Modification of Stochastic Model in Lagrangian Tracking Method

Developed for OpenFOAM-4.0
Requires: ccm26ToFoam

Author:
Jundi Xu
Chalmers University of Technology
jundi@student.chalmers.se
xujundi58@gmail.com

Peer reviewed by:
Mahish Mohan
Ebrahim Ghahramani
Håkan Nilsson

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

January 10, 2017
Learning Outcome

The reader will learn:

How to use it:

- How to use icoUncoupledKinematicParcelFoam.
- How to use stochastic model.
- How a case setup for the specified solver and stochastic model.

The theory of it:

- How to generate fluctuating velocity in RANS framework in order to model turbulent dispersion phenomenon.
- Why the Discrete Random Walk (DRW) model is not suitable for anisotropic flows and how to improve it.
- What is the main shortcoming of DRW model.
- How to improve the DRW model, in other way, how to release its shortcoming.

How it is implemented:

- What is the tracking mechanism in Lagrangian tracking method.
- How to implement to isotropic assumption of DRW model into stochastic model, i.e. in StochasticDispersionRAS.C file.

How to modify it:

- How to modify the Lagrangian tracking solver.
- How to modify the stochastic model according to the theoretical equations.
- How to compile the new stochastic model.
- How to convert the mesh file from STAR-CCM+ into OpenFOAM.
Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- Fundamentals of RANS model and stochastic model
- It is strongly recommended to gain a brief insight into the physics of the particle deposition from the following journal (if accessible):

## Contents

1. **Background and Motivation** .................................................. 1
   1.1 *Lagrangian* and *Eulerian* .............................................. 1
   1.2 Classification of different phases ........................................ 1
   1.3 Governing equation .......................................................... 1
      1.3.1 Fundamental assumption .............................................. 1
      1.3.2 Forces acting on particles ............................................ 2
   1.4 Turbulence effect on particle dispersions .............................. 3
   1.5 Stochastic model ........................................................... 4
      1.5.1 *DRW* .................................................................. 4
      1.5.2 Improvement of *DRW* ................................................. 5

2. **Insight of Solvers in OpenFOAM-4.0** ................................... 7
   2.1 Introduction ........................................................................... 7
   2.2 Description and comparison of two solvers ............................. 7
      2.2.1 Comparison of solvers .................................................. 7
      2.2.2 Main difference of two Cloud classes .............................. 9
      2.2.3 Tracking mechanism for particles ................................... 11
   2.3 Stochastic model ................................................................... 13
   2.4 Conclusion ............................................................................. 16

3. **Modification** ....................................................................... 17
   3.1 Create new solver–icolUncoupledKinematicNocollisionFoam .......... 17
   3.2 Create new stochastic model – myStochasticModel ................... 18
   3.3 Compile new class myStochasticModel .................................... 24

4. **Results from Test Case** ....................................................... 27
   4.1 Pre-requirements ................................................................... 27
   4.2 Primitive preparation .......................................................... 27
      4.2.1 Basic case .................................................................... 27
      4.2.2 Vertical pipe ............................................................... 27
      4.2.3 Modification the initial parameters in the 0 directory .......... 29
      4.2.4 Modification in the constant directory ............................ 30
      4.2.5 Modification in the system directory ............................... 30
      4.2.6 Steady flow ................................................................. 32
   4.3 Test icolUncoupledKinematicNocollisionFoam with myStochasticModel ......................................................... 32
      4.3.1 Setup the test case ....................................................... 32
      4.3.2 Modification of controlDict ......................................... 33
      4.3.3 Modification of kinematicCloudPosition .......................... 34
      4.3.4 Modification of kinematicCloudPropertie ....................... 34
      4.3.5 Run simulation ............................................................ 36
   4.4 Simulation result ..................................................................... 36
      4.4.1 Comparison between myStochasticModel and StochasticDispersionRAS ......................................................... 36
Chapter 1

Background and Motivation

1.1 *Lagrangian* and *Eulerian*

- *Lagrangian* specification of the flow field
  In classical field theory, the *Lagrangian* specification of the field is a way of looking at fluid motion where the observer follows an individual fluid parcel as it moves through space and time [2].

- *Eulerian* specification of the flow field
  The *Eulerian* specification of the flow field is a way of looking at fluid motion that focuses on specific locations in the space through which the fluid flows as time passes [3].

Dispersed particle-laden flows could be found in most technical applications, such as industrial, environmental and medical fields, and Computational Fluid Dynamics (CFD) offers a relatively accurate and reliable method of attaining an insight into the interaction amongst a wide range of fluid flows.

1.2 Classification of different phases

The order of coupling between the dispersed and continuous phase is initially determined by the volume fraction of the solid material, $\alpha_p = V_p/V$ [4], see Figure 1.1, the couplings could be divided into three ways:

1. One-way coupling, for highly dilute dispersed flows, when $\alpha_p \leq 10^{-6}$ and the flow of the carrier fluid influences the particle trajectories, the influence from particles to the flow is negligible.

2. Two-way coupling, for dilute dispersed flows, when $10^{-3} < \alpha_p < 10^{-6}$, it is necessary to account for the influence of the particles on the fluid flow [5].

3. Four-way coupling, for dense dispersed flows, when $\alpha_p \geq 10^{-3}$, particle spacing is low (smaller than about 10 particle diameters) and the transport of particles is influenced by collisions [5]. Therefore the particles have influences both on the continuum flow and the other particles.

1.3 Governing equation

1.3.1 Fundamental assumption

For the isothermal, incompressible, turbulent flow of a Newtonian fluid, the continuity and momentum equation of RANS are given by

$$ \nabla \cdot \overline{U} = 0 $$

(1.1)
1.3. GOVERNING EQUATION

CHAPTER 1. BACKGROUND AND MOTIVATION

Figure 1.1: Classification of coupling schemes and interaction between particles and turbulence provided by Elghobashi (1994)\[6\] for (1) one-way coupling, (2) two-way coupling where particles enhance turbulence production, (3) two-way coupling where particles enhance turbulence dissipation, and (4) four-way coupling.

\[
\frac{\partial \bar{U}_c}{\partial t} + \rho_c(\bar{U}_c \cdot \nabla)\bar{U}_c = -\nabla p + \mu \Delta \bar{U}_c - \nabla \cdot \tau_{RS} + f_D \tag{1.2}
\]

where \(\bar{U}_c\) is the Reynolds-averaged flow velocity, \(\rho_c\) is the fluid density, \(p\) is pressure, \(\tau\) is the fluid density, \(\mu\) is the dynamic viscosity and \(f_D\) is the additional body forces. The Reynolds stresses, \(\tau_{RS} = (\rho_c \bar{U}_c' \bar{U}_c')\), are typically modeled using an eddy-viscosity approach.

1.3.2 Forces acting on particles

A spherical particle \(P\) is defined by its position \(x_P\), diameter \(D_P\), velocity \(U_P\) and density \(\rho_P\). The mass of the particle \(P\) is \(m_P = \frac{1}{6} \rho_P \pi D_P^3\). For the dispersed phase, the particle motion is solved by integrating the force balance, which is written in a Lagrangian frame on the particles as:

\[
\frac{dx_P}{dt} = U_P \tag{1.3}
\]

\[
m_P \cdot \frac{dU_P}{dt} = \sum F_i \tag{1.4}
\]

The \(F_i\) represents the sum of all relevant forces.

\[
\sum F_i = F_D + F_G = F_D + m_P g \tag{1.5}
\]

In dilute flow, the dominant force acting on the particle is drag force \(F_D\) from the fluid phase: neglecting particle's Magnus force (assuming that particle rotation is small compare to particle translation) and other forces such as added mass, Basset history term, buoyancy force(since the particle is treated as a point)[7][4].

The particle Reynolds number is defined as:

\[
Re_P = \frac{\rho_c D_P |\bar{U}_c - U_P|}{\mu_f} \tag{1.6}
\]

The drag force can be expressed as:

\[
F_D = m_P \frac{\bar{U}_c - U_P}{\tau_P} \tag{1.7}
\]
The relaxation time $\tau_P$ of the particles is the time it takes for a particle to respond to changes in the local flow velocity, and it is given by

$$\tau_P = \frac{4}{3} \frac{\rho_P D_P}{\rho_c C_D |U_c - U_P|} \quad (1.8)$$

The drag coefficient, $C_D$, can be calculated via a non-linear function in the dependency of the particle Reynolds number as:

$$C_D = \frac{24}{Re_P} \quad Re_P < 1 \quad (1.9a)$$

$$C_D = \frac{24}{Re_P} \left(1 + \frac{3}{16} Re_P^{0.687}\right), \quad 1 < Re_P < 5 \quad (1.9b)$$

$$C_D = \frac{24}{Re_P} \left(1 + 0.15 Re_P^{0.687}\right), \quad 5 < Re_P < 1000 \quad (1.9c)$$

$$C_D = 0.44 \quad Re_P > 1000 \quad (1.9d)$$

where the Reynolds number for a particle moving in a fluid is expressed as

$$Re_P = \frac{\rho D_P |U_c - U_P|}{\mu_f} \quad (1.10)$$

The drag coefficients of spheres, $C_D$, varies with the Reynolds numbers in a highly nonlinear fashion. The drag coefficient can be calculated based on four regions of particle Reynolds number: low Reynolds number region ($Re_P < 1$); transient region ($1 < Re_P < 5$); turbulent region ($5 < Re_P < 1000$); and high turbulent region ($Re_P > 1000$). Obviously, drag coefficient in the transient region is the most complicated and several equations were proposed based on regressions of experimental data\[8\][9]. Most of these equations are sufficiently accurate and generally with 4% $Re_P < 800$ and 7% $Re_P < 1000$\[1\].

Inserting Eqn.(1.8) into Eqn.(1.7), yields

$$F_D = \frac{3 \rho_c m_P}{4 \rho_P} \cdot C_D(U_c - U_P)|U_c - U_P| \quad (1.11)$$

And Eqn. (1.5) turns to be

$$\sum F_i = F_D + F_G = \frac{3 \rho_c m_P}{4 \rho_P} \cdot C_D(U_c - U_P)|U_c - U_P| + m_P g \quad (1.12)$$

1.4 Turbulence effect on particle dispersions

In turbulent flows, DNS and LES coupled to Lagrangian particle tracking (LPT) offer the most rigorous way of treating particle dispersion in Euler/Lagrange frameworks, but they are very time-consuming and sometimes even difficult (impossible) to apply in general geometries. Therefore, the RANS seems to be more flexible and available in almost all cases.

