Multiphase Drift Flux Flocculation Model

Mohamed Elerian

September 2023

1 Introduction

The Multiphase Drift Flux Model represents a significant advancement in the field of multiphase flow modeling. This innovative model employs a mixture approach, featuring a single momentum equation for the entire mixture and a transport equation to describe concentration fields. In this context, "Multiphase" pertains to the representation of water as one phase and each distinct fraction of sediment as an individual phase. Essentially, each group of sediment particles sharing the same diameter is considered a separate fraction.

Elerian's work builds upon the foundation laid by Goeree (2018), whose PhD thesis introduced the Multiphase Drift Flux Model, complete with a structured closure for relative velocity. Notably, this model has been successfully implemented in the open-source code OpenFOAM, demonstrating its versatility and adaptability.

One key milestone in validating the model's performance is found in the work of Elerian et al., 2022, where it was rigorously tested and evaluated in the context of turbidity currents. The results of this validation process indicate that the Multiphase Drift Flux Model adeptly captures the dynamics of turbidity currents in an acceptable manner, highlighting its efficacy and accuracy.

Furthermore, Elerian's subsequent research, as detailed in Elerian et al., 2023, takes the model to new heights by incorporating the physics of flocculation. This enhancement involves integrating the population balance equation into the transport equations governing sediment fractions. To calibrate the flocculation terms, which include break-up and aggregation phenomena, experimental data from Gillard et al., 2019 were utilized. This calibrated model was then applied to real-scale simulations, specifically for deep-sea mining projects, showcasing its practical utility in addressing complex real-world scenarios.

2 OpenFOAM

The code was originally developed for OpenFOAM-v1712. Therefore, during installation, it is crucial to use OpenFOAM-v1712. It's worth noting that this version of OpenFoam may encounter compatibility issues with Ubuntu versions beyond 18.04.6 LTS. If you intend to use a different Ubuntu version, you may need to make adjustments to the code. Some container option might be a good solution to use the OpenFOAM-v1712 on a newer version of ubuntu

The solver comprises two primary classes: the first being the **multiphaseDriftMixture**, and the second being the **multiphaseRelativeVelocityModel**. Within the **multiphaseDriftMixture** class, the **phaseDrift** class is utilized. Next to the phaseDrift class the multiphaseDriftMixture class is defined in **multiphaseDriftMixture.C** and **multiphaseDriftMixture.H** files.

2.1 phaseDrift Class

Additionally, there exists a phaseDrift directory, housing **phaseDrift.H** and **phaseDrift.C** files. The **phaseDrift.H** file defines the structure and interface of the `phaseDrift` class, encompassing member variables and functions. Conversely, the **phaseDrift.C** file contains the actual code implementations for these functions. The class contains private data members, including references to `phi_` (`surfaceScalarField`), `U_` (`volVectorField`), and various properties like `name_`, `dict_`, `rho_`, `d_`, and `nuModel_`. These data members hold information related to phase properties and parameters.

A snipped code is shown at figures 1 and 2 for phaseDrift.H and phaseDrift.C repsectively.

```
*---------------------------------------------------------------------------*\
                          Class phaseDrift Declaration
*---------------------------------------------------------------------------*/

/The phaseDrift class is defined, which is derived from volScalarField. This indicates that phaseDrift is a type of scalar field used for modeling phase properties
lass phaseDrift

   public volScalarField


   // Private data


        //- reference to mixture phi
       const surfaceScalarField& phi_;

       //- reference to mixture U
       const volVectorField& U_;

       //- Name of the phaseDrift
       word name_;

       //- Phase dictionary
       dictionary dict_;

       //- Phase density
       dimensionedScalar rho_;

       //- contiuos Phase density
       //dimensionedScalar rhocc_;



       //- Phase diameter for discrete phases
       dimensionedScalar d_;

       //- Viscosity model for continous phase
       autoPtr<viscosityModel> nuModel_;
```

Figure 1: PhaseDrift class declaration.

```
//                              Constructors                             //
// Constructor for the phaseDrift class
// Initializes a phaseDrift object with provided parameters and configuration settings.

Foam::phaseDrift::phaseDrift
(
    const surfaceScalarField& phi,  // Reference to a surfaceScalarField object for phase information (rho*U)
    const volVectorField& U,        // Reference to a volVectorField object for velocity information
    const word& phaseDriftName,     // Name of the phaseDrift
    const dictionary& dict         // Dictionary containing the simulation configuration settings (transportproprteis)
)
:
    volScalarField
    (
        // Initialize the volScalarField base class with I/O settings
        IOobject
        (
            IOobject::groupName("alpha", phaseDriftName),   // Field name and group name
            U.mesh().time().timeName(),                     // Current time for I/O
            U.mesh(),                                       // Mesh associated with the field
            IOobject::MUST_READ,                            // Must read from input
            IOobject::AUTO_WRITE                           // Automatically write to output
        ),
        U.mesh()                                           // Mesh associated with the field
    ),
    phi_(phi),               // Initialize the phi_ member variable with the provided surfaceScalarField
    U_(U),                   // Initialize the U_ member variable with the provided volVectorField
    name_(phaseDriftName),   // Initialize the name_ member variable with the provided phaseDriftName
    dict_(dict),             // Initialize the dict_ member variable with the provided dictionary
    rho_("rho", dimDensity, dict_),   // Initialize the rho_ member variable with the keyword "rho," dimension of density, and dictionary
    d_("d", dimLength, -1)          // Initialize the d_ member variable with the keyword "d," dimension of length, and default value -1
{
    // Check if the "transportModel" keyword is found in the dictionary
    if (dict_.found("transportModel"))
    {
        // If found, create a viscosity model and associate it with "nu" field
        nuModel_.reset
        (
            viscosityModel::New
            (
                IOobject::groupName("nu", phaseDriftName),  // Group name for nu field
                dict_,                                      // Dictionary with configuration settings
                U_,                                         // Velocity field U
                phi_                                        // Phase field phi
            ).ptr()
        );
    }

    // Set the value of the d_ member variable to the value found in the dictionary with the key "d"
    // If not found, set it to the default value of -1 with the dimension of length (dimLength).
    d_ = d_.lookupOrDefault("d", dict_, dimLength);
}
```

Figure 2: PhaseDrift class constructor intilzation .

