository
#   include "LogLawShearStress.C"
#endif

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

#endif

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

Figure 9: `LogLawShearStress.H`, lines 112 to 161.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "LogLawShearStress.H"
#include "mathematicalConstants.H"
#include "meshTools.H"
#include "polyMeshTetDecomposition.H"
#include "turbulenceModel.H"

using namespace Foam::constant::mathematical;

// * * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * //
template<class CloudType>
void Foam::LogLawShearStress<CloudType>::bedAeroLiftInject()
{
    const fvMesh& mesh = this->owner().mesh();
    if(mesh.time().value() < SOA_)
        {
            // not in the time range: go back
            return;
        }

    ////////////////////////////////////////////////////////////////////////

    const volVectorField& U = this->owner().U();
    const volScalarField& rho = this->owner().rho();

    /////////////// SHEAR STRESS COMPUTATION: GENERAL METHOD
    const objectRegistry& obr = this->owner().mesh();
    const turbulenceModel& turbModel =obr.lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            this->owner().U().group()
```

Figure 10: `LogLawShearStress.C`, lines 1 to 57.

```
        )
    );
    volScalarField nuEff(turbModel.nuEff());

    volScalarField uFric
    (
        IOobject
        (
            "uFric",
            this->owner().db().time().timeName(),
            this->owner().mesh(),
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        this->owner().mesh(),
        dimensionedScalar("uFric", dimVelocity, 0.0)
    );

    uFric.boundaryField()[patchId_] =
                                sqrt
                                (
                                    nuEff.boundaryField()[patchId_]
                                   *mag(U.boundaryField()[patchId_].snGrad())
                                );
    const scalarField& uFp = uFric.boundaryField()[patchId_];
    const scalarField& y = turbModel.y()[patchId_];
    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchId_];
    const scalarField magUp(mag(Uw.patchInternalField() - Uw));

    ////////////////////////////////////////////////////////////////////////


    //To access the mesh information for the boundary at target patch patchId_
    const polyPatch& cPatch = mesh.boundaryMesh()[patchId_];

     //List of cells close to a boundary
    const labelUList& faceCells = cPatch.faceCells();

    forAll(faceCells, faceI)
    {//1
        label cellInd = faceCells[faceI];

        //COMPUTE SURFACE SHEAR STRESS FROM EULERIAN GRID FOR A GIVEN CELL
        vector coorC = mesh.C()[cellInd];
        vector coorf = mesh.Cf().boundaryField()[patchId_][faceI];

        const vector UCell=U[cellInd];
        const scalar rhoCell=rho[cellInd];

        vector n = -mesh.Sf().boundaryField()[patchId_][faceI]/mesh.magSf().boundaryField()[patchId_][faceI];
        vector Un  = (UCell & n)*n;

        vector Ut1  = UCell - Un;
        vector t1 = Ut1/mag(Ut1);

```

Figure 11: `LogLawShearStress.C`, lines 58 to 112.

```
        vector t2 = t1^n; //... normal to impacting plane

        scalar cellCentreDistanceToWall = mag((coorC-coorf) & n);

        // OPTIONS FOR SHEAR STRESS COMPUTATION:
        scalar oldMassCheckPatterns = this->owner().massCheckPatterns().oldTime().boundaryField()[patchId_][faceI];

        scalar tauSurface;

        if(tauLogLaw_)
        {
            tauSurface = rhoCell*pow((0.41*mag(Ut1)/(log(cellCentreDistanceToWall/z0_))),2); //log law method, z corresponds to height at center of the face
        }
        else
        {
            tauSurface = rhoCell*pow(uFp[faceI],2); //To estimate the shear stress as the main method consistent with all the wall function used for nut
        }

        scalar tauThresh = (Acst_*Acst_)*9.81*dm_*(this->owner().constProps().rho0()-rhoCell); //Shear Stress Threshold (Bagnold).
        scalar tauExcess = max(0.0,(tauSurface-tauThresh));

        scalar cellArea = mesh.magSf().boundaryField()[patchId_][faceI];
        scalar nEntrain = 1.5*tauExcess/(8.0*constant::mathematical::pi*pow(dm_,2.0)); //Sharma's paper, p.3 - Nae variable
        nEntrain=nEntrain*cellArea*this->owner().db().time().deltaTValue();

        nAeroLift_.boundaryField()[patchId_][faceI] += nEntrain;
        scalar entrainment = nAeroLift_.boundaryField()[patchId_][faceI];

        if(entrainment>pppMin_)
        {//2
            scalar tempMass = entrainment*this->owner().constProps().rho0()*constant::mathematical::pi*pow(dm_,3.0)/6.0;
            scalar depMass = (this->owner().massDeposition().boundaryField()[patchId_][faceI])*cellArea;

            if(tempMass>depMass)
            {
                tempMass=depMass;
            }

            if(tempMass>0) //If there is still mass to lift up.
            {//3
                scalar h_ang = 0.0;

                scalar d_g = sizeDistributionTriple_->logNormalSample(dm_,ds_);
                d_g = min(d_max_,max(d_g,d_min_));
```

Figure 12: `LogLawShearStress.C`, lines 113 to 156.

```

                scalar mass_g =this->owner().constProps().rho0()*constant::mathematical::pi*pow(d_g,3)/6.0;
                scalar entrainmentModified=tempMass/mass_g;
                label np1 = label(entrainmentModified/pppMin_)+1;  //This is because last parcel won't be filled up to maximum.

                scalar slope = 0.0;
                for(label ip1=1; ip1<=np1; ip1++)
                {//4
                    scalar nParticle=pppMin_;
                    if(ip1==np1) //If the number of parcels is not the last one, which might not be completely full
                    {
                        nParticle=entrainmentModified-(np1-1)*pppMin_;
                    }

                    scalar mean = (75.0-55.0*(1.0-exp(-d_g/(175e-6))))/180.0*constant::mathematical::pi;
                    scalar std = 15.0/180.0*constant::mathematical::pi;

                    //Vertical angle
                    scalar v_ang = sizeDistributionTriple_->logNormalSample(mean,std);
                    v_ang = min(constant::mathematical::piByTwo, max(-constant::mathematical::piByTwo, v_ang+slope));

                    scalar vel_fric    = sqrt(tauSurface/rhoCell);
                    mean        = 3.5*vel_fric;
                    std         = 2.5*vel_fric;
                    scalar e_vel = sizeDistributionTriple_->logNormalSample(mean,std);

                    vector Un_NewP = (e_vel*sin(v_ang))*n;
                    vector Ut1_NewP = (e_vel*cos(v_ang)*cos(h_ang))*t1;
                    vector Ut2_NewP = (e_vel*cos(v_ang)*sin(h_ang))*t2;
                    vector U_NewP = Un_NewP+Ut1_NewP+Ut2_NewP;

                    normalInject(U_NewP,coorC,coorf+(4.0*dm_)*n, d_g, nParticle);