Recall that in the assumption of the Reynolds-averaged Navier-Stokes equations the instantaneous variables of turbulent flow is to be decomposed into a mean value and a fluctuating value, i.e.

$$U_i = \overline{U_i} + U'_i \quad \text{and} \quad p = \overline{p} + p'$$

In the RANS framework, some available turbulent models (in this case is k-$\epsilon$ model) in CFD produce averaged fluid field quantities.
1.5 Stochastic model

In the event that a resolved-eddy simulation (such as direct numerical simulation or large eddy simulation) is not feasible for the continuous phase, an unresolved-eddy flow simulation, such as a Reynolds-averaged Navier-Stokes (RANS) solution, can be used to obtain results of mean particle diffusion. However, the RANS equations can not provide the instantaneous fluid velocity \( U_i = \bar{U}_i + U'_i \). Therefore, the fluctuating velocity \( U' \) has to be estimated by stochastic model in order to model turbulent dispersion phenomenon.

Whereas the continuous phase is typically described in an Eulerian sense, the particle field can either be treated in an Eulerian or Lagrangian fashion. Lagrangian diffusion models employ the mean flow properties from the RANS solution to compute a large number of particle trajectories and obtain mean diffusion statistics. This methodology can be desirable compared to Eulerian methods because it avoids an empirical particle Schmidt number formulation and can robustly handle particle-wall interaction phenomenon [10].

Generally, the flow field is computed in advance, then a large number of particles are injected in the field, and their trajectories determined by following individual particles until they are removed from the flow stream or leave the computational domain [11].

The Lagrangian stochastic computational models fall into three main categories. These can range from the least computationally intensive to the most computationally intensive as:

- Discrete Random Walk (DRW) models
- Continuous Random Walk (CRW) models
- Stochastic Differential Equation (SDE) methods.

The first two models simulate turbulence by assuming a functional form for the velocity perturbations based on the root-mean-square values of the turbulence. The more complex SDE methods employ a Langevin equation for the instantaneous velocity, and require modeling of the triple-velocity moments. In addition, wall interactions cannot be as robustly specified with the SDE methodology [10]. As such this technique is still in the developmental stage and is not yet widely used for most engineering applications [12].

1.5.1 DRW

In the discrete random walk (DRW) model, or "Eddy Interaction Model (EIM), the interaction of a particle with a succession of discrete stylized fluid phase turbulent eddies is simulated, see Figure 1.2. EIM is a popular model used to describe particle dispersion in turbulent flow, it is a stochastic random walk treatment where the particles are made to interact with the instantaneous velocity field \( U_i = \bar{U}_i + U'_i \) [11]. Stated that the eddies are isotropic, meaning that in average the eddies have no preferred direction, i.e the fluctuations in all direction are the same so that

\[ U'_1^2 \approx U'_2^2 \approx U'_3^2 \]  

(1.14)

This assumption is not true instantaneously, i.e.

\[ U'_1 \neq U'_2 \neq U'_3 \]  

(1.15)

Which leads to large over-predictions of particle deposition rates even for simple flows such as those in pipes or parallel channels.

Here, the Discrete Random Walk Model (DRW) is in focus. A detailed and comprehensive review is beyond the current tutorial. Only a brief description is presented.
1.5. STOCHASTIC MODEL

CHAPTER 1. BACKGROUND AND MOTIVATION

1.5.2 Improvement of DRW

The DRW models the turbulent dispersion of particles as a succession of interactions between a particle and eddies. Eddies have finite lengths and lifetimes in the model of turbulent dispersion of particles, which based on the model of Gosman and Ioannides is given by

\[ \tau_e = 2T_L \]  

\[ L_e = (C_L^k) \frac{3/4 k^{3/2}}{\varepsilon} \]  

where \( T_L \) is the Lagrangian integral time which equals integral time scale for tracer particles:

\[ T_L = C_L^k \frac{k}{\varepsilon} \]

In the RANS framework, the specific model (k-\( \varepsilon \)) yields the mean value velocity and the fluctuating velocity components are based on randomly distributed Gaussian variables whose rms (root mean square) values are equal and deduced from the turbulent kinetic energy \( k \) is given by

\[ \sqrt{U_1'^2} = \sqrt{U_2'^2} = \sqrt{U_3'^2} = \sqrt{\frac{2k}{3}} \]  

The fluctuating velocity as seen by a particle is given by

\[ U_1' = \lambda_1 \sqrt{U_1'^2} \]

\[ U_2' = \lambda_2 \sqrt{U_2'^2} \]

\[ U_3' = \lambda_3 \sqrt{U_3'^2} \]  

where \( \lambda \)'s are Gaussian random variables with 0 mean and unit standard deviation.

However, the most obvious and significant shortcoming of the DRW model is the isotropic turbulence assumption, because it is not realistic as turbulence is very anisotropic in boundary layers. This equation introduces a large over-prediction for the wall normal component \( U_2', y \)-dir, in the boundary layer. Furthermore, the time scale should be different along the location of particles.

1.5.2 Improvement of DRW

The improvement focuses on fluctuating components and it could be divided into two types: the one outside the boundary layer and the one inside the boundary layer.

1. For outside the boundary layer \( (y^+ > 80) \): using the default model, since the turbulent is isotropic.
2. For inside the boundary layer ($y^+ < 80$):

Taking into account anisotropy and using DNS data in channel data results [13], three functions $f_1$, $f_2$ and $f_3$ introduced for the determination of the local fluctuating velocity:

\[
\begin{align*}
U'_1 &= f_1 N_1 \sqrt{\frac{2k}{3}} \\
U'_2 &= f_2 N_2 \sqrt{\frac{2k}{3}} \\
U'_3 &= f_2 N_2 \sqrt{\frac{2k}{3}}
\end{align*}
\]  

(1.21)

where $N_1$, $N_2$ and $N_3$ are random numbers generated from a Gaussian probability density function and the functions are expressed as:

\[
\begin{align*}
f_1 &= 1 + 0.285(y^+ + 6) \exp[-0.455(y^+ + 6)^{0.53}], \\
f_2 &= 1 - \exp(-0.02y^+), \\
f_3 &= \sqrt{3 - f_1^2 - f_2^2},
\end{align*}
\]  

(1.22)

The definition of non-dimensional wall distance for a wall-bounded flow, $y^+$ is given by

\[
y^+ = \frac{u^* y}{\nu}
\]  

(1.23)

where $y$ is the normal distance to the nearest wall, $\nu$ is the local kinematic viscosity of the fluid, defined as

\[
\nu = \frac{\mu}{\rho}
\]  

(1.24)

where $\mu$ is the dynamic viscosity and $\rho$ is the density.

with $u^*$, friction velocity, defined as

\[
u^* = \sqrt{\frac{\tau_w}{\rho}}
\]  

(1.25)

as the wall shear stress, $\tau_w$

\[
t_w = \mu \left(\frac{\partial u}{\partial y}\right)_{y=0}
\]  

(1.26)

where $u$ is the flow velocity parallel to the wall and $y$ is the distance to the wall.

Inserting Eqn. (1.24), (1.25), (1.26) into Eqn. (1.23), obtained

\[
y^+ = \frac{y \cdot \sqrt{\nu \cdot (\partial u / \partial y)_{y=0}}}{\nu}
\]  

(1.27)

This is the fundamental methodology which will be implemented into the modified stochastic model in OpenFOAM.
Chapter 2

Insight of Solvers in OpenFOAM-4.0

2.1 Introduction

This tutorial is focusing on one-way coupling, i.e. the influence of a particle on the fluid flow is neglected in this flow regime, which means that the discrete phase patterns are based on a fixed continuous phase flow field.

In OpenFOAM, some promising solvers will be studied which could be used with the modified stochastic model for one-way coupling without collision, so the first thing is to understand the function of several solvers. The following sections will introduce icoUncoupledKinematicParcelFoam and uncoupledKinematicParcelFoam and study deeper if necessary.

2.2 Description and comparison of two solvers

This section covers the description and comparison of solvers icoUncoupledKinematicParcelFoam and uncoupledKinematicParcelFoam. The task is to find a simpler way to modify the code as one-way coupling Lagrangian tracking method without particle collision.

2.2.1 Comparison of solvers

To understand the difference of files included in both solvers, go to icoUncoupledKinematicParcelFoam solver directory.

```
0F4x
sol
cd lagrangian/icoUncoupledKinematicParcelFoam
tree -L 2
```

yields the structure tree under icoUncoupledKinematicParcelFoam

```
|-- createFields.H
|-- createNonInertialFrameFields.H
|-- icoUncoupledKinematicParcelDyMFoam
    |-- icoUncoupledKinematicParcelDyMFoam.
    |-- Make
```
2.2. DESCRIPTION AND COMPARISON OF TWO SOLVERS

CHAPTER 2. INSIGHT OF SOLVERS IN OPENFOAM-4.0

|-- icoUncoupledKinematicParcelFoam.C
  -- Make
    |-- files
    |-- options

As the same, go to uncoupledKinematicParcelFoam solver directory.

```
sol
cd lagrangian/uncoupledKinematicParcelFoam
ls
```

This directory contains files createFields.H and uncoupledKinematicParcelFoam.C. The function of createFields.H is to initialize the parameters which are used in uncoupledKinematicParcelFoam.C.

To investigate the differences on top level solver, go to icoUncoupledKinematicParcelFoam solver directory.

```
sol
cd lagrangian/icoUncoupledKinematicParcelFoam
vim icoUncoupledKinematicParcelFoam.C
```

Also go to uncoupledKinematicParcelFoam solver directory

```
sol
cd lagrangian/uncoupledKinematicParcelFoam
vim uncoupledKinematicParcelFoam.C
```

By opening the both top level solver files: icoUncoupledKinematicParcelFoam and uncoupledKinematicParcelFoam, it is seen that both are for compressible flow solvers using runTimeloop. If you focus on the header, a major difference is that they include two different types of clouds. Compared with uncoupledKinematicParcelFoam.C, the main difference is icoUncoupledKinematicParcelFoam.C includes the collidingModel, but the former one is for thermal transport.

Eventually, the idea is to combine different parts of these two solvers for a new one with singlePhaseTransportModel and basicKinematicCloud. As can be seen from the following table, the difference is the red part and it will be investigate later.