2.2 MultiphaseDriftMixture Class

This class is responsible for the main calculations of the sediment fraction calculations.

Solving this equation for each fraction using a function in the code uses solveAlphas() and it is programmed as follow:

```
void Foam::multiphaseDriftMixture::solveAlphas()
{
    static label nSolves = -1;
    nSolves++;

    const dictionary& alphaControls = mesh_.solverDict("
        alpha");
    label nAlphaCorr(readLabel(alphaControls.lookup("
        nAlphaCorr")));
    bool MULESCorr(readBool(alphaControls.lookup("
        MULESCorr")));
    bool limitAlphaPhi(readBool(alphaControls.lookup("
        limitAlphaPhi")));

    PtrList<surfaceScalarField> alphaPhiCorrs(phases_.
```

```
        size());

    calculateAlphaPhis(alphaPhiCorrs, phases_);
}
```

The parameter nAlphaCorr is for ensuring calculation stability. MULESCorr and limitAlphaPhi are intended to serve as flux limiters, although they may not perform as expected. Nevertheless, I recommend enabling them because I have observed that they contribute to the stability of the calculations. You can activate these options in the 'system/fvsolution' settings. For a more comprehensive understanding of the MULES method, I suggest referring to Santiago Damian's extensive research in his PhD thesis Damián and Nigro, 2014, where he has conducted thorough work on the MULES method.

2.2.1 The Advection Term

The original fraction equation of the drift flux model (without flocculation) is written as follows:

$$\frac{\partial \alpha_k}{\partial t} + \nabla.(\alpha_k \mathbf{u_k}) = \nabla.\Gamma_t \nabla \alpha_k, \tag{1}$$

$$\frac{\partial \alpha_k}{\partial t} + \nabla.(\alpha_k (\mathbf{u_m} + \mathbf{u_{km}})) = \nabla.\Gamma_t \nabla \alpha_k. \tag{2}$$

The 'calculateAlphaPhis' function plays a crucial role as it computes the second term on the left-hand side of the equation ($\nabla.(\alpha_k \mathbf{u_k})$). This term represents the fluxes of the fractions, and it involves calculations with the velocities U_m and U_{km}, as depicted in the following code. The code is thoroughly documented line by line within the original source code.

```
void Foam::multiphaseDriftMixture::calculateAlphaPhis
(
    PtrList<surfaceScalarField>& alphaPhiCorrs,
    const UPtrList<phaseDrift>& phases
)
{
    word alphaScheme("div(phi,alpha)");
    word alpharScheme("div(phirb,alpha)");

    int phasei = 0;
    forAllConstIter(UPtrList<phaseDrift>, phases, iter)
          //iteration for all phases: dispersed,
        continuous
    {
        const phaseDrift& alpha = iter();

        alphaPhiCorrs.set
```

```
        (
            phasei,
            new surfaceScalarField
            (
                "phi" + alpha.name() + "Corr",
                fvc::flux
                (
                    phi_,
                    alpha,
                    alphaScheme
                )
            )
        );

        surfaceScalarField& alphaPhiCorr = alphaPhiCorrs[
            phasei];     // alpha_k* Um

        surfaceScalarField phikm(fvc::flux(UkmPtr_[phasei
            ]));          // alpha_k U_km

        alphaPhiCorr += fvc::flux //alphPhi = alphaPhi (
            Um) + alphaPhi(Ukm)
        (
            phikm,
            alpha,
            alpharScheme
        );

        // Ensure that the flux at inflow BCs is
            preserved
        fixedFluxOnPatches(alphaPhiCorr, alpha);

        phasei++;
    }
}
```

2.2.2 Flocculation

Aggregation and breakup are the two main processes that govern phase transition among solid particles. Despite flocculation being a complex process, Hounslow et al. (1988) presents a simple discretized equation that captures the transition of particles between phases. This discretization approach is based on size classes or size groups, in which the entire size range of sediment particles is divided into a specific number of groups. Each individual class is identified by its size, meaning that class k contains one size-based fraction. For simplicity, each class is treated as a phase, with the subscript k representing the class. The discretization approach is based on the idea that the volume of a particle in phase $k+1$ is double that of a particle in phase k.

$$\upsilon_{k+1} = 2\upsilon_k, \tag{3}$$

where υ_k is the particle size in class k. Bearing this in mind, the discretized form of PBE can be described as follows:

$$\begin{aligned}\frac{dN_k}{dt} = \sum_{j=1}^{k-2} 2^{j-k+1}\gamma\beta_{k-1,j}N_{k-1}N_j + \frac{1}{2}\gamma\beta_{k-1,k-1}N_{k-1}^2 \\ -N_k\sum_{j=1}^{k-1} 2^{j-k}\gamma\beta_{k,j}N_j - N_k\sum_{j=k}^{kmax}\gamma\beta_{k,j}N_j \\ -S_kN_k + \sum_{j=i}^{imax}\zeta_{k,j}S_jN_j,\end{aligned} \tag{4}$$

where $N_k(\#/m^3) = \alpha_k/\upsilon_k$ is the number concentration in particles of phase k, γ is the collision efficiency, $\beta_{k,j}(m^3/s)$ is the collision frequency between particles in groups k and j, $S_k(s^{-1})$ is the breakage rate of particles in group k and ζ is the breakage distribution function which determines the volume fraction of particles of group k resulting from the fragmentation of particles of group j. Figure 3 provides a graphical representation of the RHS source terms of Equation 4, in which four functions that represent the particulate system are present; these functions are:

1. Collision efficiency γ,
2. Collision frequency $\beta_{k,j}$,
3. Breakage rate S_k,
4. Breakage distribution function ζ.