                }//4
                this->owner().massDeposition().boundaryField()[patchId_][faceI] -= tempMass/cellArea;
                this->owner().massCheckPatterns().boundaryField()[patchId_][faceI] -= tempMass/cellArea;
                this->owner().surfaceUfric().boundaryField()[patchId_][faceI]  = uFp[faceI];

            }//3
        nAeroLift_.boundaryField()[patchId_][faceI] = 0.0;
        }//2
        this->owner().massDepRate().boundaryField()[patchId_][faceI] = ((this->owner().massCheckPatterns().boundaryField()[patchId_][faceI])-oldMassCheckPatterns)/(this->owner().db().time().deltaTValue());
    }//1
}

```

Figure 13: `LogLawShearStress.C`, lines 157 to 201.

```
template<class CloudType>
void Foam::LogLawShearStress<CloudType>::normalInject(const vector& U_NewP, const vector& coorf, const vector& coorfr, const scalar& d_g, const scalar& nParticle)
{
    label cellI = -1;
    label tetFaceI = -1;
    label tetPtI = -1;
    vector pos = coorf;
    label posInList = -1;
    this->owner().mesh().findCellFacePt
    (
        pos,
        cellI,
        tetFaceI,
        tetPtI
    );

    if (cellI > -1)
    {
        parcelType* pPtr = new parcelType(this->owner().mesh(), coorfr, cellI, tetFaceI, tetPtI);

        //Check/set new parcel thermo properties
        this->owner().setParcelThermoProperties(*pPtr, 0.0);

        pPtr->d()=d_g; //assigning the diameter, same for all particles in the parcel

        //Check/set new parcel injection properties
        this->owner().checkParcelProperties(*pPtr, this->owner().mesh().time().deltaTValue(), false);
        pPtr->nParticle()=nParticle;

        pPtr->U()=U_NewP; //assigning the ejection linear velocity
        pPtr->typeId() = aeroLiftParcelType_;

        // Apply corrections to position for 2-D cases
        meshTools::constrainToMeshCentre(this->owner().mesh(), pPtr->position());

        // Apply correction to velocity for 2-D cases
        meshTools::constrainDirection
        (
            this->owner().mesh(),
            this->owner().mesh().solutionD(),
            pPtr->U()
        );

        this->owner().addParticle(pPtr);
    }
    else
    {
        Info << "ERROR: The cell index is negative ... coorf/cellI/tetFaceI/tetPtI" << coorf << ' ' << cellI << ' ' << tetFaceI <<  ' ' << tetPtI << endl;
    }
}

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

```

Figure 14: `LogLawShearStress.C`, lines 202 to 254.

```
template<class CloudType>
Foam::LogLawShearStress<CloudType>::LogLawShearStress
(
    const dictionary& dict,
    CloudType& owner,
    const word& modelName
)
:
    BedAerodynamicLiftInjectionModel<CloudType>(dict, owner, modelName),
    dm_(0.0),
    ds_(0.0),
    pppMin_(0.0),
    d_max_(0.0),
    d_min_(0.0),
    z0_(0.0),
    SOA_(0.0),
    Acst_(0.0),
    aeroLiftParcelType_
    (
        this->coeffDict().lookupOrDefault("aeroLiftParcelType", 2)
    ),
    nAeroLift_
    (
        IOobject
        (
            this->owner().name() + ":nAeroLift",
            this->owner().db().time().timeName(),
            this->owner().mesh(),
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        this->owner().mesh(),
        dimensionedScalar("zero", dimless, 0.0), //Number of particles already entrained
        zeroGradientFvPatchScalarField::typeName //For post-processing purposes
    ),
    sizeDistributionTriple_
    (
        distributionModelsTriple::distributionModelTriple::New
        (
            this->coeffDict().subDict("sizeDistributionTriple"),
            this->owner().rndGen()
        )
    ),
    patchName_(this->coeffDict().lookup("aerodynamicLiftPatch")),
    tauLogLaw_(this->coeffDict().lookupOrDefault("tauLogLaw", false)),
    patchId_(this->owner().mesh().boundaryMesh().findPatchID(patchName_))
{
    if (patchId_ < 0)
    {
        FatalErrorIn
        (
            "Foam::LogLawShearStress<CloudType>::LogLawShearStress"
            "("
                "const dictionary& dict,"
                "CloudType& owner,"
                "const word& modelName"
```

Figure 15: `LogLawShearStress.C`, lines 255 to 310.

```
            ")"
        )   << "Requested patch " << patchName_ << " not found" << nl
            << "Available patches are: " << this->owner().mesh().boundaryMesh().names()
            << nl << exit(FatalError);
    }

    dm_ = this->coeffDict().lookupOrDefault("dm", 0.00026);
    ds_ = this->coeffDict().lookupOrDefault("ds", 0.00013);
    pppMin_= this->coeffDict().lookupOrDefault("pppMin", 1000);
    d_max_= this->coeffDict().lookupOrDefault("d_max", 0.002);
    d_min_= this->coeffDict().lookupOrDefault("d_min", 0.00005);
    z0_= this->coeffDict().lookupOrDefault("z0",0.0001);
    SOA_= this->coeffDict().lookupOrDefault("SOA",100.0);
    Acst_= this->coeffDict().lookupOrDefault("Acst",0.1);
}


template<class CloudType>
Foam::LogLawShearStress<CloudType>::LogLawShearStress
(
    LogLawShearStress<CloudType>& cm
)
:
    BedAerodynamicLiftInjectionModel<CloudType>(cm),
    sizeDistributionTriple_(cm.sizeDistributionTriple_().clone().ptr()),
    dm_(cm.dm_),
    ds_(cm.ds_),
    pppMin_(cm.pppMin_),
    d_max_(cm.d_max_),
    d_min_(cm.d_min_),
    z0_(cm.z0_),
    SOA_(cm.SOA_),
    Acst_(cm.Acst_),
    aeroLiftParcelType_(cm.aeroLiftParcelType_),
    nAeroLift_(cm.nAeroLift_),
    patchName_(cm.patchName_),
    tauLogLaw_(cm.tauLogLaw_),
    patchId_(cm.patchId_)
{

}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

template<class CloudType>
Foam::LogLawShearStress<CloudType>::~LogLawShearStress()
{}