<table>
<thead>
<tr>
<th>icoUncoupledKinematicParcelFoam.C</th>
<th>uncoupledKinematicParcelFoam.C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &quot;fvCFD.H&quot;</code></td>
<td><code>#include &quot;fvCFD.H&quot;</code></td>
</tr>
<tr>
<td><code>#include &quot;singlePhaseTransportModel.H&quot;</code></td>
<td><code>#include &quot;psiThermo.H&quot;</code></td>
</tr>
<tr>
<td><code>#include &quot;turbulentTransportModel.H&quot;</code></td>
<td><code>#include &quot;turbulentTransportModel.H&quot;</code></td>
</tr>
<tr>
<td><code>#include &quot;basicKinematicCollidingCloud.H&quot;</code></td>
<td><code>#include &quot;basicKinematicCloud.H&quot;</code></td>
</tr>
</tbody>
</table>

Additionally, their main function is really similar, the while loop of icoUncoupledKinematicParcelFoam.C is presented below (the other one is similar). kinematicCloud.evolve() is the main function and it will be discussed in later section 2.2.3.

```
icoUncoupledKinematicParcelFoam.C

63   while (runTime.loop())
64    {
65      Info<< "Time = " << runTime.timeName() << nl << endl;
```
2.2. DESCRIPTION AND COMPARISON OF TWO SOLVERS

CHAPTER 2. INSIGHT OF SOLVERS IN OPENFOAM-4.0

The further investigation is comparing the differences between these two pairs: basicKinematicCollidingCloud.H and basicKinematicCloud.H, singlePhaseTransportModel.H and psiThermo.H. But the discussion of the latter pair is not of importance for this case and will not present there.

2.2.2 Main difference of two Cloud classes

This section will present the main difference between basicKinematicCollidingCloud.H and basicKinematicCloud.H.

Go to find the local directory of each file .C in /src directory according to the following command:

1. For basicKinematicCollidingCloud.H:

```
src
find -name "basicKinematicCollidingCloud.H"
vim ./lagrangian/intermediate/clouds/derived/basicKinematicCollidingCloud/
```

2. use the same way to find basicKinematicCloud.H location:

The difference is shown in the declaration part. The red part is the main different and the blue one will be discussed deeply in the next section:

```
#include "Cloud.H"
#include "Cloud.H"
#include "Cloud.H"
#include "basicKinematicParcel.H"
#include "basicKinematicParcel.H"
```

It can be seen that from the following codes in basicKinematicCloud.H, basicKinematicCloud is a short name (typedef) for different layers of Cloud on top of each other. KinematicCloud is layered on base cloud class Cloud (templated on particle type). In every cloud layer, new functionalities (models) are added to the base cloud layer. The templated Cloud class is instantiated with class basicKinematicParcel as type parameter.

```
namespace Foam {

typedef KinematicCloud<Cloud<basicKinematicParcel>> basicKinematicCloud;
}
```
Similarly, for `basicKinematicCollidingCloud.H`, the `basicKinematicCollidingCloud` is a short name for different layers of clouds on top of each other, since it includes collision model, the sub-model under `CollidingCloud` (see next paragraph) is added as the highest top level layered on `KinematicCloud` which layered on the base cloud class `Cloud`. The template `Cloud` class is instantiated with class `basicKinematicCollidingParcel`, the definition of `basicKinematicCollidingParcel` will be described briefly as following.

```cpp
42 namespace Foam
43 {
44   typedef CollidingCloud
45     <
46     KinematicCloud
47     <
48     Cloud
49     <
50     basicKinematicCollidingParcel
51     >
52     > basicKinematicCollidingCloud;
53 }
```

The description of `CollidingCloud.H` as below, the description indicates that the function is to add collisions to `KinematicCloud`:

```
Class Foam::CollidingCloud
Description Adds collisions to kinematic clouds
```

Now, it is needed to investigate the definition of `basicKinematicParcel` class. As a comparison, both `basicKinematicCollidingParcel.H` and `basicKinematicParcel.H` are presented here:

```
#include "contiguous.H"
#include "particle.H"
#include "KinematicParcel.H"
#include "CollidingParcel.H"
```

Firstly, you can look up the main function of `basicKinematicCollidingParcel` in `.H`:

```cpp
typedef CollidingParcel<KinematicParcel<particle>>
   basicKinematicCollidingParcel;
```

According to the above table, the main difference is the red one, `CollidingParcel.H`, its description shows the function of `CollidingParcel.H`, adding `collision modelling`:

```
Description
   Wrapper around kinematic parcel types to add collision modelling
```
By contrast, class `basicKinematicParcel` without collision model and it is equal to class `KinematicParcel` with specified parameter as `particle`.

```cpp
typedef KinematicParcel<particle> basicKinematicParcel;
```

### 2.2.3 Tracking mechanism for particles

As mentioned in Section [2.2.1](#), the core member function for both solvers is the `kinematicCloud<evolve>` member function. Therefore, class `kinematicCloud` needs to be investigated deeply to learn about the tracking mechanism for particles in the source file.

For `kinematicCloud.H`, as the header shows:

```
Class Foam::kinematicCloud
Description Virtual abstract base class for templated KinematicCloud
SourceFiles kinematicCloud.H
```

Therefore, the class `kinematicCloud` is just a virtual abstract class without any definition, but it is defined in the template class `KinematicCloud`.

In the template class `KinematicCloud`, the `evolve` member function has been defined in line 675-685, (refer to Appendix 6.1). As `CloudType` is `basicKinematicParcel` under the class `Cloud` of `KinematicCloud`, which means whenever `CloudType` used in `KinematicCloud`, it is equivalent to `Cloud<basicKinematicParcel>`.

1. Line 683: it calls another member function `solve(td)`, `td` is reference of `TrackData`, thus refer to line 89-126 for the next step.

2. Line 89-126: since the case is focusing on no coupled motion, the `if – else` loop will execute function `evolveCloud()` no matter what the value of `solution_.steadyState()` is (true or false). Here, to call member function `evolveCloud()`, go to line 174 -215.

3. Line 183-215: when `solution_.transient()` is true, line 202 and line 204 will be executed, otherwise, line 213 `CloudType::move` will be performed. This `CloudType` has be defined as `Cloud<basicKinematicParcel>` under class `Cloud`. Thus, go to file `Cloud.C` to investigate the member function `CloudType::move`.

4. Refer to the code `Cloud.C` in Appendix 6.2
   - Line 214: Initialise the stepFraction moved for the ALL particles;
   - Line 242-378: ”while loop” shows how to transfer the particles through the `move` member function;
2.2. DESCRIPTION AND COMPARISON OF TWO SOLVERS

- Line 251-295: this loop is for all particles, line 256-294 shows how to move the particle, if it has passed through an inlet or outlet, it will be kept, otherwise, the particle will be deleted. In line 253, p is defined as reference of pIter() under the class ParticleType. The class ParticleType has been defined as basicKinematicParcel in section 2.2.2, it will be discussed further in next section. Line 329-344 shows how to define the transferred particle.

5. Back to the class KinematicCloud, line 197: injectors_.inject(td) will implement the injected particles into flow.

6. Line 202: td.cloud().motion(td) will call the motion function, it will call the class KinematicParcel, and the motion member function as following:

```cpp
688 template<class CloudType>
689 template<class TrackData>
690 void Foam::KinematicCloud<CloudType>::motion(TrackData& td) {
691     td.part() = TrackData::tpLinearTrack;
692     CloudType::move(td, solution_.trackTime());
693     updateCellOccupancy();
694 }
```

For the no collision scenario (i.e this case), only the motion member function under class KinematicParcel will be called to calculate the particle velocity. The KinematicParcel.H declares the components:

```cpp
279     // Protected Member Functions
280     //-- Calculate new particle velocity
281     template<class TrackData>
282     const vector calcVelocity (TrackData& td,
283         const scalar dt,       // timestep
284         const label celli,    // owner cell
285         const scalar Re,      // Reynolds number
286         const scalar mu,      // local carrier viscosity
287         const scalar mass,    // mass
288         const vector& Su,     // explicit particle momentum source
289         vector& dUTrans,     // momentum transfer to carrier
290         scalar& Spu          // linearised drag coefficient
291     ) const;
```

And KinematicParcel.C shows more detailed for the particle profile.

```cpp
39     template<class ParcelType>
40     template<class TrackData>
41     void Foam::KinematicParcel<ParcelType>::setCellValues (TrackData& td,
42         (TrackData& td,
```

12
2.3 STOCHASTIC MODEL

The above codes will calculate \( \rho_c, U_c, \mu_c \) for the flow. In line 68-77, the member function \texttt{update} is called to add the turbulent fluctuation velocity to the mean flow velocity based on the dispersion model. Recall the section 1.4, \( U_c = U_c + U_{turb} \), till now, the description of the particle tracking is completed.

2.3 Stochastic model

Since the collision scenarios will not happen in our case, the \texttt{StochasticDispersionRAS} model seems to be a promising candidate. Find \texttt{StochasticDispersionRAS.H} file and look up this file to understand the whole procedure applied in turbulence model:

```cpp
const scalar dt,
const label celli
{
    tetIndices tetIs = this->currentTetIndices();
    rhoc_ = td.rhoInterp().interpolate(this->position(), tetIs);
    if (rhoc_ < td.cloud().constProps().rhoMin())
    {
        if (debug)
        {
            WarningInFunction
            << "Limiting observed density in cell " << celli << " to "
            << td.cloud().constProps().rhoMin() << nl << endl;
        }
        rhoc_ = td.cloud().constProps().rhoMin();
    }
    rhoc_ = td.cloud().constProps().rhoMin();
    Uc_ = td.UInterp().interpolate(this->position(), tetIs);
    muc_ = td.muInterp().interpolate(this->position(), tetIs);
    // Apply dispersion components to carrier phase velocity
    Uc_ = td.cloud().dispersion().update
    (dt, // time step
     celli, // cell number
     U_, // particle mean velocity
     Uc_, // flow mean velocity
     UTurb_, // flow fluctuating velocity in turbulence
     tTurb_ // time fluctuating velocity in turbulence
    );
}
```
2.3. STOCHASTIC MODEL

CHAPTER 2. INSIGHT OF SOLVERS IN OPENFOAM-4.0

where $\sigma$ is defined below

```cpp
#ifndef StochasticDispersionRAS_H
#define StochasticDispersionRAS_H

#include "DispersionRASModel.H"

namespace Foam {
    /*---------------------------------------------------------------------------*
     * Class StochasticDispersionRAS Declaration
     *---------------------------------------------------------------------------*/

    template<class CloudType>
    class StochasticDispersionRAS
    :
      public DispersionRASModel<CloudType>
    {
      public:
        
        // Runtime type information
        TypeName("stochasticDispersionRAS");

        // Constructors

        // Construct from components
        StochasticDispersionRAS(const dictionary& dict, CloudType& owner);

        // Construct copy
        StochasticDispersionRAS(const StochasticDispersionRAS<CloudType>& dm)

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

        // Destructor
    
        virtual ~StochasticDispersionRAS();

        // Member Functions
```
As the description shows, the Gaussian random number distribution with variance \( \sigma \) as generating the fluctuating velocity part has been implemented in this model. In OpenFOAM, the spatial randomness of turbulence is described as additional random vector \( \mathbf{d} \) for the calculation of \( U' \):

\[
U' = \zeta \mathbf{d} \sigma \tag{2.1}
\]

with the turbulent kinetic energy \( k \) and dissipation \( \varepsilon \) provided by the turbulence model, here, \( k-\varepsilon \) equations are used to get \( k \), look up the next code, line 78 - line 85.