Such functions require closures, as they are the main driver of the phase transition process. Each closure will be discussed in the following subsections.

2.2.3 Collision Efficiency (γ)

The aggregation probability between two particles from different phases k and j can be described by the collision efficiency γ, which ranges from 0 to 1. When every collision results in floc formation, $\gamma = 1$. On the other hand, when no collision leads to floc formation, $\gamma = 0$. Determining the collision efficiency is a complex process as it depends on the surface properties of the particles, interaction forces between particles, and hydrodynamic effects within the aggregate. In some cases, γ is considered as a constant value (Biggs & Lant, 2002; Golzarijalal et al., 2018), while in others, it is considered as an adjustable parameter (Wickramasinghe et al., 2005; Zhang & Li, 2003).

The collision efficiency is declared as follows:

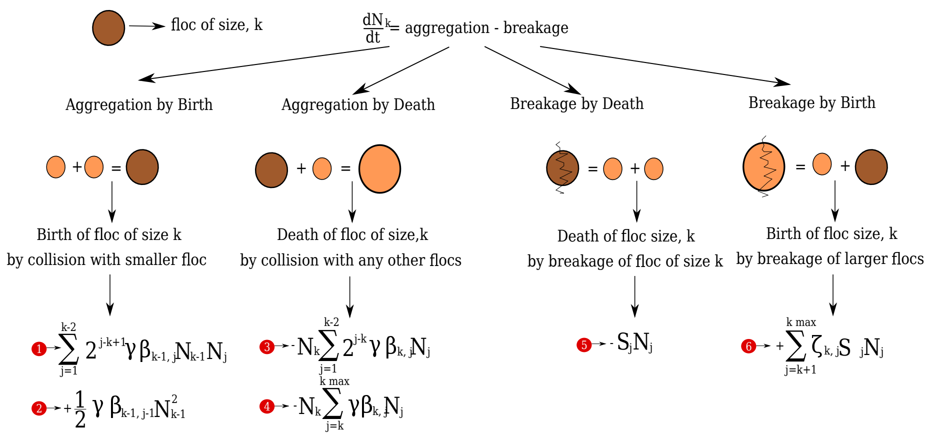

Figure 3: Aggregation an breakage dynamics of the discrtized PBE, Equation 4, adapted from (Biggs & Lant, 2002). Following Equation 4, first term is formation of floc k due to collision of unequal particle sizes, second term is formation of floc k due to collision of equal particle sizes, third term is death of floc k due to collision with smaller particles, fourth term is death of floc k due to collision with equal or large particles, fifth term is death of floc k due to breakage, sixth term is formation of floc k due to breakage of large particles.

```
// Collision Efficiency
scalar Alpha_;
```

By default, its value is set to 1, but it can be adjusted in the `transportProperties` file using the `Alpha` keyword. This parameter is declared in the constructor as follows:

```
Alpha_(lookupOrDefault<scalar>("Alpha", 1.0)),
```

This provides the flexibility to configure the collision efficiency to match specific requirements.

2.2.4 Collision Frequency ($\beta_{k,j}$)

Assessing the frequency of collisions is challenging due to the intricate interaction of various factors, including:

1. Perikinetic aggregation of flocs, i.e. Brownian motion, $\beta^{Br}_{k,j}$,
2. Orthokinetic aggregation of flocs, i.e. aggregation occurs because of a velocity gradient in a fluid, $\beta^{sh}_{k,j}$,
3. Gravity aggregation,i.e. differential settling, $\beta^{g}_{k,j}$.

For any two particles, each belonging to different phases k or j, the collision frequency is calculated as the sum of three components:

$$\beta_{k,j} = \beta_{k,j}^{Br} + \beta_{k,j}^{sh} + \beta_{k,j}^{g}. \tag{5}$$

The variable β is declared as a scalar pointer list in the code as follows:

```
// Collision Frequency
PtrList<volScalarField>  Beta_;
```

In the constructor, it is initialized to have a size of ‘phasesk$_{.}size() * phasesk_{.}size()$‘ :

```
Beta_(phasesk_.size() * phasesk_.size());
```

Within a loop iterating over fractions, the individual components are calculated and summed up:

```
Beta_[I] = Beta_shear + Beta_settling + Beta_Brownian;
```

The components, including Brownian motion, shear-induced aggregation, and gravitational settling, contribute to the overall collision frequency between particles in different phases.

The flocs resulting from an aggregation process are irregular, permeable structures. Two parameters are introduced by (Veerapaneni & Wiesner, 1996) to correct for the fluid collision efficiency,

1. The efficiency of fluid accumulation inside a floc η,
2. The ratio of the drag force of a permeable aggregate to the drag force of an impermeable aggregate Ω.

2.2.5 *β_Brownian*

The collision frequency between permeable and irregular flocs is determined by the following equations:

$$\beta_{k,j}^{Br} = \frac{2k_bT}{3\mu}\left(\frac{1}{\Omega_k r_k} + \frac{1}{\Omega_j r_j}\right)(r_k + r_j) \tag{6}$$

Equation 6 represents the collision frequency due to Brownian motion, where k_b is the Boltzmann constant, T is the temperature, and μ is the dynamic viscosity. The terms Ω_k and Ω_j correspond to certain coefficients associated with the phases.