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

```

Figure 16: `LogLawShearStress.C`, lines 311 to 362.

3.2.4 Linking libraries

As a final step, the new BedAerodynamicLiftInjectionModel submodel needs to be linked to the kinematicCloud and parcel classes for compilation purposes. To do so, go to the `KinematicCloud` directory located at:

```
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/...
clouds/Templates/KinematicCloud
```

Inside this directory, open the `KinematicCloud.C`, `KinematicCloud.H` and `KinematicCloudI.H` files. There, search for the "stochasticCollision" term and copy each line of code containing it, but with replacing every instance of it by "BedAerodynamicLiftInjection". The new submodel also needs to be linked to the parcel object. Using the same first line than the previous path, go to

```
...parcels/include
```

and create a file `makeKinematicParcelBedAerodynamicLiftInjectionModels.H` similarly to the one related to the stochastic collision submodel, named `makeParcelStochasticCollisionModels.H`.

Next, go to

```
...parcels/derived
```

and in the `makeBasic*ParcelSubmodels.C` files of each subfolder, add the reference to the *.H file created above, just as the other submodels. Because the StochasticCollision model served as a template for our aerodynamic lift model, they should appear in the exact same places: this can provide guidance for adding the BedAerodynamicLiftInjectionModel submodel in the appropriate files.

3.3 Submodel 2: rebound-splash entrainment

3.3.1 Copying the template

The first step in the implementation of the rebound-splash submodel is to make a copy of the localInteraction submodel located at the following path:

```
$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/submodels/...
Kinematic/PatchInteractionModel/LocalInteraction
```

Once the folder has been copied, follow the subsequent steps:

1. Rename the `LocalInteraction` directory by `LocalInteractionReboundingSplashing`;
2. In the the files contained within, replace every instance of the term "LocalInteraction" by "LocalInteractionStickReboundSplash", and the term "patchInteraction" by "patchInteractionStickReboundSplash";

3. In the file `LocalInteractionStickReboundSplash.C`, go to the correct() function and copy the switch case called itRebound. Replace the term "itRebound" by "itStickReboundSplash". It is in this case that the rebounding-splashing equations are implemented.

Both the rebound and splash-related equations are included in this submodel.

3.3.2 Implementation of rebound equations

The first part of the implemented submodel is related to the rebounding of grains. It is described in section S.1.4.2 from the work of Sharma et al. (2018) and relates to the definition of the probability of rebound defined in Eq.5. It is implemented at the beginning of the itStickReboundSplash case switch, at lines *312* - *339* of the `LocalInteractionStickReboundSplash.C` file (see Fig. 25 and 26). The adopted approach is that once a particle gets close to the boundary, a random number is generated and compared to P_r. If it is within the probability range, the particle is kept and assumed to rebound. If not, it is removed from the numerical domain.

3.3.3 Implementation of splash equations

The second part of the submodel is related to the ejection (splashing) of grains due to the effect of particles impacting the surface. It is implemented after the rebound of particles in the itStickReboundSplash case, at lines *340* - *442*. Line *358* (Fig.26) of the code refers to the energy-related number of ejected particles N_E (Eq.6) while line *359* refers to the momentum-related one, N_M (Eq.7). Both of these equations are used to determine the number of splashed particles (line *360*). The lines located after relate to the random sampling of the particle properties and the generation of parcels, until line *442* (Fig.28).

3.3.4 OpenFOAM scripts

Figures 17 to 19 display the content of the `LocalInteractionStickReboundSplash.H` file which defines the data and functions used in the script. On the other hand, Figures 20 to 30 show the `LocalInteractionStickReboundSplash.C` script. The variables employed in the code should also be added within the `patchInteractionStickReboundSplashData.*` files (not displayed in this tutorial).

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::LocalInteractionStickReboundSplash

Description
    Patch interaction specified on a patch-by-patch basis

\*---------------------------------------------------------------------------*/

#ifndef LocalInteractionStickReboundSplash_H
#define LocalInteractionStickReboundSplash_H

#include "PatchInteractionModel.H"
#include "patchInteractionStickReboundSplashDataList.H"
#include "Switch.H"
#include "distributionModelTriple.H"
#include "Random.H"


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

namespace Foam
{
/*---------------------------------------------------------------------------*\
                      Class LocalInteractionStickReboundSplash Declaration
\*---------------------------------------------------------------------------*/

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

        //- List of participating patches
```

Figure 17: `LocalInteractionStickReboundSplash.H`, lines 1 to 57.

```
        const patchInteractionStickReboundSplashDataList patchData_;


        // Counters for particle fates

            //- Number of parcels escaped
            List<label> nEscape_;

            //- Mass of parcels escaped
            List<scalar> massEscape_;

            //- Number of parcels stuck to patches
            List<label> nStick_;

            //- Mass of parcels stuck to patches
            List<scalar> massStick_;

        //- Flag to output data as fields
        Switch writeFields_;

        //- Mass escape field
        autoPtr<volScalarField> massEscapePtr_;

        //- Mass stick field
        autoPtr<volScalarField> massStickPtr_;

        //- Mass deposition field
        //autoPtr<volScalarField> massDepositionPtr_;

protected:

        //- Convenience typedef to the cloud's parcel type
        typedef typename CloudType::parcelType parcelType;

        //- Parcel size distribution model
        const autoPtr<distributionModelsTriple::distributionModelTriple> sizeDistributionTriple_;

public:

    //- Runtime type information
    TypeName("localInteractionStickReboundSplash");


    // Constructors

        //- Construct from dictionary
        LocalInteractionStickReboundSplash(const dictionary& dict, CloudType& owner);

        //- Construct copy from owner cloud and patch interaction model
        LocalInteractionStickReboundSplash(const LocalInteractionStickReboundSplash<CloudType>& pim);

        //- Construct and return a clone using supplied owner cloud
        virtual autoPtr<PatchInteractionModel<CloudType> > clone() const
        {
            return autoPtr<PatchInteractionModel<CloudType> >
```