From the beginning, line 51 indicates that class StochasticDispersionRAS is inherited from class DispersionRASModel, which is the sub-class of the DispersionModel class (RAS means Reynolds-averaged simulation).

The member function update (line 84) will calculate the fluctuating component for velocity and time, i.e \( \mathbf{U_{Turb}} \), \( t_{Turb} \). Now go to the file StochasticDispersionRAS.C where the exact mathematical equation applied.

In its main function:
2.4. Conclusion

This section described the main differences between icoUncoupledKinematicParcelFoam and uncoupledKinematicParcelFoam. The main classes and member functions also are explained in detail. Moreover, this section depicts the promising stochastic model which could be modified later.

The next section will focus on creating a new solver which combines the icoUncoupledKinematicParcelFoam and the uncoupledKinematicParcelFoam, and the modification of StochasticDispersion-RAS which turns it from isotropic turbulent model to be anisotropic turbulent model in the boundary layer.
Chapter 3

Modification

3.1 Create new solver–icoUncoupledKinematicNocollisonFoam

In $WM_PROJECT_USER_DIR$ a new directory called icoUncoupledKinematicNocollisonFoam is created as a copy of icoUncoupledKinematicParcelFoam which is originally located in $FOAM_SOLVERS/lagrangian$.

```
OF4x
  cp -r $FOAM_SOLVERS/lagrangian/icoUncoupledKinematicParcelFoam/
  $WM_PROJECT_USER_DIR/icoUncoupledKinematicNocollisonFoam
  →
  cd $WM_PROJECT_USER_DIR/icoUncoupledKinematicNocollisonFoam
  mv icoUncoupledKinematicParcelFoam.C icoUncoupledKinematicNocollisonFoam.C
```

As discussed in the previous section, the header file `#include "basicKinematicCollidingCloud.H"` should be replaced by `#include "basicKinematicCloud.H"` since the collision will not occur in this case. The modification is done as

```
cd icoUncoupledKinematicNocollisonFoam
  sed -i s/basicKinematicCollidingCloud.H/basicKinematicCloud.H/g
  icoUncoupledKinematicNocollisonFoam.C
  sed -i s/basicKinematicCollidingCloud/basicKinematicCloud/g creatFields.H
```

The files file under the Make directory should be changed as

```
Make/files:

  icoUncoupledKinematicNocollisonFoam.C
  EXE = $(FOAM_USER_APPBIN)/icoUncoupledKinematicNocollisonFoam
```

The options file under the Make directory does not need to be changed because the header file `basicKinematicCloud.H` has already been included in the directory:

```
EXE_INC =-I$(LIB_SRC)/lagrangian/intermediate/lnInclude
```

Then, by compiling, the icoUncoupledKinematicNocollisonFoam solver is created:

```
wclean
wmake
```
3.2 Create new stochastic model – myStochasticModel

To improvement of the DRW model as described in section 1.5.2, the code should be modified from line 99 - line 108. Therefore, a new class named myStochasticModel is created based on the class StochasticDispersionRAS.

The class StochasticDispersionRAS is located in $FOAM_SRC/lagrangian/turbulence/submodels/Kinematic/DispersionModel/StochasticDispersionRAS and follow the steps below to copy it and do the initial modification to create a new class.

```bash
src
  cp -r $FOAM_SOLVERS/lagrangian/turbulence/WM_PROJECT_USER_DIR
  cd $WM_PROJECT_USER_DIR
  cp -r turbulence/submodels/Kinematic/DispersionModel/StochasticDispersionRAS/
  turbulence/submodels/Kinematic/DispersionModel/myStochasticModel
  mv StochasticDispersionRAS.C myStochasticModel.C
  mv StochasticDispersionRAS.H myStochasticModel.H
  sed -i s/StochasticDispersionRAS/myStochasticModel/g myStochasticModel.H
  sed -i s/StochasticDispersionRAS/myStochasticModel/g myStochasticModel.C
```

Somehow, the sed command does not work well for myStochasticModel.H, so you have to go inside the file to modify the code manually in line `TypeName`:

```
TypeName("myStochasticModel");
```

To implement Eqn. 1.21, the value of $y^+$ should be calculated at every particle position. To achieve Eqn. 1.27, the turbulence velocity vector $U_w$, the distance to the nearest wall $y$, kinematic viscosity $\nu_w$ should be known before the calculation. These three variable can be obtained via class `turbulenceModel`. The member function `snGrad` yields the gradient $(\partial u/\partial y)_{y=0}$.

Since the class DispersionRASModel is setup to `const`, the object $yP_-$ under class `volScalarField` could not be assigned a new value, therefore, `const_cast` has to be used here to remove this limitation.

```bash
vim myStochasticModel.H
```

The following code shows the final modification.

The modification in myStochasticModel.H is:

```bash
myStochasticModel.H

#ifndef myStochasticModel_H
#define myStochasticModel_H

#include "DispersionRASModel.H"

#include "Dis mult's dispersion model.

The modification in myStochasticModel.H is:

```
namespace Foam
{

/*@---------------------------------------------------------------------------*
  Class myStochasticModel Declaration
  *---------------------------------------------------------------------------*/

template<class CloudType>
class myStochasticModel
{
public DispersionRASModel<CloudType>
{
public:

  // Runtime type information
  TypeName("myStochasticModel");

  // declaration of yPlus
  volScalarField yP_;

  // Constructors

    // Construct from components
    myStochasticModel(const dictionary& dict, CloudType& owner);

    // Construct copy
    myStochasticModel(const myStochasticModel<CloudType>& dm);

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

    // Destructor
    virtual ~myStochasticModel();

  // Member Functions

    // Update (disperse particles)
    virtual vector update
    {
      const scalar dt,
      const label celli,
      const vector& U,
      const vector& Uc,
The modification in myStochasticModel.C is:

```cpp
#include "myStochasticModel/H"
#include "constants.H"
#include "nearWallDist.C"

using namespace Foam::constant::mathematical;

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

template<class CloudType>
Foam::myStochasticModel<CloudType>::myStochasticModel
(
    const dictionary& dict,
    CloudType& owner
)
:
    DispersionRASModel<CloudType>(dict, owner),
    yP_
    (IOobject
        ("yP",
         this->owner_.mesh().time().timeName(),
         this->owner_.mesh(),
         IOobject::NO_READ
    ),
```
3.2. CREATE NEW STOCHASTIC MODEL – MYSTOCHASTICMODEL

Chapter 3. Modification

IOobject::NO_READ,
IOobject::AUTO_WRITE
),
this->owner_.mesh(),
dimensionedScalar("yP", dimless, 0.0)
}

{

template<class CloudType>
Foam::myStochasticModel<CloudType>::myStochasticModel
(const myStochasticModel<CloudType>& dm)
:
DispersionRASModel<CloudType>(dm),
yP_(dm.yP_)
{}

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

template<class CloudType>
Foam::myStochasticModel<CloudType>::~myStochasticModel()
{

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

template<class CloudType>
Foam::vector Foam::myStochasticModel<CloudType>::update
(const scalar dt,
const label celli,
const vector& U,
const vector& Uc,
vector& UTurb,
scalar& tTurb)
{
    cachedRandom& rnd = this->owner().rndGen();
    const scalar cps = 0.16432;

    const scalar k = this->kPtr_->primitiveField()[celli];
    const scalar epsilon =
        this->epsilonPtr_->primitiveField()[celli] + ROOTVSMALL;

    const scalar UrelMag = mag(U - Uc - UTurb);

    const scalar tTurbLoc =
        min(k/epsilon, cps*pow(k, 1.5)/epsilon/(UrelMag + SMALL));
// Parcel is perturbed by the turbulence
if (dt < tTurbLoc)
{
    tTurb += dt;

    if (tTurb > tTurbLoc)
    {
        tTurb = 0;

    // Calculate yPlus value
    const fvPatchList& patches = this->owner().mesh().boundary();
    const objectRegistry& obr = this->owner().mesh();
    const word turbName = IOobject::groupName(turbulenceModel::propertiesName,
                                             this->owner().U().group());
    const turbulenceModel& turbModel = obr.lookupObject<turbulenceModel>(turbName);

    forAll(patches, patchi)
    {
        // obtain the current Patch
        const fvPatch& currPatch = patches[patchi];

        // calculation
        if (typeid(currPatch) == typeid(wallFvPatch))
        {
            const scalarField& y_ = turbModel.y()[patchi];
            const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
            const tmp<scalarField> tnuw = turbModel.nu(patchi);
            const scalarField& nuw = tnuw();

            // yPlus for the wallFvPatch
            fvPatchField<scalar>& yPpatch = const_cast<fvPatchField<scalar>&>(y_.boundaryField()[patchi]);
            yPpatch=(y_*sqrt(nuw*mag(Uw.snGrad()))/nuw);

            if (yPpatch[patchi] < 100.0)
            {
                scalar f_x;
                scalar f_y;
                scalar f_z;
3.2. CREATE NEW STOCHASTIC MODEL – MYSTOCHASTICMODEL

Chapter 3. Modification

```plaintext
scalar dir_x;
scalar dir_y;
scalar dir_z;

const scalar sigma = sqrt(2*k/3.0);
const scalar theta = rnd.sample01<scalar>()*twoPi;
const scalar u = 2*rnd.sample01<scalar>() - 1;
const scalar a = sqrt(1 - sqr(u));

f_x = 1 + 0.285*(yPpatch[patchi] + 6)*exp(-0.455*pow(yPpatch[patchi] + 6, 0.53));
f_y = 1 - exp(-0.02*yPpatch[patchi]);
f_z = sqrt(3 - sqr(f_x) - sqr(f_y));

// update the turbulence fluctuating velocity for particle

dir_x = f_x*a*cos(theta);
dir_y = f_y*a*sin(theta);
dir_z = f_z*u;

const vector dir(dir_x, dir_y, dir_z);
UTurb = sigma*dir;
}
else
{

const scalar sigma = sqrt(2*k/3.0);

// Calculate a random direction dir distributed uniformly
// in spherical coordinates

const scalar theta = rnd.sample01<scalar>()*twoPi;
const scalar u = 2*rnd.sample01<scalar>() - 1;
const scalar a = sqrt(1 - sqr(u));
const vector dir(a*cos(theta), a*sin(theta), u);
UTurb = sigma*mag(rnd.GaussNormal<scalar>())*dir;
}
}
else
{
tTurb = GREAT;
UTurb = Zero;
}
```
After finished this modification, some correction should be accomplished before compilation.

