Cite as: Zhang, J.: Modifying coalChemistryFoam for dense gas-solid simulation. In Proceedings of CFD with OpenSource Software, 2018, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2018

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Modifying coalChemistryFoam for dense gas-solid simulation

Developed for OpenFOAM-v1806
Requires: coalChemistryFoam

Author:
Jingyuan Zhang
Norwegian University of
Science and Technology
jingyuan.zhang@ntnu.no

Peer reviewed by:
Ebrahim Ghahramani
Marcus Jansson

Licensed under CC-BY-NC-SA, https://creativecommons.org/licenses/

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.

December 22, 2018

Learning outcomes

The reader will learn:

How to use it:

- How to use coalChemistryFoam

The theory of it:

- The theory of Eulerian-Lagrangian approach in modeling the thermal conversion of solid particles

How it is implemented:

- How to solve governing equations for the continuity phase coupled with dispersed phase

How to modify it:

- A new field variable will be created and solved by modified governing equations

Prerequisites

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

- Fundamentals of Eulerian-Lagrangian approach in simulating gas-solid system
- How to run standard document tutorials like coalChemistryFoam tutorial

Preface

In OpenFOAM-v1806 the Lagrangian solvers are `coalChemistryFoam`, `DPMFoam`, `simpleCoalParcelFoam`, `reactingParcelFoam`, `sprayFoam`, `icoUncoupledKinematicParcleFoam` and `UncoupledKinematicParcleFoam`. (solvers like `DPMDyMFoam` are not listed, because they are extensions of relevant solvers. For example, `DyM` means dynamic mesh in `DPMDyMFoam`). Not only the particle models for the Lagrangian tracking solid phase are different, but also the coupling methods between the continuous phase and the dispersed phase are distinguishable. The last two mentioned solvers, for compressible and incompressible fluid, are not two-way coupling and require a pre-calculated velocity field. The rest, except `DPMFoam`, are for compressible fluid. As for the coupling method, only `DPMFoam` considers the effects of the volume fraction of particles on the continuous phase. In addition, `simpleCoalParcel` is a steady state solver.

In the simulation of the thermal conversion of solid particles, the porous media model is widely used. It requires averaging methods when modeling the solid phase, and it has difficulties in simulating particles individual behaviours and interactions. In order to include more physics, the Eulerian-Lagrangian approach could be a better option. In the situation that the solid phase is quite dense, such as in grate fire furnace, we prefer to take both compressibility and effects of volume fraction of particles into account. In this respective, we need to modify the existing Lagrangian solvers in OpenFOAM.

In this tutorial, a new solver `coalChemistryAlphaFoam` will be developed based on `coalChemistryFoam`. The difference between the new solver and the original one is that the effects of the volume fraction of particles will be considered. The first section is theory and introduction, and then a detail understanding of the source code of `coalChemistryFoam` will be given. The third part is the implementation of the new solver, as well as a tutorial about how to modify the solvers. At last a sample test case will be conducted to show how to use the new solver.

Contents

Chapter 1

Theoretical Background of Eulerian-Lagrangian Approach

In Eulerian-Lagrangian approach, the continuous phase is treated as a continuum by solving the Navier-Stokes equations and the conservation equations (strictly speaking, Navier-Stokes equations are the momentum equation for viscous fluid, but in some papers we can also see that people use Navier-Stokes equations also including the mass and heat balance equations. Here we prefer the generalized definition.), and the dispersed phase is calculated by tracking of a large number of particles which follow the Newton's second law. In modeling granular flows, discrete element method (DEM) and Discrete Phase Model (DPM) are the models under Eulerian-Lagrangian framework. There is no strict distinction between DPM and DEM. DEM emphasize on the collision between individual particles, while DPM stresses the particles behaviour as a group.

When deriving the governing equations for the continuous phase in multi-phase flow systems, how to deal with the averaging method and coupling between the different phases will result in different formulations. According to the different ways of the decomposition of particle-fluid interaction force term in the governing equations for the continuous phase, the momentum equations are classified into models A and B [1].

The equation set of model A is

$$\frac{\partial \alpha \rho_c \mathbf{u}}{\partial t} + \nabla \cdot (\alpha \rho_c \mathbf{u}\mathbf{u}) = -\alpha \nabla p - \mathbf{F}^A + \alpha \nabla \cdot \boldsymbol{\tau} + \alpha \rho_c \mathbf{g} \tag{1.1}$$

where α is fluid volume phase fraction, ρ_c is fluid phase density, $\mathbf{g}$ is gravity, $\boldsymbol{\tau}$ is deviatoric stress tensor and $\mathbf{F}^A$ is the volumetric particle-fluid interaction force. In model A, the interaction force contains drag force $\mathbf{f}_{d,i}$ and the rest particle-fluid interaction forces, i stands for the individual particle label.

$$\mathbf{F}^A = \frac{1}{\Delta V} \sum_{i=1}^{n} (\mathbf{f}_{d,i} + \mathbf{f}_i^{''}) \tag{1.2}$$