Figure 18: `LocalInteractionStickReboundSplash.H`, lines 58 to 112.

```
            (
                new LocalInteractionStickReboundSplash<CloudType>(*this)
            );
        }


    //- Destructor
    virtual ~LocalInteractionStickReboundSplash();


    // Member Functions

        //- Return access to the massEscape field
        volScalarField& massEscape();

        //- Return access to the massStick field
        volScalarField& massStick();

        //- Return access to the massDeposition field
        //volScalarField& massDeposition();

        //- Apply velocity correction
        //  Returns true if particle remains in same cell
        virtual bool correct
        (
            typename CloudType::parcelType& p,
            const polyPatch& pp,
            bool& keepParticle,
            const scalar trackFraction,
            const tetIndices& tetIs
        );

        // I-O

            //- Write patch interaction info to stream
            virtual void info(Ostream& os);
};


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

} // End namespace Foam

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

#ifdef NoRepository
#   include "LocalInteractionStickReboundSplash.C"
#endif

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

#endif

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

Figure 19: `LocalInteractionStickReboundSplash.H`, lines 113 to 166.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2014 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "LocalInteractionStickReboundSplash.H"
#include "mathematicalConstants.H"
#include "meshTools.H"

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

template<class CloudType>
Foam::LocalInteractionStickReboundSplash<CloudType>::LocalInteractionStickReboundSplash
(
    const dictionary& dict,
    CloudType& cloud
)
:
    PatchInteractionModel<CloudType>(dict, cloud, typeName),
    patchData_(cloud.mesh(), this->coeffDict()),
    nEscape_(patchData_.size(), 0),
    massEscape_(patchData_.size(), 0.0),
    nStick_(patchData_.size(), 0),
    massStick_(patchData_.size(), 0.0),
    writeFields_(this->coeffDict().lookupOrDefault("writeFields", true)),
    massEscapePtr_(NULL),
    massStickPtr_(NULL),
    sizeDistributionTriple_
    (
        distributionModelsTriple::distributionModelTriple::New
        (
            this->coeffDict().subDict("sizeDistributionTriple"),
            this->owner().rndGen()
        )
    )
{
```

Figure 20: `LocalInteractionStickReboundSplash.C`, lines 1 to 56.

```
 57        if (writeFields_)
 58        {
 59            word massEscapeName(this->owner().name() + ":massEscape");
 60            word massStickName(this->owner().name() + ":massStick");
 61            Info<< "    Interaction fields will be written to " << massEscapeName << ","
 62            << " and " << massStickName << endl;
 63
 64            (void)massEscape();
 65            (void)massStick();
 66        }
 67        else
 68        {
 69            Info<< "    Interaction fields will not be written" << endl;
 70        }
 71
 72        // check that interactions are valid/specified
 73        forAll(patchData_, patchI)
 74        {
 75            const word& interactionTypeName =
 76                patchData_[patchI].interactionTypeName();
 77            const typename PatchInteractionModel<CloudType>::interactionType& it =
 78                this->wordToInteractionType(interactionTypeName);
 79
 80            if (it == PatchInteractionModel<CloudType>::itOther)
 81            {
 82                const word& patchName = patchData_[patchI].patchName();
 83                FatalErrorIn("LocalInteractionStickReboundSplash(const dictionary&, CloudType&)")
 84                    << "Unknown patch interaction type "
 85                    << interactionTypeName << " for patch " << patchName
 86                    << ". Valid selections are:"
 87                    << this->PatchInteractionModel<CloudType>::interactionTypeNames_
 88                    << nl << exit(FatalError);
 89            }
 90        }
 91    }
 92
 93
 94    template<class CloudType>
 95    Foam::LocalInteractionStickReboundSplash<CloudType>::LocalInteractionStickReboundSplash
 96    (
 97        const LocalInteractionStickReboundSplash<CloudType>& pim
 98    )
 99    :
100        PatchInteractionModel<CloudType>(pim),
101        patchData_(pim.patchData_),
102        nEscape_(pim.nEscape_),
103        massEscape_(pim.massEscape_),
104        nStick_(pim.nStick_),
105        massStick_(pim.massStick_),
106        writeFields_(pim.writeFields_),
107        massEscapePtr_(NULL),
108        massStickPtr_(NULL),
109        sizeDistributionTriple_(pim.sizeDistributionTriple_().clone().ptr())
110    {}
111
```

Figure 21: `LocalInteractionStickReboundSplash.C`, lines 57 to 111.

```

// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

template<class CloudType>
Foam::LocalInteractionStickReboundSplash<CloudType>::~LocalInteractionStickReboundSplash()
{}


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

template<class CloudType>
Foam::volScalarField& Foam::LocalInteractionStickReboundSplash<CloudType>::massEscape()
{
    if (!massEscapePtr_.valid())
    {
        const fvMesh& mesh = this->owner().mesh();

        massEscapePtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + ":massEscape",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimMass, 0.0)
            )
        );
    }

    return massEscapePtr_();
}


template<class CloudType>
Foam::volScalarField& Foam::LocalInteractionStickReboundSplash<CloudType>::massStick()
{
    if (!massStickPtr_.valid())
    {
        const fvMesh& mesh = this->owner().mesh();

        massStickPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + ":massStick",
                    mesh.time().timeName(),
```

Figure 22: `LocalInteractionStickReboundSplash.C`, lines 112 to 165.

```
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimMass, 0.0)
            )
        );
    }

    return massStickPtr_();
}

template<class CloudType>
bool Foam::LocalInteractionStickReboundSplash<CloudType>::correct
(
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const scalar trackFraction,
    const tetIndices& tetIs
)
{
    label patchI = patchData_.applyToPatch(pp.index());

    if (patchI >= 0)
    {
        vector& U = p.U();
        bool& active = p.active();

        typename PatchInteractionModel<CloudType>::interactionType it =
            this->wordToInteractionType
            (
                patchData_[patchI].interactionTypeName()
            );

        switch (it)
        {
            case PatchInteractionModel<CloudType>::itEscape:
            {
                scalar dm = p.mass()*p.nParticle();

                keepParticle = false;
                active = false;
                U = vector::zero;
                nEscape_[patchI]++;
                massEscape_[patchI] += dm;
                if (writeFields_)
                {
                    label pI = pp.index();
                    label fI = pp.whichFace(p.face());
                    massEscape().boundaryField()[pI][fI] += dm;
                }
                break;
            }
            case PatchInteractionModel<CloudType>::itStick:
            {
```

Figure 23: `LocalInteractionStickReboundSplash.C`, lines 166 to 222.

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

                keepParticle = true;
                active = false;
                U = vector::zero;
                nStick_[patchI]++;
                massStick_[patchI] += dm;
                if (writeFields_)
                {
                    label pI = pp.index();
                    label fI = pp.whichFace(p.face());
                    massStick().boundaryField()[pI][fI] += dm;
                }
                break;
            }
            case PatchInteractionModel<CloudType>::itRebound:
            {
                keepParticle = true;
                active = true;

                vector nw;
                vector Up;

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

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

                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;
            }
            ///////////////////////////////////////////////////////////////////////////////////
            //CASE SWITCH FOR REBOUND-SPLASH OF SNOW GRAINS
            case PatchInteractionModel<CloudType>::itStickReboundSplash:
            {
                vector nw;
                vector Up;
                this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

                const fvMesh& mesh = this->owner().mesh();
                cachedRandom& ranGen = this->owner().rndGen();
                label pI = pp.index();
                label fI = pp.whichFace(p.face());
```

Figure 24: `LocalInteractionStickReboundSplash.C`, lines 223 to 277.