### 3.3 Compile new class myStochasticModel

Go to the Make file, assume you are in `$WM_PROJECT_USER_DIR/turbulence/submodels/Kinematic/DispersionModel/myStochasticModel`

```bash
cd ../../../..
vim Make/files
```

The compiler requires a list of .C files that must be compiled, therefore the `Make/files` should contain the main .C file. In this case, the source files are some make-files in which specify the source files need to be compile, as the name end with .C, you can see below:

```
PARCELS=parcels
DERIVEDPARCELS=$(PARCELS)/derived
KINEMATICPARCEL=$(DERIVEDPARCELS)/basicKinematicParcel
$(KINEMATICPARCEL)/makeBasicKinematicParcelSubmodels.C
LIB = $(FOAM_USER_LIBBIN)/liblagrangianTurbulence
```

To compile the new class `myStochasticModel`, the file `makeBasicKinematicParcelSubmodels.C` need to be investigated:

```bash
cd ..
find -name "makeBasicKinematicParcelSubmodels.C"
vim ./parcels/derived/basicKinematicParcel/makeBasicKinematicParcelSubmodels.C
```

```
makeBasicKinematicParcelSubmodels.

---

```C
#include "basicKinematicCloud.H"
#include "makeParcelTurbulenceDispersionModels.H"

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

namespace Foam {

makeParcelTurbulenceDispersionModels(basicKinematicCloud);

} // namespace Foam

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

The function of the header file `makeParcelTurbulenceDispersionModels.H` is to include the options under sub-models for `DispersionModel`, refer to section 4.3.4.
Then, modify the file `makeParcelTurbulenceDispersionModels.H`:

```
vim ./parcels/include/makeParcelTurbulenceDispersionModels.H
```

Old `makeParcelTurbulenceDispersionModels.H`

```
 ifndef makeParcelTurbulenceDispersionModels_h
 define makeParcelTurbulenceDispersionModels_h

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

 #include "GradientDispersionRAS.H"
 #include "StochasticDispersionRAS.H"

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

 define makeParcelTurbulenceDispersionModels(CloudType)
     typedef Foam::CloudType::kinematicCloudType kinematicCloudType;
     defineNamedTemplateTypeNameAndDebug
       ( Foam::DispersionRASModel<kinematicCloudType>, 0 );
     makeDispersionModelType(GradientDispersionRAS, CloudType);
     makeDispersionModelType(StochasticDispersionRAS, CloudType);

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

Modified to:

```
ifndef makeParcelTurbulenceDispersionModels_h
#define makeParcelTurbulenceDispersionModels_h

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

#include "GradientDispersionRAS.H"
#include "StochasticDispersionRAS.H"
#include "myStochasticModel.H"

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

define makeParcelTurbulenceDispersionModels(CloudType)
    typedef Foam::CloudType::kinematicCloudType kinematicCloudType;
    defineNamedTemplateTypeNameAndDebug
      ( Foam::DispersionRASModel<kinematicCloudType>, 0 );
    makeDispersionModelType(GradientDispersionRAS, CloudType);
    makeDispersionModelType(StochasticDispersionRAS, CloudType);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```
3.3. Compile New Class MystochasticModel

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

Until now, the modification of the class `myStochasticModel` is done. Go to compile to add the new model as a submodel of `DispersionModel`:

```
wclean
wmake
```
Chapter 4

Results from Test Case

This section shows how to use and modify the files for icoUncoupledKinematicNocollisionFoam with myStochasticModel in case setup.

4.1 Pre-requirements

Before you start this test case, ensure that you have installed the ccm26ToFoam, if you have not installed, then go visit the following website and install the application correctly corresponding to your OpenFOAM version:

https://github.com/wyldckat/localCCM26ToFOAM

4.2 Primitive preparation

Before implementing the new solver with the new model, a steady flow with $k$ and $\varepsilon$ values should be obtained.

4.2.1 Basic case

As explained in section 2.3, the $k-\varepsilon$ model is essential for myStochasticModel as turbulent kinetic energy $k$ and dissipation $\varepsilon$ will be used to calculate the fluctuating velocity $UTub$. As mentioned in the former section, RAS $k-\varepsilon$ with simpleFoam is utilised to obtain a steady flow, as well ask and $\varepsilon$ values. Go to the OpenFOAM tutorials directory and copy the original case cavity to the run directory (the case is under pisoFoam, but it is fine to use simpleFoam with some modifications).

```
tut
cp -r incompressible/pisoFoam/ras/cavity $FOAM_RUN/myPipe
run
cd myPipe
/* remove the blockMeshDict, since the mesh model import from another source*/
rm myPipe/system/blockMeshDict
```

4.2.2 Vertical pipe

The vertical pipe model is created by Star-CCM+ and ccm26ToFoam is used to convert into OpenFOAM. The parameters and the configuration as following:
4.2. PRIMITIVE PREPARATION

Dimensions:
Radius: 0.01m
Length: 1.5m

Mesh stats
points: 7469
faces: 20760
internal faces: 18456
cells: 6688
faces per cell: 5.86364
boundary patches: 3
point zones: 0
face zones: 0
cell zones: 0

Patch Faces Points
shroud 2128 2156
outlet 88 97
inlet 88 97

Convert into OpenFoam (you should install ccm26ToFoam before the operation):

```bash
cp /*the path where you download the mesh.ccm*/ WM_PROJECT_USER_DIR/myPipe
ccm26ToFoam mesh.ccm
transformPoints -scale (0.1 0.1 1)
```

Since the inlet should be the upper-most patch, but the mesh sets the lower-most patch as inlet, you should exchange the patch names manually in the boundary file:

```bash
vim constant/ployMesh/boundary
```

```bash
3
(
shroud
{
type wall;
inGroups 1(wall);
nFaces 2128;
startFace 18456;
}
outlet
{
type patch;
nFaces 88;
startFace 20584;
}
inlet
{
type patch;
nFaces 88;
startFace 20672;
}
```
4.2.3 Modification the initial parameters in the 0 directory

As the basic case is different, the initial parameters in the 0 directory must be modified, the modifications of 0/p, 0/U, 0/k, 0/epsilon are posted here:

0/p:
```plaintext
dimensions [0 2 -2 0 0 0 0];
internalField uniform 0;
boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    outlet
    {
        type fixedValue;
        value uniform 0;
    }
    shroud
    {
        type zeroGradient;
    }
}
```

0/U:
```plaintext
dimensions [0 1 -1 0 0 0 0];
internalField uniform (0 0 0);
boundaryField
{
    inlet
    {
        type fixedValue;
        value uniform (0 0 -12);
    }
    outlet
    {
        type zeroGradient;
    }
    shroud
    {
        type fixedValue;
        value uniform (0 0 0);
    }
}
```

0/k:
```plaintext
dimensions [0 2 -2 0 0 0 0];
internalField uniform 0.0006;
boundaryField
{
    inlet
    {
        type fixedValue;
        value uniform 0.0006;
    }
    outlet
    {
        type zeroGradient;
    }
    shroud
    {
        type kqRWallFunction;
        value uniform 0.00375;
    }
}
```

0/epsilon:
```plaintext
dimensions [0 2 -3 0 0 0 0];
internalField uniform 0.00754;
boundaryField
{
    inlet
    {
        type zeroGradient;
    }
    outlet
    {
        type zeroGradient;
    }
    shroud
    {
        type epsilonWallFunction;
        value uniform 0.00754;
    }
}
```
4.2.4 Modification in the constant directory

To set similar conditions at those used by A. Dehbi, the transportProperties should be changed for the same Re number.

For flow in a pipe or tube, the Reynolds number is generally defined as:

\[ Re = \frac{Q D_H}{\nu A} = \frac{v D}{\nu} \approx 12000 \]  

\[ \rho_{\text{Inf}} = [1 -3 0 0 0 0] 1.2; \]
\[ \text{transportModel Newtonian;} \]
\[ \nu = [0 2 -1 0 0 0] \times 10^{-5}; \]

4.2.5 Modification in the system directory

The controlDict under system directory also need to be modified as:

```
application simpleFoam;
startFrom startTime;
startTime 0;
stopAt endTime;
endTime 600;
deltaT 1;
writeControl timeStep;
writeInterval 50;
purgeWrite 0;
writeFormat ascii;
writePrecision 6;
writeCompression off;
timeFormat general;
timePrecision 6;
runTimeModifiable true;
```

And fvSchemes:

```
ddtSchemes
{
    default steadyState;
}
gradSchemes
{
    default Gauss linear;
}
divSchemes
{
    default none;
div(\phi,\mathbf{U}) bounded Gauss linearUpwind grad(\mathbf{U});
div(\phi,\mathbf{k}) bounded Gauss upwind;
div(\phi,\epsilon) bounded Gauss upwind;
div(\phi,\nu_{\text{Tilda}}) Gauss limitedLinear 1;
div((\nu_{\text{Eff}} \cdot \text{dev2(T(grad(\mathbf{U})))))} Gauss linear;
```
4.2. PRIMITIVE PREPARATION

CHAPTER 4. RESULTS FROM TEST CASE

```plaintext
laplacianSchemes
{
    default Gauss linear corrected;
}
interpolationSchemes
{
    default linear;
}
snGradSchemes
{
    default corrected;
}

As well as fvSolution:

solvers
{
    p
    {
        solver GAMG;
        tolerance 1e-06;
        relTol 0;
        smoother GaussSeidel;
    }

    pFinal
    {
        $p;
        tolerance 1e-06;
        relTol 0;
    }

    "(U|k|epsilon|omega|R|nuTilda)"
    {
        solver smoothSolver;
        smoother GaussSeidel;
        tolerance 1e-06;
        relTol 0;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 0;
    pRefCell 0;
    pRefValue 0;
}

SIMPLE
{
    consistant yes;
}
```
4.3 TEST ICOUNCOUPLEDKINEMATICNOCOLLISIONFOAM WITH MYSTOCHASTICMODEL

CHAPTER 4. RESULTS FROM TEST CASE

relaxationFactors
{
equations
{
  U 0.5;
p 0.3;
  epsilon 0.3;
  k 0.3;
}
}

4.2.6 Steady flow

The modifications are done now and run the simulation:

```bash
simpleFoam > & log&
```

The steady flow shows below, the velocity $U$ is steady in the last time directory, around 14.8m/s for the entire inner field.

![Figure 4.1: Steady Flow](image)

4.3 Test icoUncoupledKinematicNocollisionFoam with myStochasticModel

4.3.1 Setup the test case

After the `simpleFoam` simulation, there are a few more time directories. Delete the time directories except the last one, the 600 directory, and then rename directory 600 to directory 0:

```bash
rm -r [1-5]*
mv 600 0
```

And the files in directory 0 also need to be modified, correct "600" to "0" in each file.
Since the test case uses `icoUncoupledKinematicNocollisonFoam`, the setting files under the test case of the original solver (`icoUncoupledKinematicParcelFoam`) could be adopted.