Within a loop that iterates along the fractions, the following calculations take place in the C++ code:

```
volScalarField Beta_Brownian = (2.0/3.0) * (Kb * T_ / (
    phasec_.first().mu() / muunit)) * ((1 / (omegacoef_[i]
     * colldiameters_[i])) + (1 / (omegacoef_[j] *
    colldiameters_[j]))) * (colldiameters_[i] +
    colldiameters_[j]);
```

Here, Kb represents the Boltzmann constant, and T is the temperature as specified in the code.

```
scalar Kb = 1.38064852e-23;
// Temperature
scalar T_;
T_(lookupOrDefault<scalar>("T", 293.0));
```

Considering the fractal dimension, the effective radius (r_k) of a floc is determined by the following equation:

$$r_k = r_2 \left(\frac{\upsilon_k}{\upsilon_2}\right)^{1/d_f} \tag{7}$$

Here, $r_2 = d_2/2$ is the primary particle radius, and d_f represents the fractal dimension, which ranges from 1 to 3. A value of 1 signifies a line of particles, while 3 represents a solid sphere. The irregular shape and permeability of a floc are influenced by the value of d_f.

```
\\fractal dimension
df_(lookupOrDefault<scalar>("df", 1));
```

The effective capture radius of an aggregate of phase k is stored in the list `colldiameters_`.

```
// Collision diameter list
scalarField colldiameters_;

// Collision Particle Diameter for all sediment phases
colldiameters_(phasesk_.size());
```

The collision diameters are calculated using the `calccollDiameters` function, iterating through the phases.

```
void Foam::multiphaseDriftMixture::calccollDiameters()
{
    label x = 0;
    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        colldiameters_[x] = (diameters_[0]/2) * pow((x+1)
            /kc_, (1/df_));
        x++;
    }
}
```

2.2.6 β_{Shear}

The collision frequency due to shear, denoted as $\beta^{sh}_{k,j}$, is calculated as follows:

$$\beta^{sh}_{k,j} = 1.294G\left(\sqrt{\eta_k}r_k + \sqrt{\eta_j}r_j\right)^3, \tag{8}$$

The calculation is performed using the following code:

```
volScalarField Beta_shear = 1.294 * G_ * pow((sqrt(
    etacoef_[i]) * colldiameters_[i] + sqrt(etacoef_[j]) *
    colldiameters_[j]), 3.0);
```

Here, $G = \left(\frac{\epsilon}{\nu}\right)^{1/2}$ represents the shear rate.

```
volScalarField G_;
```

The turbulent shear rate G is calculated as:

```
G_(
    IOobject("G", mesh_.time().timeName(), mesh_,
        IOobject::NO_READ, IOobject::AUTO_WRITE),
    mesh_,
    dimensionedScalar("G", dimless, scalar(0.0))
),
```

The function for calculating the shear rate is defined as:

```
void Foam::multiphaseDriftMixture::CalcG()
{
    kk_ = turbulencePtr_->k()();
    volScalarField epsField = turbulencePtr_->epsilon()()
        ;
    volScalarField nuField = phasec_.first().nu()();
    volScalarField Gunit(
        IOobject("Gunit", mesh_.time().timeName(), mesh_)
            ,
        mesh_,
        dimensionedScalar("Gunit", dimless/dimTime/
            dimTime, scalar(1.0))
    );
    G_ = pow(epsField / nuField / Gunit, 1.0/2.0); // G =
        (epsilon/nu)^{1/2}
}
```

In the equations, μ represents the dynamic viscosity, and Ω in Equation 6 is the ratio between the force exerted by the fluid on a permeable aggregate and the force exerted by the fluid on an impervious sphere of equivalent size (Jeldres et al., 2015). It is given by:

$$\Omega = \frac{2\xi^2\left(1 - \frac{\tanh\xi}{\xi}\right)}{2\xi^2 + 3\left(1 - \frac{\tanh\xi}{\xi}\right)}, \tag{9}$$

Here, $\xi = \frac{r}{\sqrt{K}}$ is the dimensionless permeability, where K represents the permeability of an aggregate. The Brinkman and Happel permeability equation is used to calculate K (Li & Logan, 2001).

2.3 Ω Calculation

The variable Ω is declared as follows:

```
scalarField omegacoef_;
```

In the constructor, it is initialized for each phase:

```
omegacoef_(phasesk_.size()),
```

The calculation of Ω is performed using the following function:

```
void Foam::multiphaseDriftMixture::Calcomegacoef()
{
    label l = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        omegacoef_[l] = (2 * pow(zetacoef_[l], 2) * (1 -
            (tanh(zetacoef_[l]) / zetacoef_[l]))) / (2 *
            pow(zetacoef_[l], 2) + 3 * (1 - (tanh(
            zetacoef_[l]) / zetacoef_[l])));
        l++;
    }
}
```

The variable ξ is declared as follows:

```
scalarField zetacoef_;
```

In the constructor, it is initialized for each phase:

```
zetacoef_(phasesk_.size()),
```

The calculation of ξ is performed using the following function:

```
void Foam::multiphaseDriftMixture::Calczetacoef()
{
    label yy = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        zetacoef_[yy] = aggradius_[yy] / pow(
            aggpermeability_[yy], 0.5);
        yy++;
    }
}
```

The variable r is calculated and declared as follows:

```
// Radius of the aggregates taking into account the
   fractal shape
scalarField aggradius_;
```

In the constructor, it is initialized for each phase:

```
// aggregate radius using the fractal dimension
aggradius_(phasesk_.size()),
```

The calculation of r is performed using the following function:

```
// Calculate the radius of aggregates: r = r_0 (V/V_0)
    ^(1/2)
void Foam::multiphaseDriftMixture::Calcaggradius()
{
    aggradius_[0] = diameters_[0] / 2;

    for (label xx = 1; xx <= volumes_.size() - 1; xx++)
    {
        aggradius_[xx] = aggradius_[0] * pow((volumes_[xx
            ] / volumes_[0]), 1 / df_);
    }
}
```

The permeability K is calculated according to the equation:

$$K = \frac{d_2^2}{72}\left(3 + \frac{3}{1-\phi} - \sqrt[3]{\frac{8}{1-\phi} - 3}\right) \tag{10}$$

The variable K is declared as follows:

```
// permeability of the aggregates
scalarField aggpermeability_;
```

In the constructor, it is initialized for each phase:

```
// aggregate permeability
aggpermeability_(phasesk_.size()),
```

The calculation of K is performed using the following function:

```
// Calculate the permeability of aggregates
void Foam::multiphaseDriftMixture::Calcaggpermeability()
{
    label y = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        aggpermeability_[y] = (pow(diameters_[y], 2) /
            72) * (3 + (3 / (1 - aggporositiy_[y])) - pow
            ((8 / (1 - aggporositiy_[y]) - 3), 1/3));
        y++;
    }
}
```

The porosity ϕ of a floc is calculated using the fractal dimension approach:

$$\phi = 1 - C_b \left(\frac{d_k}{d_2}\right)^{d_f - 3}, \tag{11}$$

where C_b is the packing coefficient, which is assumed to be 1 in this work. As demonstrated by Equation 11, there is an inverse relationship between the fractal dimension and floc porosity. As the fractal dimension increases, the floc porosity decreases. This trend continues until the fractal dimension reaches its maximum value of $d_f = 3$, at which point the floc porosity reaches its minimum value of 0, indicating a solid case with no voids or pores (Li & Logan, 2001).