```
                 scalar cellArea = mesh.magSf().boundaryField()[pI][fI];
                 label cellInd = mesh.faceOwner()[fI];

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

                 vector n = -nw;
                 vector Un  = (U & n)*n;

                 vector Ut1  = U - Un;
                 vector t1 = Ut1/(mag(Ut1)+ROOTVSMALL);

                 vector t2 = t1^n;

                 // Impact Properties
                 scalar i_vel = mag(U);
                 scalar n_impact = p.nParticle();             //number of particles in the parcel
                 scalar pMassParcel=p.nParticle()*p.mass();   //mass of the parcel
                 scalar i_ene=0.5*pMassParcel*pow(i_vel,2);
                 scalar i_mom=pMassParcel*i_vel;




                 scalar i_ang1 = atan(mag(Un)/ (mag(Ut1)+ROOTVSMALL) );


                 scalar slope = 0.0;

                // i_ang2 is impacting angle with respect to bed surface, or vang of impacting particle
                // To get the horizontal angle with respect to impacting plane, must set i_ang2

                 i_ang2=0.0;

                  // PART I: REBOUNDING OF GRAINS
                 scalar prob_reb= 0.9*(1.0-exp(-2.0*i_vel)); //Probability of rebound for particles
                 scalar rand = ranGen.sample01<scalar>();
                 if(rand<prob_reb && (U & n)<=0. )
                 {
                     //Sampling ejection angle from distribution
                     scalar r_vel=0.5*i_vel;
                     scalar mean= 45.0/180*constant::mathematical::pi;
                     scalar v_ang = sizeDistributionTriple_->exponentialSample(mean,0.0);
                     v_ang  = min(constant::mathematical::piByTwo, max(-constant::mathematical::piByTwo, v_ang+slope));
                     Un = (r_vel*sin(v_ang))*n;
                     Ut1 = (r_vel*cos(v_ang))*t1;
                     U = Ut1+Un;
                     // Return velocity to global space
                     U += Up;
                     keepParticle = true;
                     active = true;
```

Figure 25: `LocalInteractionStickReboundSplash.C`, lines 278 to 328.

```
329                 }
330                 else
331                 {
332                     //If probability < probability of rebound: back to the snow bed
333                     this->owner().massDeposition().boundaryField()[pI][fI] += (pMassParcel)/cellArea;
334                     this->owner().massCheckPatterns().boundaryField()[pI][fI] += (pMassParcel)/cellArea;
335                     keepParticle = false;
336                     active = false;
337                     U = vector::zero;
338                 }
339 
340                 // PART II: SPLASHING OF GRAINS
341                 scalar epsilonf_= 0.96*(1.0-prob_reb*patchData_[patchI].epsilonr());
342                 scalar av_d3= pow(patchData_[patchI].dm()+pow(patchData_[patchI].ds(),2.0)/patchData_[patchI].dm(),3.0);
343                 scalar sd_d3 = av_d3*sqrt(pow(1.0+pow(patchData_[patchI].ds()/patchData_[patchI].dm(),2.0),9.0)-1.0);
344                 scalar av_vel = 0.25*pow(i_vel,0.3);
345                 scalar av_vel2 = 2.0*pow(av_vel,2.0);
346                 scalar av_mass = p.rho()*constant::mathematical::pi/6.0*av_d3;
347                 scalar sd_vel  = av_vel;
348                 scalar sd_vel2 = 2.0*sqrt(5.0)*pow(av_vel,2.0);
349                 scalar sd_mass = p.rho()*constant::mathematical::pi/6.0*sd_d3;
350 
351                 scalar cos_a = 0.75;
352                 scalar cos_b  = 0.96;
353                 scalar cos_i = cos(i_ang1);
354 
355                 scalar av_massvel = av_mass*av_vel*cos_a*cos_b + patchData_[patchI].corrm()*sd_mass*sd_vel;
356                 scalar av_massvel2 = av_mass*av_vel2 + patchData_[patchI].corre()*sd_mass*sd_vel2;
357                 // Number of ejected particles based on energy/momentum conservation (Comola & Lehning, 2017)
358                 scalar n_splash1 = i_ene*(1.0-prob_reb*patchData_[patchI].epsilonr() - epsilonf_)/(0.5*av_massvel2+patchData_[patchI].bEne()+ROOTVSMALL);
359                 scalar n_splash2 = i_mom*cos_i*(1.0 - prob_reb*patchData_[patchI].mur() - patchData_[patchI].muf())/(av_massvel+ROOTVSMALL);
360                 scalar n_splash = min(n_splash1,n_splash2);
361 
362                 // Sampling of particle properties
363                 // Condition: number of splashed particles >= number of impacting particles (model choice)
364                 if(n_splash>=n_impact)
365                 {
366                     // Taking into account an unfilled last parcel
367                     label np1 = label(n_splash/patchData_[patchI].pppMax())+1;
368 
369                     for(label ip1=1; ip1<=np1; ip1++)
370                     {
371 
```

Figure 26: `LocalInteractionStickReboundSplash.C`, lines 329 to 371.

```
                        scalar d_g = sizeDistributionTriple_->logNormalSample(patchData_[patchI].dm(),patchData_[patchI].ds());
                        d_g = min(patchData_[patchI].d_max(),max(d_g,patchData_[patchI].d_min()));
                        scalar mass_g = p.rho()*constant::mathematical::pi*pow(d_g,3)/6.0;

                        scalar dep_mass = (this->owner().massDeposition().boundaryField()[pI][fI])*cellArea;
                        scalar temp_mass = 0.0;

                        if(ip1!=np1)
                        {
                            temp_mass = patchData_[patchI].pppMax()*mass_g;
                        }
                        else   // Condition for unfilled last parcel
                        {
                            temp_mass = (n_splash-(np1-1)*patchData_[patchI].pppMax())*mass_g;
                        }

                        if(temp_mass > dep_mass)
                        {
                            temp_mass = dep_mass;
                        }

                        if(temp_mass > 0.0) //If negative, no more snow at the surface
                        {
                        parcelType* pPtr = new parcelType(mesh, p.position(),p.cell(), p.tetFace(), p.tetPt());

                        // Check/set new parcel thermo properties
                        this->owner().setParcelThermoProperties(*pPtr, 0.0);

                        pPtr->d()=d_g;

                        pPtr->nParticle()=temp_mass/mass_g;

                        // Check/set new parcel injection properties
                        this->owner().checkParcelProperties(*pPtr, 0.0*mesh.time().deltaTValue(), false);

                        // Random sampling of velocities and angles from statistical distributions
                        scalar e_vel=sizeDistributionTriple_->exponentialSample(av_vel,0.0);
                        scalar v_ang=sizeDistributionTriple_->exponentialSample(50.0/180.0*constant::mathematical::pi,0.0);
                        scalar h_ang=sizeDistributionTriple_->normalSample(i_ang2,15.0/180.0*constant::mathematical::pi);
```

Figure 27: `LocalInteractionStickReboundSplash.C`, lines 372 to 410.