Go to the OpenFOAM tutorials directory and copy the original case `hopperInitialState` (under `lagrangian/icoUncoupledKinematicParcelFoam`) to the run directory named as `caseStudy`.

```bash
tut
cp -r lagrangian/icoUncoupledKinematicParcelFoam/hopper/hopperInitialState $FOAM_RUN/caseStudy
```

Copy the files under `constant` (not `transportProperties` and `turbulenceProperties`) and `system` (not `blockMeshDict`) to `myPipe`

```bash
run
cd myPipe
cp ../caseStudy/constant/kinematicCloudPositions constant
cp ../caseStudy/constant/kinematicCloudProperties constant
cp ../caseStudy/constant/g constant
cp ../caseStudy/system/controlDict system
cp ../caseStudy/system/fvSchemes system
cp ../caseStudy/system/fvSolution system
```

### 4.3.2 Modification of controlDict

The `controlDict` contributes to the simulation procedure, set as:

```plaintext
application icoUncoupledKinematicNocollisionFoam;
startFrom startTime;
startTime 0;
stopAt endTime;
endTime 0.3;
deltaT 1e-5;
writeControl timeStep;
writeInterval 100;
purgeWrite 0;
writeFormat ascii;
writePrecision 6;
writeCompression uncompressed;
timeFormat general;
timePrecision 6;
```
4.3.3 Modification of kinematicCloudPosition

The kinematicCloudPosition file contributes to the injected particle position. Generally, the number of particle should be around 5000 - 10000 to be regarded as sufficient. However, this case is set up just to test the new solver and model and only 74 particle positions are created, refer to appendix 6.7.

4.3.4 Modification of kinematicCloudProperties

Modify the subModels dictionary and value of fixedValueDistribution in the file kinematicCloudProperties. The diameter of the particle is set as 10µm for a better estimation of the new stochastic model and the dispersionModel under subModels is myStochasticModel. As a control, the other test case will set dispersionModel as StochasticDispersionRAS for comparison.

```plaintext
subModels
{
    particleForces
    {
        sphereDrag;
        gravity;
    }

    injectionModels
    {
        model1
        {
            type manualInjection;
            massTotal 0;
            parcelBasisType fixed;
            nParticle 1;
            SOI 0;
            positionsFile "kinematicCloudPositions";
            U0 (0 0 0);
            sizeDistribution
            {
                type fixedValue;
                fixedValueDistribution
                {
                    value 10e-6;
                }
            }
        }
    }

    dispersionModel myStochasticModel;
    patchInteractionModel none;
    surfaceFilmModel none;
    stochasticCollisionModel none;
    collisionModel none;
    pairCollisionCoeffs
```
4.3. TEST ICOUNCOUPLEDKINEMATICNOCOLLISIONFOAM WITH MYSTOCHASTICMODEL

CHAPTER 4. RESULTS FROM TEST CASE

```plaintext
{
    // Maximum possible particle diameter expected at any time
    maxInteractionDistance 1e-6;

    writeReferredParticleCloud no;

    pairModel pairSpringSliderDashpot;

    pairSpringSliderDashpotCoeffs
    {
        useEquivalentSize no;
        alpha 0.12;
        b 1.5;
        mu 0.52;
        cohesionEnergyDensity 0;
        collisionResolutionSteps 12;
    };

    wallModel wallLocalSpringSliderDashpot;

    wallLocalSpringSliderDashpotCoeffs
    {
        useEquivalentSize no;
        collisionResolutionSteps 12;
        inlet
        {
            youngsModulus 1e10;
            poissonsRatio 0.23;
            alpha 0.12;
            b 1.5;
            mu 0.43;
            cohesionEnergyDensity 0;
        }

        outlet
        {
            youngsModulus 1e10;
            poissonsRatio 0.23;
            alpha 0.12;
            b 1.5;
            mu 0.1;
            cohesionEnergyDensity 0;
        }

        shroud
        {
            youngsModulus 1e10;
            poissonsRatio 0.23;
            alpha 0.12;
            b 1.5;
            mu 0.43;
            cohesionEnergyDensity 0;
        }
    }

};
```
4.4. SIMULATION RESULT

4.3.5 Run simulation

Now, the setup has been done. Run the simulation with `icoUncoupledKinematicNocollisionFoam`:

```
icoUncoupledKinematicNocollisionFoam >& log&
```

If you want to check the log file during simulating, via:

```
less log
```

4.4 Simulation result

After finished the simulation, you can use `paraview` to check the particle tracking:

```
// create the foam file for paraview
touch myPipe.foam
// 0 time directory should be non-read for the particle tracking in paraview
mv 0 0_Orig
paraview
```

The particles in transition, from top view:

![Image of particles in transition](image)

Figure 4.2: The particles in transition, top view

4.4.1 Comparison between myStochasticModel and StochasticDispersion-RAS

The experimental studies of the deposition of particles in turbulent vertical pipe flows have been done by a number of researches. The well know experiments by Liu and Agarwal could be used as contrast results for the predictions, but the comparison is beyond the current tutorial. Only a brief description of the results is presented.

According to these two pairs comparison, the new model yields a small fluctuating velocity in turbulence model, especially in the y-direction, 0.231 vs 1.4 (max UTub-Y).
There is a problem of statistics of the particle deposit positions since the particle just deposits in one position transiently and will be transported by the flow finally. According to the time limitation, the author has not found a efficient way to figure out this problem. It will be one part of future work.

The accuracy of the position of the particles will not be discussed at this stage as it is beyond the scope of this tutorial.

The entire simulation indicates that the modification works and the new stochastic model could be implemented for the new solver, but the accuracy of the new model need to test in the future, since the simulation result has not been compared with the experimental result.

![Figure 4.3: myStochasticModel UTurb, top view](image1)

![Figure 4.4: RAS UTurb, top view](image2)
4.4. SIMULATION RESULT

CHAPTER 4. RESULTS FROM TEST CASE

Figure 4.5: myStochasticModel UTurb y-dir, top view

Figure 4.6: RAS UTurb y-dir, top view
Chapter 5

Conclusions and Further Work

5.1 Conclusion

In this tutorial the reader should have seen some possibilities to do the entire process chain in CFD with OpenFOAM, including the modifications in the both source file and the solver. The efficient process to create a new solver or new model is:

1. modifying on a similar solver, model or class;
2. adding some particular function according to the users’ requirements;
3. compiling the modified document as following the default path or similar one;
4. testing it in a case to ensure it works.

A simple geometry was converted from Star-CCM+ to OpenFOAM, showing that OpenFOAM is flexible and could work together with another software.

5.2 Further work

The accuracy of the new model should be tested with sufficient particles and fine meshed model in the future, and the results should be compared with the experimental data.
Study Questions

1. What is the main difference between the new stochastic model and the original one? Which part has been improved?

2. How to implement the class `basicKinematicCloud` into solver instead of the class `basicKinematicCollidingCloud`?

3. How does the original stochastic model implement the isotropic assumption for the turbulent fluctuating velocity?

4. How to call the class `turbulentModel` in the class `myStochasticModel`?

5. Where find the make-compile file for the dispersion stochastic model?

6. Why you setup a steady flow for calculating the particle velocity in this test case?

7. Which command to modify the mesh size?

8. How to switch the patch settings of inlet or outlet in `polyMesh`?
Bibliography


Chapter 6

Appendix

6.1 KinematicCloud.C

```c++
// **************************************** solve member function **************************************** //

template<class CloudType>
template<class TrackData>
void Foam::KinematicCloud<CloudType>::solve(TrackData& td) {
    if (solution_.steadyState()) {
        td.cloud().storeState();
        td.cloud().preEvolve();
        evolveCloud(td);
    }
    else {
        td.cloud().preEvolve();
        evolveCloud(td);
        if (solution_.coupled()) {
            td.cloud().scaleSources();
        }
    }
    td.cloud().info();
    td.cloud().postEvolve();
    if (solution_.steadyState())
```
```cpp
123  {
124      td.cloud().restoreState();
125  }
126

// * * * * * * * * * * * evolveCloud member function * * * * * * * * * * * //

174 template<class CloudType>
175 template<class TrackData>
176 void Foam::KinematicCloud<CloudType>::evolveCloud(TrackData& td)
177 {
178    if (solution_.coupled())
179    {
180      td.cloud().resetSourceTerms();
181    }
182
183    if (solution_.transient())
184    {
185      label preInjectionSize = this->size();
186      this->surfaceFilm().inject(td);
187      // Update the cellOccupancy if the size of the cloud has changed
188      // during the injection.
189      if (preInjectionSize != this->size())
190      {
191        updateCellOccupancy();
192        preInjectionSize = this->size();
193      }
194
195      injectors_.inject(td);
196
197      // Assume that motion will update the cellOccupancy as necessary
198      // before it is required.
199      td.cloud().motion(td);
200      stochasticCollision().update(solution_.trackTime());
201    }
202  else
203    {
204      // this->surfaceFilm().injectSteadyState(td);
205      injectors_.injectSteadyState(td, solution_.trackTime());
206      td.part() = TrackData::tpLinearTrack;
207      CloudType::move(td, solution_.trackTime());
208    }
209

// * * * * * * * * * * * evolve member function * * * * * * * * * * * //

675 template<class CloudType>
676 void Foam::KinematicCloud<CloudType>::evolve()
```
677 {
678     if (solution_.canEvolve())
679     {
680         typename parcelType::template
681             TrackingData<KinematicCloud<CloudType>> td(*this);
682
683         solve(td);
684     }
685 }

template<class ParticleType>
template<class TrackData>
void Foam::Cloud<ParticleType>::move(TrackData& td, const scalar trackTime)
{
    const polyBoundaryMesh& pbm = pMesh().boundaryMesh();
    const globalMeshData& pData = polyMesh_.globalData();

    // Which patches are processor patches
    const labelList& procPatches = pData.processorPatches();

    // Indexing of patches into the procPatches list
    const labelList& procPatchIndices = pData.processorPatchIndices();

    // Indexing of equivalent patch on neighbour processor into the
    // procPatches list on the neighbour
    const labelList& procPatchNeighbours = pData.processorPatchNeighbours();

    // Which processors this processor is connected to
    const labelList& neighbourProcs = pData[Pstream::myProcNo()];

    // Indexing from the processor number into the neighbourPocs list
    labelList neighbourProcIndices(Pstream::nProcs(), -1);
    forAll(neighbourProcs, i)
    {
        neighbourProcIndices[neighbourProcs[i]] = i;
    }