The variable ϕ is declared as follows:

```
// porosity of the aggregates
scalarField aggporositiy_;
```

In the constructor, it is initialized for each phase:

```
aggporositiy_(phasesk_.size()),
```

The calculation of ϕ is performed using the following function:

```
void Foam::multiphaseDriftMixture::Calcaggporositiy()
{
    label xxx = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        aggporositiy_[xxx] = 1 - (packc_ * pow((
            diameters_[xxx] / diameters_[0]), df_-3));
        xxx++;
    }
}
```

The variable *packc* is defaulted to be 1.

```
packc_(lookupOrDefault<scalar>("C", 1)),
```

The calculation of η is done based on the Brinkman equations (Wickramasinghe et al., 2005). It's calculated using the equation:

$$\eta = 1 - \frac{d}{\xi} - \frac{c}{\xi^3}, \tag{12}$$

The variable η is declared as follows:

```
// eta coefficient: the ratio of the flow moving through
    an aggregate to the total flow approaching the
    aggregate
scalarField etacoef_;
```

In the constructor, it is initialized for each phase:

```
etacoef_(phasesk_.size()),
```

The calculation of η is performed using the following function:

```
void Foam::multiphaseDriftMixture::Calcetacoef()
{
    label nn = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        etacoef_[nn] = 1 - (dcoef_[nn] / zetacoef_[nn]) -
            (ccoef_[nn] / (pow(zetacoef_[nn], 2)));
        nn++;
    }
}
```

The coefficients d, c, and J are calculated as follows:

$$d = \frac{3}{J}\xi^3 \left(1 - \frac{tanh(\xi)}{\xi}\right), \tag{13}$$

$$c = -\frac{1}{J}\left(\xi^5 + 6\xi^3 - \frac{tanh(\xi)}{\xi}\left(3\xi^5 + 6\xi^3\right)\right), \tag{14}$$

$$J = 2\xi^2 + 3 - 3\frac{tanh(\xi)}{\xi}. \tag{15}$$

The variables J, c, and d are declared as follows:

```
// J coefficient
scalarField Jcoef_;
// c coefficient
scalarField ccoef_;
// d coefficient
scalarField dcoef_;
```

In the constructor, these variables are initialized for each phase:

```
// J coefficient
Jcoef_(phasesk_.size()),
// c coefficient
ccoef_(phasesk_.size()),
// d coefficient
dcoef_(phasesk_.size())
```

The calculation functions for J, c, and d are as follows:

```
% Calculate J coefficient
void Foam::multiphaseDriftMixture::CalcJcoef()
{
```

```
    for (label yyy = 0; yyy <= volumes_.size() - 1; yyy
        ++)
    {
        Jcoef_[yyy] = (2 * pow(zetacoef_[yyy], 2)) + 3 -
            (3 * tanh(zetacoef_[yyy]) / zetacoef_[yyy]);
    }
}

% Calculate c coefficient
void Foam::multiphaseDriftMixture::Calcccoef()
{
    label z = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        ccoef_[z] = (-1 / Jcoef_[z]) * (pow(zetacoef_[z],
             5) + (6 * pow(zetacoef_[z], 3)) - ((tanh(
            zetacoef_[z]) / zetacoef_[z]) * ((3 * pow(
            zetacoef_[z], 5)) + (6 * pow(zetacoef_[z], 3))
            ));
        z++;
    }
}

% Calculate d coefficient
void Foam::multiphaseDriftMixture::Calcdcoef()
{
    label zz = 0;

    forAllIter(PtrDictionary<phaseDrift>, phasesk_, iter)
    {
        dcoef_[zz] = (3 / Jcoef_[zz]) * (pow(zetacoef_[zz
            ], 3) * (1 - (tanh(zetacoef_[zz]) / zetacoef_[
            zz])));
        zz++;
    }
}
```

2.4 $\beta_{\textbf{settling}}$

$$\beta^g_{k,j} = \pi \left(\sqrt{\eta_k} r_k + \sqrt{\eta_j} r_j\right)^2 |(u_{kr} - u_{jr})|, \tag{16}$$

```
        volScalarField Beta_settling = Foam::constant::
            mathematical::pi *(pow((pow(etacoef_[i],0.5)*
            aggradius_[i])+(pow(etacoef_[j],0.5)*
```

```
aggradius_[j]) ,2.0))*(mag(mag(UkmPtr_[i]-
UkmPtr_[j])/Vunit));
```

2.4.1 Breakage Rate (S_k)

In general, a floc breaks up when the imposed external force on the floc exceeds the floc's strength. The breakage rate S_k is determined as follows (Winterwerp, 1998):

$$S_i = E_b G \left(\frac{d_k - d_2}{d_2}\right)^{3-d_f} \left(\frac{\mu G}{F_y/d_k^2}\right)^{\frac{1}{2}}, \tag{17}$$

where F_y is floc strength. Very little is known about F_y, but Van Leussen, 1994 estimated it to be approximately $10^{-10}N$. E_b is the breakage coefficient.