```
                        h_ang   = min(constant::mathematical::pi, max(-constant::mathematical::pi, h_ang));
                        v_ang   = min(constant::mathematical::piByTwo, max(-constant::mathematical::piByTwo, v_ang+slope));

                        vector Un_splashing = (e_vel*sin(v_ang))*n;
                        vector Ut1_splashing = (e_vel*cos(v_ang)*cos(h_ang))*t1;
                        vector Ut2_splashing = (e_vel*cos(v_ang)*sin(h_ang))*t2;
                        vector Ut_splashing = Un_splashing+Ut1_splashing+Ut2_splashing;

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

                        // Assigning the splashing linear velocity
                        pPtr->U()=Ut_splashing;

                        // Apply corrections to position for 2-D cases
                        meshTools::constrainToMeshCentre(mesh, pPtr->position());

                        // Apply correction to velocity for 2-D cases
                        meshTools::constrainDirection
                        (
                            mesh,
                            mesh.solutionD(),
                            pPtr->U()
                        );
                        this->owner().addParticle(pPtr);
                        this->owner().massDeposition().boundaryField()[pI][fI] -= temp_mass/cellArea;
                        this->owner().massCheckPatterns().boundaryField()[pI][fI] -= temp_mass/cellArea;
                        }
                    }
                }
                break;
            }
            ///////////////////////////////////////////////////////////////////////////////////

            default:
            {
                FatalErrorIn
                (
                    "bool LocalInteractionStickReboundSplash<CloudType>::correct"
                    "("
                        "typename CloudType::parcelType&, "
                        "const polyPatch&, "
                        "bool&, "
                        "const scalar, "
                        "const tetIndices&"
                    ") const"
                )   << "Unknown interaction type "
                    << patchData_[patchI].interactionTypeName()
                    << "(" << it << ") for patch "
```

Figure 28: `LocalInteractionStickReboundSplash.C`, lines 411 to 459.

```
                    << patchData_[patchI].patchName()
                    << ". Valid selections are:" << this->interactionTypeNames_
                    << endl << abort(FatalError);
            }
        }

        return true;
    }

    return false;
}


template<class CloudType>
void Foam::LocalInteractionStickReboundSplash<CloudType>::info(Ostream& os)
{
    // retrieve any stored data
    labelList npe0(patchData_.size(), 0);
    this->getModelProperty("nEscape", npe0);

    scalarList mpe0(patchData_.size(), 0.0);
    this->getModelProperty("massEscape", mpe0);

    labelList nps0(patchData_.size(), 0);
    this->getModelProperty("nStick", nps0);

    scalarList mps0(patchData_.size(), 0.0);
    this->getModelProperty("massStick", mps0);

    // accumulate current data
    labelList npe(nEscape_);
    Pstream::listCombineGather(npe, plusEqOp<label>());
    npe = npe + npe0;

    scalarList mpe(massEscape_);
    Pstream::listCombineGather(mpe, plusEqOp<scalar>());
    mpe = mpe + mpe0;

    labelList nps(nStick_);
    Pstream::listCombineGather(nps, plusEqOp<label>());
    nps = nps + nps0;

    scalarList mps(massStick_);
    Pstream::listCombineGather(mps, plusEqOp<scalar>());
    mps = mps + mps0;



    forAll(patchData_, i)
    {
        os  << "    Parcel fate (number, mass)      : patch "
            <<  patchData_[i].patchName() << nl
            << "      - escape                      = " << npe[i]
            << ", " << mpe[i] << nl
            << "      - stick                       = " << nps[i]
            << ", " << mps[i] << nl;
    }
```

Figure 29: `LocalInteractionStickReboundSplash.C`, lines 460 to 516.

```

    if (this->outputTime())
    {
        this->setModelProperty("nEscape", npe);
        nEscape_ = 0;

        this->setModelProperty("massEscape", mpe);
        massEscape_ = 0.0;

        this->setModelProperty("nStick", nps);
        nStick_ = 0;

        this->setModelProperty("massStick", mps);
        massStick_ = 0.0;
    }
}
// ************************************************************************* //
```

Figure 30: `LocalInteractionStickReboundSplash.C`, lines 517 to 533.

3.3.5 Linking libraries

The newly implemented submodel needs to be linked to other classes for compilation purposes. As a first step, the itStickReboundSplash case needs to be added to the PatchInteractionModel. For this purpose, open all the files found at the following path:

`$WM_PROJECT_USER_DIR/src/lagrangianCRYOS/intermediateCRYOS/submodels/...`

`Kinematic/PatchInteractionModel/PatchInteractionModel/`

For each document, copy the lines where the term "itStick" appears. Replace the latter by the "itStickReboundSplash" expression to insure that the new switch case is taken into account. In addition, the LocalInteractionStickReboundSplash submodel should be specified in `.../intermediateCRYOS/Make/files`. For this purpose, all the lines with the term "LocalInteraction" should be copied and the term replaced by "LocalInteractionStickReboundSplash" within them.

3.4 Adding a momentum source

3.4.1 Definition

A pressure source term $\mathscr{P}$ was added to the right hand side (RHS) of the flow momentum equations in the DPMFoam solver to drive the motion of the continuous phase. It is a large-scale pressure gradient in the streamwise direction x described as:

$$\mathscr{P} = -\frac{1}{\rho_f}\frac{\partial \widetilde{p_\infty}}{\partial x} = \frac{u_*^2}{L_z} \tag{8}$$

with p the pressure, u_* the surface friction velocity and L_z the vertical extent of the domain (Sharma et al., 2018). This term was introduced in the `createFields.H` and `UcEqn.H` files both located at:

```
$WM_PROJECT_USER_DIR/applications/solvers/snowDPMFoam/
```

The pressure gradient value is computed within `createFields.H` using the user-defined friction velocity u_*, flow direction and the height of the domain L_z as an input. These variables are specified in the `run/case/constant/transportProperties` file. Once the term is computed within `createFields.H`, it is integrated in the momentum equation within `UcEqn.H`.

3.4.2 OpenFOAM scripts

The two scripts accounting for the momentum source are presented in this subsection. They also relate to the next section which describes the initial velocity profile settings (sect. 3.5). Figures 31 to 35 show the `createFields.H` file. On the other hand, Figure 36 shows `UcEqn.H`. Lines *170* to *250* of `createFields.H` contain the part that was implemented for the snow transport model. The pressure gradient is computed through several variables (lines *176 - 190*) and stored in the volVectorField gradP (lines *185* to *210*). The latter is then inserted on the RHS of the equation in `UcEqn.H` (line *7*).