    // Initialise the stepFraction moved for the particles
    forAllIter(typename Cloud<ParticleType>, *this, pIter)
    {
        pIter().stepFraction() = 0;
    }

    // Reset nTrackingRescues
    nTrackingRescues_ = 0;

    // List of lists of particles to be transfered for all of the
    // neighbour processors
    List<IDLList<ParticleType>> particleTransferLists
    ( neighbourProcs.size() );

    // List of destination processorPatches indices for all of the
    // neighbour processors
    List<DynamicList<label>> patchIndexTransferLists
    ( neighbourProcs.size() );
// Allocate transfer buffers
PstreamBuffers pBufs(Pstream::nonBlocking);

// While there are particles to transfer
while (true) {
    particleTransferLists = IDLList<ParticleType>();
    forAll(patchIndexTransferLists, i) {
        patchIndexTransferLists[i].clear();
    }

    // Loop over all particles
    forAllIter(typename Cloud<ParticleType>, *this, pIter) {
        ParticleType& p = pIter();

        // Move the particle
        bool keepParticle = p.move(td, trackTime);

        // If the particle is to be kept
        // (i.e. it has passed through an inlet or outlet)
        if (keepParticle) {
            // If we are running in parallel and the particle is on a
            // boundary face
            if (Pstream::parRun() & p.face() >= pMesh().nInternalFaces()) {
                label patchi = pbm.whichPatch(p.face());

                // ... and the face is on a processor patch
                // prepare it for transfer
                if (procPatchIndices[patchi] != -1) {
                    label n = neighbourProcIndices[patchi];
                    refCast<const processorPolyPatch> (pbm[patchi].neighbProcNo());
                    p.prepareForParallelTransfer(patchi, td);
                    particleTransferLists[n].append(this->remove(&p));
                    patchIndexTransferLists[n].append (procPatchNeighbours[patchi]);
                }
            }
        }
    }
}


6.2. CLOUD.C

CHAPTER 6. APPENDIX

290     }
291     else
292     {
293         deleteParticle(p);
294     }
295     }
296
297     if (!Pstream::parRun())
298     {
299         break;
300     }
301
302
303     // Clear transfer buffers
304     pBuFs.clear();
305
306     // Stream into send buffers
307     forAll(particleTransferLists, i)
308     {
309         if (particleTransferLists[i].size())
310             {
311                 UOPstream particleStream
312                     (neighbourProcs[i],
313                     pBuFs
314                     );
315
316                     particleStream
317                     <<= patchIndexTransferLists[i]
318                     << particleTransferLists[i];
319             }
320     }
321     }
322
323     // Start sending. Sets number of bytes transferred
324     labelList allNTrans(Pstream::nProcs());
325     pBuFs.finishedSends(allNTrans);
326
327
328     bool transferred = false;
329
330     forAll(allNTrans, i)
331     {
332         if (allNTrans[i])
333             {
334                 transferred = true;
335                 break;
336             }
337     }
338     reduce(transferred, orOp<bool>());
339
340     if (!transferred)
341     {
342         break;
343     }
// Retrieve from receive buffers
forAll(neighbourProcs, i)
{
    label neighbProci = neighbourProcs[i];
    label nRec = allNTrans[neighbProci];
    if (nRec)
    {
        UIPstream particleStream(neighbProci, pBuFS);
        labelList receivePatchIndex(particleStream);
        IDLList<ParticleType> newParticles
        {
            particleStream,
            typename ParticleType::iNew(polyMesh_)
        };
        label pI = 0;
        forAllIter(typename Cloud<ParticleType>, newParticles, newpIter)
        {
            ParticleType& newp = newpIter();
            label patchi = procPatches[receivePatchIndex[pI++]];
            newp.correctAfterParallelTransfer(patchi, td);
            addParticle(newParticles.remove(&newp));
        }
    }  
}
if (cloud::debug)
{
    reduce(nTrackingRescues_, sumOp<label>()) ;
    if (nTrackingRescues_ > 0)
    {
        Info<< nTrackingRescues_ << " tracking rescue corrections" << endl;
    }
}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //

template<class ParcelType>
template<class TrackData>
bool Foam::KinematicParcel<ParcelType>::move
(
    TrackData& td,
    const scalar trackTime
)
{
    typename TrackData::cloudType::parcelType& p =
        static_cast<typename TrackData::cloudType::parcelType&>(*this);

td.switchProcessor = false;
    td.keepParticle = true;

    const polyMesh& mesh = td.cloud().pMesh();
    const polyBoundaryMesh& pbMesh = mesh.boundaryMesh();
    const scalarField& cellLengthScale = td.cloud().cellLengthScale();
    const scalar maxCo = td.cloud().solution().maxCo();

    scalar tEnd = (1.0 - p.stepFraction())*trackTime;
    scalar dtMax = trackTime;
    if (td.cloud().solution().transient())
    {
        dtMax *= maxCo;
    }

    bool tracking = true;
    label nTrackingStalled = 0;

    while (td.keepParticle && !td.switchProcessor && tEnd > ROOTVSMALL)
    {
        // Apply correction to position for reduced-D cases
        meshTools::constrainToMeshCentre(mesh, p.position());

        const point start(p.position());

        // Set the Lagrangian time-step
        scalar dt = min(dtMax, tEnd);

        // Cache the parcel current cell as this will change if a face is hit
        const label celli = p.cell();

        const scalar magU = mag(U_);
        if (p.active() && tracking && (magU > ROOTVSMALL))
        {
            const scalar d = dt*magU;
            const scalar dCorr = min(d, maxCo*cellLengthScale[celli]);
            dt *=
                dCorr/d
*p.trackToFace(p.position() + dCorr*U_/magU, td);
}

tEnd -= dt;

scalar newStepFraction = 1.0 - tEnd/trackTime;

if (tracking)
{
  if
  (mag(p.stepFraction() - newStepFraction)
   < particle::minStepFractionTol
  )
  {
    nTrackingStalled++;

    if (nTrackingStalled > maxTrackAttempts)
    {
      tracking = false;
    }
  }
  else
  {
    nTrackingStalled = 0;
  }
}

p.stepFraction() = newStepFraction;

bool calcParcel = true;
if (!tracking && td.cloud().solution().steadyState())
{
  calcParcel = false;
}

// Avoid problems with extremely small timesteps
if ((dt > ROOTVSMALL) && calcParcel)
{
  // Update cell based properties
  p.setCellValues(td, dt, celli);

  if (td.cloud().solution().cellValueSourceCorrection())
  {
    p.cellValueSourceCorrection(td, dt, celli);
  }

  p.calc(td, dt, celli);
}

if (p.onBoundary() && td.keepParticle)
{
  if (isA<processorPolyPatch>(pbMesh[p.patch(p.face())]))
  {
    
  }
}
td.switchProcessor = true;
}
    }
    p.age() += dt;
    td.cloud().functions().postMove(p, celli, dt, start, td.keepParticle);
}
return td.keepParticle;
}

template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::hitFace(TrackData& td)
{
    typename TrackData::cloudType::parcelType& p =
        static_cast<typename TrackData::cloudType::parcelType&>(*this);
    td.cloud().functions().postFace(p, p.face(), td.keepParticle);
}

template<class ParcelType>
void Foam::KinematicParcel<ParcelType>::hitFace(int& td)
{}

template<class ParcelType>
template<class TrackData>
bool Foam::KinematicParcel<ParcelType>::hitPatch(
    const polyPatch& pp,
    TrackData& td,
    const label patchi,
    const scalar trackFraction,
    const tetIndices& tetIs
)
{
    typename TrackData::cloudType::parcelType& p =
        static_cast<typename TrackData::cloudType::parcelType&>(*this);
    // Invoke post-processing model
    td.cloud().functions().postPatch(
        p,
        pp,
        trackFraction,
        tetIs,
        td.keepParticle
    );
    // Invoke surface film model

if (td.cloud().surfaceFilm().transferParcel(p, pp, td.keepParticle))
{
    // All interactions done
    return true;
}
else
{
    // Invoke patch interaction model
    return td.cloud().patchInteraction().correct
    (p,
     pp,
     td.keepParticle,
     trackFraction,
     tetIs
    );
}

template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::hitProcessorPatch

    (const processorPolyPatch&,
     TrackData& td
    )
{
    td.switchProcessor = true;
}

template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::hitWallPatch

    (const wallPolyPatch& wpp,
     TrackData& td,
     const tetIndices&
    )
{
    // Wall interactions handled by generic hitPatch function
}

template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::hitPatch

    (const polyPatch&,
     TrackData& td
    )
{
    td.keepParticle = false;
template<class ParcelType>
void Foam::KinematicParcel<ParcelType>::transformProperties(const tensor& T)
{
    ParcelType::transformProperties(T);
    U_ = transform(T, U_);
}

template<class ParcelType>
void Foam::KinematicParcel<ParcelType>::transformProperties(const vector& separation)
{
    ParcelType::transformProperties(separation);
}

template<class ParcelType>
Foam::scalar Foam::KinematicParcel<ParcelType>::wallImpactDistance(const vector& ) const
{
    return 0.5*d_;
}
#ifndef myStochasticModel_H
#define myStochasticModel_H
#include "DispersionRASModel.H"

namespace Foam {

/*---------------------------------------------------------------------------*
 Class myStochasticModel Declaration
 *---------------------------------------------------------------------------*/

template<class CloudType>
class myStochasticModel :
    public DispersionRASModel<CloudType>
{
   public:

   // Runtime type information
   TypeName("myStochasticModel");

   // declaration of yPlus
   volScalarField yP_;  

   // Constructors
   
   // Construct from components
   myStochasticModel(const dictionary& dict, CloudType& owner);

   // Construct copy
   myStochasticModel(const myStochasticModel<CloudType>& dm);

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

   // Destructor  
   virtual "myStochasticModel();

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

};
}
#endif /* myStochasticModel_H */
// Member Functions

//-- Update (disperse particles)
virtual vector update
(
    const scalar dt,
    const label celli,
    const vector& U,
    const vector& Uc,
    vector& UTurb,
    scalar& tTurb
);

};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#ifdef NoRepository
    #include "myStochasticModel.C"
#endif

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif

// ***************************************************** //
6.5 myStochasticModel.C

```cpp
#include "myStochasticModel.H"
#include "constants.H"
#include "nearWallDist.C"

using namespace Foam::constant::mathematical;

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

template<class CloudType>
Foam::myStochasticModel<CloudType>::myStochasticModel
(const dictionary& dict,
 CloudType& owner)
:
  DispersionRASModel<CloudType>(dict, owner),
  yP_(
    IOobject
    ("yP",
     this->owner_.mesh().time().timeName(),
     this->owner_.mesh(),
     IOobject::NO_READ,
     IOobject::AUTO_WRITE
    ),
    this->owner_.mesh(),
    dimensionedScalar("yP", dimless, 0.0)
  )
{}

template<class CloudType>
Foam::myStochasticModel<CloudType>::myStochasticModel
(const myStochasticModel<CloudType>& dm)
:
  DispersionRASModel<CloudType>(dm),
  yP_(dm.yP_)
{}

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

template<class CloudType>
```
Foam::myStochasticModel<CloudType>::myStochasticModel()
{