The following constants are declared for the break-up process:

```
//- A constant parameter for break-up process
scalar Eb_;

//- Floc Strength
scalar Fy_;

// Declare and initialize Eb_ and Fy_
Eb_(lookupOrDefault<scalar>("Eb", 1e-5)),
Fy_(lookupOrDefault<scalar>("Fy", 1e-10)),
```

The implementation is as follows:

```
// Compute Breakage Rate
BrkRate();

// Construct Breakage rate
void Foam::multiphaseDriftMixture::BrkRate()
{
    volScalarField muunit
    (
        IOobject
        (
            "muunit",
            mesh_.time().timeName(),
            mesh_
        ),
        mesh_,
        dimensionedScalar("muunit", dimMass/dimLength/
            dimTime, scalar(1.0))
    );
```

```
    forAll(volumes_, i)
    {
        S_[i] = Eb_ * G_ * pow(((diameters_[i] -
            diameters_[0]) / diameters_[0]), 3.0 - df_)
                * pow((phasec_.first().mu() / muunit * G_
                    ) / (Fy_ / pow(diameters_[i], 2.0)),
                    0.5);
    }
}
```

2.4.2 Breakage distribution function

Determining the size distribution of daughter flocs produced from the breakup of a parent floc is challenging. Theoretical breakup distribution functions are used to find the best fit for the experimental data. In this chapter, we adopt a binary breakage function, which we believe will be adequate, as discussed in (Chen et al., 1990; Jeldres et al., 2015).

$$\zeta_{k,j} = \frac{\upsilon_j}{\upsilon_k},\ j = k+1\ and\ \zeta_{k,j} = 0\ for\ j \neq k+1. \tag{18}$$

And is declared and calculated as follow:

```
// Breakage Distribution Function
scalar Gamma_;


Gamma_(lookupOrDefault<scalar>("Gamma", 2.0)),
```

It is elaborated more in the sixth source term.

2.5 Coupling Between Drift-Flux Model and PBE

We couple fluid dynamics and phase transition by using the relationship between the number density N_k and the volume concentration of particle phase, which remains constant over time.

$$N_k = \frac{\alpha_k}{\upsilon_k}. \tag{19}$$

From Equation 19 and Equation 1 we can deduce the following equation:

$$
\begin{aligned}
\frac{\partial \alpha_k}{\partial t} + \nabla.(\alpha_k(\mathbf{u_m} + \mathbf{u_{km}})) = \Bigg[& \upsilon_k \sum_{j=1}^{k-2} 2^{j-k+1} \gamma \beta_{k-1,j} N_{k-1} N_j \\
& + \frac{1}{2} \upsilon_k \gamma \beta_{k-1,k-1} N_{k-1}^2 - \upsilon_k N_k \sum_{j=1}^{k-1} 2^{j-k} \gamma \beta_{k,j} N_j - \\
\upsilon_k N_k \sum_{j=i}^{imax} \gamma \beta_{k,j} N_j - S_k N_k \upsilon_k + \upsilon_k \sum_{j=i}^{imax} \zeta_{k,j} S_j N_j \Bigg] & + \nabla.\Gamma_t \nabla \alpha_k.
\end{aligned} \tag{20}
$$

Thus, we replace Equation 1 by Equation 20, obtaining a particle of phase k that travels through time, space (advection and diffusion), and phase domains (i.e, volume as an internal coordinate).

Inside the function of solving the equation `void Foam::multiphaseDriftMixture::solveAlphas()`

Declaration of α_t (sumAlpha) and fluxes of the fractions (`rhoPhi_`)

```
volScalarField sumAlpha
  (
      IOobject
      (
          "sumAlpha",
          mesh_.time().timeName(),
          mesh_
      ),
      mesh_,
      dimensionedScalar("sumAlpha", dimless, 0.0)
  );


  // Reset rhoPhi
  rhoPhi_ = dimensionedScalar("rhoPhi", dimMass/dimTime
      , 0.0);
```

Then calucations of the paramters that is used for the floccuation

```
  // Calculate ShearRate
  CalcG();
  // Compute collision Frequency
  ColFreq();
  // Compute collision effecenci
  Coleff();
  // Comput Breakage Rate
  BrkRate();
```

Initializing and calculating the floccuation source terms

```
// Initialize the summation of flocculation source
    terms
forAll(SumSource_ ,cellI)
{
    SumSource_[cellI] = 0;
}
for (label phasei = 0; phasei <= phasesk_.size()-1;
    phasei++)
{
    // Calculate Flocculation Source Terms for Each
        Sediment Phase
    SrcList[phasei]=calSource(phasei);
    SumSource_ += SrcList[phasei];

}
```