```
Info<< "\nReading transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE,
        false
    )
);

word contiuousPhaseName(transportProperties.lookup("contiuousPhaseName"));

dimensionedScalar rhocValue
(
    IOobject::groupName("rho", contiuousPhaseName),
    dimDensity,
    transportProperties.lookup
    (
        IOobject::groupName("rho", contiuousPhaseName)
    )
);

volScalarField rhoc
(
    IOobject
    (
        rhocValue.name(),
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    rhocValue
);

Info<< "Reading field U\n" << endl;
volVectorField Uc
(
    IOobject
    (
        IOobject::groupName("U", contiuousPhaseName),
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field p\n" << endl;
volScalarField p
```

Figure 31: `createFields.H`, lines 1 to 57.

```
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );


    Info<< "Reading/calculating continuous-phase face flux field phic\n"
        << endl;

    surfaceScalarField phic
    (
        IOobject
        (
            IOobject::groupName("phi", contiuousPhaseName),
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        linearInterpolate(Uc) & mesh.Sf()
    );

    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PIMPLE"), pRefCell, pRefValue);

    Info<< "Creating turbulence model\n" << endl;

    singlePhaseTransportModel continuousPhaseTransport(Uc, phic);

    volScalarField muc
    (
        IOobject
        (
            IOobject::groupName("mu", contiuousPhaseName),
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        rhoc*continuousPhaseTransport.nu()
    );

    Info << "Creating field alphac\n" << endl;
    // alphac must be constructed before the cloud
    // so that the drag-models can find it
    volScalarField alphac
    (
        IOobject
        (
```

Figure 32: `createFields.H`, lines 58 to 114.

```
            IOobject::groupName("alpha", contiuousPhaseName),
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar("0", dimless, 0)
    );

    word kinematicCloudName("kinematicCloud");
    args.optionReadIfPresent("cloudName", kinematicCloudName);

    Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
    basicKinematicTypeCloud kinematicCloud
    (
        kinematicCloudName,
        rhoc,
        Uc,
        muc,
        g
    );

    // Particle fraction upper limit
    scalar alphacMin
    (
        1.0
      - readScalar
        (
            kinematicCloud.particleProperties().subDict("constantProperties")
           .lookup("alphaMax")
        )
    );

    // Update alphac from the particle locations
    alphac = max(1.0 - kinematicCloud.theta(), alphacMin);
    alphac.correctBoundaryConditions();

    surfaceScalarField alphacf("alphacf", fvc::interpolate(alphac));
    surfaceScalarField alphaPhic("alphaPhic", alphacf*phic);


    autoPtr<PhaseIncompressibleTurbulenceModel<singlePhaseTransportModel> >
    continuousPhaseTurbulence
    (
        PhaseIncompressibleTurbulenceModel<singlePhaseTransportModel>::New
        (
            alphac,
            Uc,
            alphaPhic,
            phic,
            continuousPhaseTransport
        )
    );

scalar vKC_ = readScalar(transportProperties.lookup("vKC"));
Info << "Reading k_, the Von Kármán constant  " << vKC_ << "\n" << endl;
```

Figure 33: `createFields.H`, lines 115 to 171.

```

scalar Z0_ = readScalar(transportProperties.lookup("Z0"));
Info << "Reading Z0_, the surface roughness, in m " << Z0_ << "\n" << endl;

scalar Ustar_ = readScalar(transportProperties.lookup("Ustar"));
Info << "Reading Ustar_, the friction velocity, in m s-1" << Ustar_ << "\n" << endl;

scalar H_ = readScalar(transportProperties.lookup("H"));
Info << "Reading H_, the height for the fluid domain  " << H_ << "\n" << endl;

vector flowDirection_(transportProperties.lookup("flowDirection"));
Info << "Reading flowDirection_, the flow direction " << flowDirection_ << "\n" << endl;

bool constantPGrad_(transportProperties.lookupOrDefault<bool>("constantPGrad", false));
Info << "Reading the flag if constant pressure gradients is applied to momentum , " << constantPGrad_ << "\n" << endl;

vector dP_dx = (constantPGrad_) ? (Foam::pow(Ustar_,2.0)/H_)*flowDirection_ : vector::zero;
Info << "Calculating the pressure gradient in the flow direction " << dP_dx << "\n" << endl;

scalar noiseFactor_ = readScalar(transportProperties.lookup("noiseFactor"));
Info << "Reading noiseFactor_, the noise factor " << noiseFactor_ << "\n" << endl;


const pointField& ctrs = mesh.cellCentres();

volVectorField gradP
(
    IOobject
    (
        "gradP",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedVector("gradP", dimForce/dimVolume/dimDensity,  dP_dx),
    zeroGradientFvPatchVectorField::typeName
);


if ( runTime.timeName() == "0")
{
    Random ranGen_(label(0));

    label totalCellNumber=ctrs.size();
    reduce(totalCellNumber, sumOp<label>());
    scalarField randomNumbersAllMesh(totalCellNumber, 0.0);
    forAll(randomNumbersAllMesh, i)
    {
        randomNumbersAllMesh[i]=ranGen_.scalar01();
    }
```

Figure 34: `createFields.H`, lines 172 to 223.

```
    Info << "\nthe total cell number: " << totalCellNumber << endl << endl;

    labelList LcellN(Pstream::nProcs());
    LcellN[Pstream::myProcNo()] = ctrs.size();
    Pstream::gatherList(LcellN);
    Pstream::scatterList(LcellN);

    label startLable=0;
    for(label proc=1; proc<=Pstream::myProcNo(); proc++)
    {
        startLable+=LcellN[proc-1];
    }

    Info<< "The streamwise velocity is initialized based on log law at Time = " << runTime.timeName() << nl << endl;
    forAll(ctrs, cellI)
    {
        scalar randNumber=randomNumbersAllMesh[cellI+startLable];
        scalar noise_ = (2.0*randNumber)-1.0;
        scalar varianceFact_ = 3.0*noiseFactor_*pow(Ustar_,2);
        scalar cellHeight = ctrs[cellI].z();
        Uc[cellI] = (((Ustar_/vKC_)*Foam::log(cellHeight/Z0_))+varianceFact_*noise_*((H_-0.9*cellHeight)/H_))*flowDirection_;
        Info << "Uc[cellI]: " << Uc[cellI] << endl;

    }
    Uc.correctBoundaryConditions();
    phic=linearInterpolate(Uc) & mesh.Sf();
}
```

Figure 35: `createFields.H`, lines 224 to 250.