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

    template<class CloudType>
    Foam::vector Foam::myStochasticModel<CloudType>::update
    (  
        const scalar dt,
        const label celli,
        const vector& U,
        const vector& Uc,
        vector& UTurb,
        scalar& tTurb
    )
    {
        cachedRandom& rnd = this->owner().rndGen();
        
        const scalar cps = 0.16432;
        
        const scalar k = this->kPtr_->primitiveField()[celli];
        const scalar epsilon = this->epsilonPtr_->primitiveField()[celli] + ROOTVSMALL;
        const scalar UrelMag = mag(U - Uc - UTurb);
        const scalar tTurbLoc = min(k/epsilon, cps*pow(k, 1.5)/epsilon/(UrelMag + SMALL));
        
        // Parcel is perturbed by the turbulence
        if (dt < tTurbLoc)
        {
            tTurb += dt;
            if (tTurb > tTurbLoc)
            {
                tTurb = 0;
            }
            // Calculate yPlus value
            const fvPatchList& patches = this->owner().mesh().boundary();
            const objectRegistry& obr = this->owner().mesh();
            const word turbName = IOobject::groupName(turbulenceModel::propertiesName, this->owner().U().group());
            const turbulenceModel& turbModel = obr.lookupObject<turbulenceModel>(turbName);
        }
    }
forAll(patches, patchi)
{
    // obtain the current Patch
    const fvPatch& currPatch = patches[patchi];

    // calculation
    if (typeid(currPatch) == typeid(wallFvPatch))
    {
        const scalarField& y_ = turbModel.y()[patchi];
        const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
        const tmp<scalarField> tnuw = turbModel.nu(patchi);
        const scalarField& nuw = tnuw();

        // yPlus for the wallFvpatch
        fvPatchField<scalar>& yPpatch = const_cast<fvPatchField<scalar>&>(yP_.boundaryField()[patchi]);
        yPpatch = (y_*sqrt(nuw*mag(Uw.snGrad()))/nuw);

        if (yPpatch[patchi] < 100.0)
        {
            scalar f_x;
            scalar f_y;
            scalar f_z;
            scalar dir_x;
            scalar dir_y;
            scalar dir_z;

            scalar dir_x = f_x*a*cos(theta);
            scalar dir_y = f_y*a*sin(theta);
            scalar dir_z = f_z*u;
            const vector dir(dir_x, dir_y, dir_z);
        }
    }

    scalar f_x = 1 + 0.285*(yPpatch[patchi]+6)*exp(-0.455*pow(yPpatch[patchi]+6, 0.53));
    scalar f_y = 1 - exp(-0.02*yPpatch[patchi]);
    scalar f_z = sqrt(3-sqr(f_x)-sqr(f_y));

    // update the turbulence fluctuating velocity for particle
    dir_x = f_x*a*cos(theta);
    dir_y = f_y*a*sin(theta);
    dir_z = f_z*u;
    const vector dir(dir_x, dir_y, dir_z);
UTurb = sigma * dir;

} else {

    const scalar sigma = sqrt(2 * k / 3.0);

    // Calculate a random direction dir distributed uniformly
    // in spherical coordinates

    const scalar theta = rnd.sample01<scalar>() * twoPi;
    const scalar u = 2 * rnd.sample01<scalar>() - 1;

    const scalar a = sqrt(1 - sqr(u));
    const vector dir(a * cos(theta), a * sin(theta), u);

    UTurb = sigma * mag(rnd.GaussNormal<scalar>()) * dir;
}
else {
    tTurb = GREAT;
    UTurb = Zero;
}
return Uc + UTurb;
6.6 icoUncoupledKinematicNocollisionFoam.C

Just replaced "basicKinematicCollidingCloud.H" by "basicKinematicCloud.H".

```c++
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "basicKinematicCloud.H"

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

int main(int argc, char *argv[]) {
    argList::addOption
    (  
        "cloudName",
        "name",
        "specify alternative cloud name. default is 'kinematicCloud'"
    );

    #include "postProcess.H"
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"

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

    Info<< "Starting time loop
    Info<< "\nStarting time loop\n" << endl;
    while (runTime.loop()) {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        Info<< "Evolving " << kinematicCloud.name() << endl;
        laminarTransport.correct();
        mu = laminarTransport.nu()*rhoInfValue;
        kinematicCloud.evolve();
        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << " ClockTime = " << runTime.elapsedClockTime() << " s"
    }
}
```
79    << nl << endl;
80 }
81
82 Info<< "End\n" << endl;
83
84 return 0;
85 }
86
87 // ************************************************************** //
6.7 kinematicCloudPosition

The path of this file is Constant/kinematicCloudPosition, modified as:

```
FoamFile
{
    version 2.0;
    format ascii;
    class vectorField;
    object kinematicCloudPositions;
}
```

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

(0.002 0.00599 1.5)
(0.005 0.00305 1.5)
(0.004 0.00411 1.5)
(0.003 0.00317 1.5)
(0.004 0.00323 1.5)
(0.006 0.00329 1.5)
(0.001 0.00335 1.5)
(0.003 0.00341 1.5)
(0.007 0.00347 1.5)
(-0.009 0.00353 1.5)
(-0.008 0.00359 1.5)
(-0.007 0.00365 1.5)
(-0.006 0.00371 1.5)
(-0.005 0.00377 1.5)
(-0.004 0.00383 1.5)
(-0.003 0.00389 1.5)
(-0.002 0.00395 1.5)
(0.002 0.00401 1.5)
(0.001 0.00407 1.5)
(0.003 0.00413 1.5)
(0.001 0.00299 1.5)
(0.002 0.00305 1.5)
(0.007 0.00311 1.5)
(0.005 0.00323 1.5)
(0.004 0.00329 1.5)
(0.003 0.00335 1.5)
(0.002 0.00341 1.5)
(0.001 0.00347 1.5)
(-0.001 0.00353 1.5)
(-0.002 0.00359 1.5)
(-0.003 0.00365 1.5)
(-0.004 0.00371 1.5)
(-0.005 0.00377 1.5)
(-0.006 0.00383 1.5)
(-0.007 0.00389 1.5)
```

21
\(-0.008\ 0.00395\ 1.5\)
\(0.002\ -0.00599\ 1.5\)
\(0.005\ -0.00305\ 1.5\)
\(0.004\ -0.00411\ 1.5\)
\(0.003\ -0.00317\ 1.5\)
\(0.004\ -0.00323\ 1.5\)
\(0.006\ -0.00329\ 1.5\)
\(0.001\ -0.00335\ 1.5\)
\(0.003\ -0.00341\ 1.5\)
\(0.007\ -0.00347\ 1.5\)
\(-0.009\ -0.00353\ 1.5\)
\(-0.008\ -0.00359\ 1.5\)
\(-0.007\ -0.00365\ 1.5\)
\(-0.006\ -0.00371\ 1.5\)
\(-0.005\ -0.00377\ 1.5\)
\(-0.004\ -0.00383\ 1.5\)
\(-0.003\ -0.00389\ 1.5\)
\(-0.002\ -0.00395\ 1.5\)
\(0.002\ -0.00401\ 1.5\)
\(0.001\ -0.00407\ 1.5\)
\(0.003\ -0.00413\ 1.5\)
\(0.001\ -0.00299\ 1.5\)
\(0.002\ -0.00305\ 1.5\)
\(0.007\ -0.00311\ 1.5\)
\(0.006\ -0.00317\ 1.5\)
\(0.005\ -0.00323\ 1.5\)
\(0.004\ -0.00329\ 1.5\)
\(0.003\ -0.00335\ 1.5\)
\(0.002\ -0.00341\ 1.5\)
\(0.001\ -0.00347\ 1.5\)
\(-0.001\ -0.00353\ 1.5\)
\(-0.002\ -0.00359\ 1.5\)
\(-0.003\ -0.00365\ 1.5\)
\(-0.004\ -0.00371\ 1.5\)
\(-0.005\ -0.00377\ 1.5\)
\(-0.006\ -0.00383\ 1.5\)
\(-0.007\ -0.00389\ 1.5\)
\(-0.008\ -0.00395\ 1.5\)
6.8 kinematicCloudProperties

The path of this file is `Constant/kinematicCloudProperties`, modified as:

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

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

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

    integrationSchemes
    {
        U Euler;
    }
}

constantProperties
{
    rho0 964;
    youngsModulus 6e8;
    poissonsRatio 0.35;
}

subModels
{
    particleForces
    {
        sphereDrag;
        gravity;
    }
}
```
injectionModels {
  model1 {
    type manualInjection;
    massTotal 0;
    parcelBasisType fixed;
    nParticle 1;
    SOI 0;
    positionsFile "kinematicCloudPositions";
    U0 (0 0 0);
    sizeDistribution {
      type fixedValue;
      fixedValueDistribution {
        value 10e-6;
      }
    }
  }
}
dispersionModel myStochasticModel;
patchInteractionModel none;
surfaceFilmModel none;
stochasticCollisionModel none;
collisionModel pairCollision;
pairCollisionCoeffs {
  // Maximum possible particle diameter expected at any time
  maxInteractionDistance 10e-6;
  writeReferredParticleCloud no;
  pairModel pairSpringSliderDashpot;
pairSpringSliderDashpotCoeffs {
    useEquivalentSize no;
    alpha 0.12;
    b 1.5;
    mu 0.52;
    cohesionEnergyDensity 0;
    collisionResolutionSteps 12;
  }
  wallModel wallLocalSpringSliderDashpot;
wallLocalSpringSliderDashpotCoeffs
{
  useEquivalentSize no;
  collisionResolutionSteps 12;
  inlet
  {
    youngsModulus 1e10;
    poissonsRatio 0.23;
    alpha 0.12;
    b 1.5;
    mu 0.43;
    cohesionEnergyDensity 0;
  }
  outlet
  {
    youngsModulus 1e10;
    poissonsRatio 0.23;
    alpha 0.12;
    b 1.5;
    mu 0.43;
    cohesionEnergyDensity 0;
  }
  shroud
  {
    youngsModulus 1e10;
    poissonsRatio 0.23;
    alpha 0.12;
    b 1.5;
    mu 0.1;
    cohesionEnergyDensity 0;
  }
};
}

cloudFunctions
{}

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