This function `calSource(phasei)` is described as follows.

```
Foam::multiphaseDriftMixture::calSource(label i)
{

   //Declare source terms
    volScalarField Source
    (
        IOobject
        (
            "Source",
            mesh_.time().timeName(),
            mesh_
        ),
        mesh_,
        dimensionedScalar("Source", dimless, scalar(0.0))
    );

    volScalarField Source1
    (
        IOobject
        (
            "Source1",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("Source1", dimless, scalar(0.0)
            )
```

```
);
volScalarField Source2
(
    IOobject
    (
        "Source2",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh_,
    dimensionedScalar("Source2", dimless, scalar(0.0)
        )
);
volScalarField Source3
(
    IOobject
    (
        "Source3",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh_,
    dimensionedScalar("Source3", dimless, scalar(0.0)
        )
);
volScalarField Source4
(
    IOobject
    (
        "Source4",
        mesh_.time().timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh_,
    dimensionedScalar("Source4", dimless, scalar(0.0)
        )
);
volScalarField Source5
(
    IOobject
```

```
        (
            "Source5",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("Source5", dimless, scalar(0.0)
            )
    );
    volScalarField Source6
    (
        IOobject
        (
            "Source6",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("Source6", dimless, scalar(0.0)
            )
    );
//------------------------------------------------------------
    //First Term: Unequal Size
    if (i>=2)
    {
        for (label j = 0; j <= i-2 ; j++)
        {
                Source1+=pow(2.0,(j-i+1)*1.0)*Alpha_*
                    Beta_[j +(i-1)*phasesk_.size()]*
                    phasesk_[phaseskName_[i-1]]/volumes_[i
                    -1]*phasesk_[phaseskName_[j]]/volumes_
                    [j]*volumes_[i];
        }

    }
    else {
            Source1 == 0;
         }
//------------------------------------------------------------
    //- Second Term: Equal Size Aggregation
    if (i>=1)
    {
```

```
        Source2 = (0.5)*Alpha_*Beta_[i-1 +(i-1)*phasesk_.
            size()]*phasesk_[phaseskName_[i-1]]/volumes_[i
            -1]*phasesk_[phaseskName_[i-1]]/volumes_[i-1]
            *volumes_[i];
    }
    else {
            Source2 == 0;
        }
//---------------------------------------------------------------
      //- Third Term: Aggregation with smaller particles
    if(i>=1 && i<=phasesk_.size()-2)
   {
        for (label j = 0; j <= i-1 ; j++)

        {
        Source3+= (-1.0)*pow(2.0,(j-i)*1.0) * Alpha_*
            Beta_[j+i*phasesk_.size()] * phasesk_[
            phaseskName_[j]]/volumes_[j] * phasesk_[
            phaseskName_[i]]/ volumes_[i] * volumes_[i] ;
        }
    }
    else {
            Source3 == 0;
        }
//---------------------------------------------------------------
    //- Fourth Term: Aggregation with euql or larger
        particles
  for (label j = i; j <= phasesk_.size()-2; j++)
    {
         Source4+=(-1.0)*Alpha_*Beta_[j+i*phasesk_.size()
             ]*phasesk_[phaseskName_[j]]/volumes_[j]*
             phasesk_[phaseskName_[i]]/volumes_[i] *
             volumes_[i];
    }

//---------------------------------------------------------------
    //- Fifth Term (self-breakage)
   if (i>=1)
    {
        Source5=(-1.0)*S_[i]*phasesk_[phaseskName_[i]] /
            volumes_[i] * volumes_[i];
    }
    else {
            Source5 == 0;
        }
//---------------------------------------------------------------
```

```
    //- Sixth Term (due to breakage of i+1)
     if(i<=phasesk_.size()-2)
    {
        //- Here set binary breakup by default; Gamma_ =
            2, Gamma_/v{i+1}*v{i} = 2 *1/2 = 1
        Source6=S_[i+1]*phasesk_[phaseskName_[i+1]];
    }
      else {
            Source6 == 0;
         }
//----------------------------------------------------------------
    Source = Source1+Source2+Source3+Source4+Source5+
        Source6;
```

In order to test the source terms (Source1_, ...), they are initialized within the `multiphaseDriftMixture` constructor. This allows for the evaluation and examination of these terms while solving the case. So you can keep them or leave them as you prefer.

```
Source1_ = Source1;
Source2_ = Source2;
Source3_ = Source3;
Source4_ = Source4;
Source5_ = Source5;
Source6_ = Source6;
```

2.5.1 The Limiter (Packing limit)

In reality, the maximum concentration of sediment typically falls within the range of 0.5% to 0.6%. It's important to note that a value of 1 represents the entire mixture, which includes both water and sediment. Therefore, we must ensure that the sediment's maximum fraction within any cell does not exceed this established limit, often referred to as the "defined limit." Within the code, the keywords `alphaMax` and `alphaDiffusion` are significant in this context:

```
// Maximum dispersed phase-fraction
scalar alphaMax_;
// Numerical Diffusion
scalar alphaDiffusion_;
alphaMax_(lookupOrDefault<scalar>("alphaMax", 0.45));
alphaDiffusion_(lookupOrDefault<scalar>("alphaDiffusion",
    1.0));
```

The Mules method uses 'alphaMax' in its calculations, and 'alphaDiffusion' is used separately as a diffusion term in the fraction equation.

```
// Calculation of the alphaDiffusion 1
```

```
forAll(alphas_, cellI)
{
    if (alphas_[cellI] > alphaMax_)
    {
        alphaDiffusion_1_[cellI] = alphaDiffusion_;
    }
}

// Fraction equation with diffusion term
fvScalarMatrix alpha1Eqn
(
    fvm::ddt(alpha) - fvc::ddt(alpha)
    - fvm::laplacian(turbulencePtr_->nut() +
        alphaDiffusion_1_, alpha) - Sa
);
```

You must ensure that both of them are defined in your transportProperties file.

In the context of calculating the volumetric concentration of the continuous phase, the following C++ code snippet is used:

```
volScalarField& alphac = phasec_.first();
alphac = 1.0 - sumAlpha;

phaseDrift& alpha = phasec_.first();

surfaceScalarField alphaPhi
(
    IOobject
    (
        "alphaPhi",
        mesh_.time().timeName(),
        mesh_
    ),
    mesh_,
    dimensionedScalar("0", phi_.dimensions(), 0.0)
);
calculateAlphaPhi(alphaPhi, alpha, phasei);
fixedFluxOnPatches(alphaPhi, alpha);
rhoPhi_ += alphaPhi * alpha.rho();
Info << alpha.name() << " volume fraction, min, max = "
     << alpha.weightedAverage(mesh_.V()).value()
     << ' ' << min(alpha).value()
     << ' ' << max(alpha).value()
     << endl;
sumAlpha += alpha;
```

This code calculates and manages the volumetric concentration of the continuous phase in a multi-phase flow simulation. It includes boundary conditions and diagnostic information.