```
fvVectorMatrix UcEqn
(
    fvm::ddt(alphac, Uc) + fvm::div(alphaPhic, Uc)
  - fvm::Sp(fvc::ddt(alphac) + fvc::div(alphaPhic), Uc)
  + continuousPhaseTurbulence->divDevRhoReff(Uc)
 ==
     gradP
  + (1.0/rhoc)*cloudSU

);

UcEqn.relax();

volScalarField rAUc(1.0/UcEqn.A());
surfaceScalarField rAUcf("Dp", fvc::interpolate(rAUc));

surfaceScalarField phicForces
(
    (fvc::interpolate(rAUc*cloudVolSUSu/rhoc) & mesh.Sf())
 +
    rAUcf*(g & mesh.Sf())
);

if (pimple.momentumPredictor())
{
    solve
    (
        UcEqn
     ==
        fvc::reconstruct
        (
            phicForces/rAUcf - fvc::snGrad(p)*mesh.magSf()
        )
    );
}
```

Figure 36: `UcEqn.H`, lines 1 to 35.

3.5 Initial velocity profile

In order to reach faster the flow equilibrium, an initial velocity profile is imposed at the beginning of the simulation. It is expected that turbulent eddies lead to an irregular logarithmic law velocity profile. This is taken into account through the varianceFact_ and noise_ scalars that add some variability and noise to the theoretical velocity curve (line *244* of `createFields.H`).

3.6 Implementation of volScalarField objects

In order to visualize the erosion and deposition of particles occurring at the snow bed as well as the friction velocity, several volScalarField objects were inserted directly into the KinematicCloud template files (path specified in section 3.2.4). This allows to have objects updated by both the aerodynamic lift and rebound-splash submodels. Note that this particular step requires extra care from the user as changes are brought to the core classes of the lagrangian library. To add these variables, search the already implemented variable called "Ucoeff" and copy every instance of it in the `KinematicCloud.C`, `KinematicCloud.H` and `KinematicCloudI.H` files. Within the copied text, replace the DimensionedField type by volScalarField. Also bring changes to the units by adding "dimMass/dimArea" for the mass per area objects or "dimVelocity" for the surface friction velocity in the definition of the object. Table 2 summarizes the implemented objects that belong to the volScalarField type.

volScalarField name	**Utility**
massDeposition	Allows the control of the amount of particles that gets generated within each cell. The available snow mass per surface area is constantly updated when particles get deposited or eroded. A negative value for this object prevents particles from being created in the numerical domain. Occurence: Figures 13-14 (LogLawShearStress) and 26-27-28 (LocalInteractionStickReboundSplash).
massCheckPatterns	Records cumulatively the mass per unit area that gets eroded and deposited in each cell for the whole simulation. At each time step, this object is updated by both submodels and allows the visualization of the snow distribution patterns resulting from the model. Occurence: Figures 14 (LogLawShearStress, line *221*) and 26-28 (LocalInteractionStickReboundSplash, lines *334* and *437*).
massDepRate	Reports the mass deposition/erosion rates per cell at each timestep based on the newly computed massCheckPatterns values and the ones from the previous time step. Occurence: Figures 14 (LogLawShearStress, line *231*).
surfaceUfric	Stores the surface friction velocity computed within each surface cell and which is used within the aerodynamic lift submodel. The friction velocity can be computed in two ways. Occurence: Figure 14 (LogLawShearStress, line *224*).

Table 2: List of the volScalarField objects implemented in OpenFOAM.

End note

This tutorial shows the main implementation scripts of the new OpenFOAM lagrangian submodels created to simulate the aeolian transport of snow. We refer to this first version of the model as *snowBedFoam 1.0.* The parts of the scripts that were not displayed in the figures (e.g. the KinematicCloud files) can be found within the official repository of the code (WSL-SLF GitLab).

References

Anderson, R. S. and Haff, P. K. (1991). Wind modification and bed response during saltation of sand in air. pages 21–51. Springer Vienna.

Bagnold, R. (1941). *The Physics of Blown Sand and Desert Dune*, pages 77–84.

Brito Melo, D. (2019). Phd candidacy report: Snow transport in extreme environments - from small to large scale modelling.

Clifton, A. and Lehning, M. (2008). Improvement and validation of a snow saltation model using wind tunnel measurements. *Earth Surface Processes and Landforms*, 33(14):2156–2173.

Clifton, A., Rüedi, J.-D., and Lehning, M. (2006). Snow saltation threshold measurements in a drifting-snow wind tunnel. *Journal of Glaciology*, 52:585–596.

Comola, F. and Lehning, M. (2017). Energy and momentum conserving model of splash entrainment in sand and snow saltation: Splash entrainment of sand and snow. *Geophysical Research Letters*.

CRYOS (2021). Laboratory of cryospheric sciences (epfl). https://www.epfl.ch/labs/cryos/.

Doorschot, J. and Lehning, M. (2002). Equilibrium saltation: Mass fluxes, aerodynamic entrainment, and dependence on grain properties. *Boundary-Layer Meteorology*, 104:111–130.

Fernandes, C., Semyonov, D., Ferrás, L., and Nobrega, J. (2018). Validation of the cfd-dpm solver dpmfoam in openfoam® through analytical, numerical and experimental comparisons. *Granular Matter*, 20.

Groot Zwaaftink, C., Mott, R., and Lehning, M. (2013). Seasonal simulation of drifting snow sublimation in alpine terrain. *Water Resources Research*, 49:1581–1590.

Kok, J. and Rennó, N. (2009). A comprehensive numerical model of steady state saltation (comsalt). *Journal of Geophysical Research (Atmospheres)*, 114:17204–.

Sharma, V., Comola, F., and Lehning, M. (2018). On the suitability of the thorpe–mason model for calculating sublimation of saltating snow. *The Cryosphere*, 12(11):3499–3509.

Vionnet, V., Martin, E., Masson, V., Guyomarc'h, G., Naaim-Bouvet, F., Prokop, A., Durand, Y., and Lac, C. (2013). Simulation of wind-induced snow transport in alpine terrain using a fully coupled snowpack/atmosphere model. *The Cryosphere Discussions*, 7:2191–2245.