3 The Calculation of the settling velocity

Investigating flocculation issues requires a close examination of the PSD, as it plays a crucial role in determining the effects of flocculation on mixture hydrodynamics. T

In shear-induced flows, aggregation of particles results in an increase in floc size and a decrease in density as water becomes trapped within the flocs during the aggregation process. This has a direct impact on the settling velocity of the flocs, which then affects the accuracy of predictions for mixture hydrodynamics. To account for these changes, we use an improved definition of settling velocity that takes into account the floc density by incorporating the primary particle size ($d_{k=2}$) and the fractal dimension (d_f). This updated formula extends the one proposed by Ferguson and Church (2004) and is as follows:

$$u_{kr} = \frac{R_{s,k} g d_k^2}{b_1 \nu_c + (0.75 b_2 R g d_k^3)^{1/2}}, \tag{21}$$

where ν_c is the kinematic viscosity of the carrier fluid (water), and b_1 and b_2 are coefficients that account for particle shape and drag, respectively. $R_{s,k} = (\rho_k - \rho_1)/\rho_k$ represents the submerged specific gravity and highlights the correlation between floc density and settling velocity. $R_{s,k}$ is calculated differently as per Kranenburg (1994) and Strom and Keyvani (2011), as follows:

$$R_{f,k} = R_{s,k} \left(\frac{d_k}{d_2}\right)^{d_f - 3}, \tag{22}$$

By substituting Equation 22 with Equation 21, we arrive at a general explicit formulation for the settling velocity of a floc as follow:

$$u_{kr} = \frac{R g d_k^{d_f - 1}}{b_1 \nu_c d_2^{d_f - 3} + b_2 (0.75 R g d_k^{d_f} d_2^{d_f - 3})^{1/2}}. \tag{23}$$

```
forAll (V0, i)
{
    V0[i] =
        R.value()*g*pow(phasek.d().value(), dff_.
            value()-1)
      / (
         C1_.value()*nuf[i] * pow(d_p_.value(), dff_.
             value()-3 )
       + (
```

```
            C2_.value()*Foam::sqrt
        (
            R.value()*mag(g)*pow(phasek.d().value(),
                dff_.value())* pow(d_p_.value(), dff_.
                value()-3 )
        )
        )
      );
}
```

The implementation of hindered settling, as described in Richardson and Zaki, 1997, is already in place. However, to fully account for hindered settling effects, further development is needed to incorporate the formula introduced by Spearman and Manning, 2017.

References

Biggs, C., & Lant, P. (2002). Modelling activated sludge flocculation using population balances. *Powder Technology*, *124*(3), 201–211.

Chen, W., Fischer, R. R., & Berg, J. C. (1990). Simulation of particle size distribution in an aggregation-breakup process. *Chemical Engineering Science*, *45*(9), 3003–3006. https://doi.org/https://doi.org/10.1016/0009-2509(90)80201-O

Damián, S. M., & Nigro, N. M. (2014). *An extended mixture model for the simultaneous treatment of small-scale and large-scale interfaces* (Publication No. 8) [Doctoral dissertation]. Wiley Online Library.

Elerian, M., Huang, Z., van Rhee, C., & Helmons, R. (2023). Flocculation effect on turbidity flows generated by deep-sea mining: A numerical study. *Ocean Engineering*, *277*, 114250.

Elerian, M., Van Rhee, C., & Helmons, R. (2022). Experimental and numerical modelling of deep-sea-mining-generated turbidity currents. *Minerals*, *12*(5). https://doi.org/10.3390/min12050558

Ferguson, R., & Church, M. (2004). A simple universal equation for grain settling velocity. *Journal of sedimentary Research*, *74*(6), 933–937.

Gillard, B., Purkiani, K., Chatzievangelou, D., Vink, A., Iversen, M. H., & Thomsen, L. (2019). Physical and hydrodynamic properties of deep sea mining-generated, abyssal sediment plumes in the clarion clipperton fracture zone (eastern-central pacific). *Elementa: Science of the Anthropocene*, *7*.

Goeree, J. (2018). *Drift-flux modeling of hyper-concentrated solid-liquid flows in dredging applications* [Doctoral dissertation].

Golzarijalal, M., Zokaee Ashtiani, F., & Dabir, B. (2018). Modeling of microalgal shear-induced flocculation and sedimentation using a coupled cfd-population balance approach. *Biotechnology Progress*, *34*(1), 160–174.

Hounslow, M., Ryall, R., & Marshall, V. (1988). A discretized population balance for nucleation, growth, and aggregation. *AIChE journal*, *34*(11), 1821–1832.

Jeldres, R. I., Concha, F., & Toledo, P. G. (2015). Population balance modelling of particle flocculation with attention to aggregate restructuring and permeability. *Advances in colloid and interface science*, *224*, 62–71.

Kranenburg, C. (1994). The fractal structure of cohesive sediment aggregates. *Estuarine, Coastal and Shelf Science*, *39*(5), 451–460.

Li, X.-Y., & Logan, B. E. (2001). Permeability of fractal aggregates. *Water research*, *35*(14), 3373–3380.

Richardson, J., & Zaki, W. (1997). Sedimentation and fluidisation: Part i. *Chemical Engineering Research and Design*, *75*, S82–S100.

Spearman, J., & Manning, A. J. (2017). On the hindered settling of sand-mud suspensions. *Ocean Dynamics*, *67*, 465–483.

Strom, K., & Keyvani, A. (2011). An explicit full-range settling velocity equation for mud flocs. *Journal of Sedimentary Research*, *81*(12), 921–934.

Van Leussen, W. (1994). Estuarine macroflocs and their role in fine-grained sediment transport. ministry of transport. *Public Works and Water Management, National Institute for Coastal and Marine Management (RIKZ)*.

Veerapaneni, S., & Wiesner, M. R. (1996). Hydrodynamics of fractal aggregates with radially varying permeability. *Journal of Colloid and Interface Science*, *177*(1), 45–57.

Wickramasinghe, S., Han, B., Akeprathumchai, S., Jaganjac, A., & Qian, X. (2005). Modeling flocculation of biological cells. *Powder technology*, *156*(2-3), 146–153.

Winterwerp, J. C. (1998). A simple model for turbulence induced flocculation of cohesive sediment. *Journal of hydraulic research*, *36*(3), 309–326.

Zhang, J.-j., & Li, X.-y. (2003). Modeling particle-size distribution dynamics in a flocculation system. *AIChE Journal*, *49*(7), 1870–1882.