The total particle-fluid interaction force on an individual particle $\mathbf{f}$ should include pressure gradient force $\mathbf{f}_{\nabla p,i}$ (which also includes buoyancy force) and viscous force $\mathbf{f}_{\nabla \cdot \tau,i}$, and can be written as

$$\mathbf{f}^A = \mathbf{f}_{d,i} + \mathbf{f}_{\nabla p,i} + \mathbf{f}_{\nabla \cdot \tau,i} + \mathbf{f}_i^{''} \tag{1.3}$$

Similar to model A,the equation set according to model B is

$$\frac{\partial \alpha \rho_c \mathbf{u}}{\partial t} + \nabla \cdot (\alpha \rho_c \mathbf{u}\mathbf{u}) = -\nabla p - \mathbf{F}^B + \nabla \cdot \boldsymbol{\tau} + \alpha \rho_c \mathbf{g} \tag{1.4}$$

$$\mathbf{F}^B = \frac{1}{\Delta V}\sum_{i=1}^{n}(\mathbf{f}_{d,i} + \mathbf{f}_{\nabla p,i} + \mathbf{f}_{\nabla\cdot\tau,i} + \mathbf{f}_i^{''}) \tag{1.5}$$

$$\mathbf{f}^B = \mathbf{f}_{d,i} + \mathbf{f}_{\nabla p,i} + \mathbf{f}_{\nabla\cdot\tau,i} + \mathbf{f}_i^{''} \tag{1.6}$$

Model B is most commonly used as another simplified equation set. The equation for the fluid is the same as in original model B, except that the particle-fluid interaction force terms are different. The simplified one is

$$\mathbf{F}^{B*} = \frac{1}{\alpha\Delta V}\sum_{i=1}^{n}(\mathbf{f}_{d,i} + \mathbf{f}_i^{''}) + \frac{1}{\Delta V}\sum_{i=1}^{n}(\rho_c V_{p,i}\mathbf{g}) \tag{1.7}$$

$$\mathbf{f}^{B*} = \frac{(\mathbf{f}_{d,i} + \mathbf{f}_i^{''})}{\alpha} - \rho_c V_{p,i}\mathbf{g} \tag{1.8}$$

where $V_{p,i}$ is the particle volume, and this simplification is conditional [1]. It is valid when

$$\alpha\rho_c[\frac{\partial\mathbf{u}}{\partial t} + \nabla\cdot(\mathbf{uu})] = 0 \tag{1.9}$$

For all of the equation sets above, the "rest particle-fluid interaction forces" usually includes virtual mass force $\mathbf{f}_{vm,i}$, Basset force $\mathbf{f}_{B,i}$ and lift forces such as the Saffman force $\mathbf{f}_{Saff,i}$ and Magnus force $\mathbf{f}_{Mag,i}$.

$$\mathbf{f}_i^{''} = \mathbf{f}_{vm,i} + \mathbf{f}_{B,i} + \mathbf{f}_{Saff,i} + \mathbf{f}_{Mag,i} \tag{1.10}$$

While in this work, we are focusing on the gas-solid flow, as compared with drag force these force can be neglect.

Although model A and B have different equation forms, both of them are correct and mathematically equivalent[2]. The equations here are only the momentum equations without source term. If there are mass transfer between fluid and particles, it will usually rise a momentum source or sink term for the momentum equations. In this tutorial case, during the conversion of the solid particle, product gas will be released which results a source term. Besides the momentum equation, the mass balance, heat transport, as well as the species transport equations need to be solved. If the turbulence models are used, the governing equations should also include turbulence transport equations. These equations will be given with the certain problems in the following chapters.

Chapter 2

coalChemistryFoam

In this chapter, the implementation of `coalChemistryFoam` in OpenFOAM will be described.

2.1 Governing equations for the gas phase

In `coalChemistryFoam` the influence of the solid volume fraction is not included. The gas phase is compressible fluid. The continuity equation in differential form is

$$\frac{\partial(\rho_g)}{\partial t} + \nabla \cdot (\rho_g \mathbf{U}_g) = S_m \tag{2.1}$$

The momentum equation is given by

$$\frac{\partial(\rho_g \mathbf{U}_g)}{\partial t} + \nabla \cdot (\rho_g \mathbf{U}_g \mathbf{U}_g) - \nabla \cdot (\tau_g) - \nabla \cdot (\rho_g \mathbf{R}_g) = -\nabla p + \rho_g \mathbf{g} + S_{\mathbf{U}} \tag{2.2}$$

The energy transport equation is

$$\frac{\partial(\rho_g h)}{\partial t} + \frac{\partial(\rho_g K)}{\partial t} + \nabla \cdot (\rho_g \mathbf{U}_g h) + \nabla \cdot (\rho_g \mathbf{U}_g K) - \nabla \cdot \big(\alpha_{eff} \nabla(h)\big) = \nabla p + \rho_g \mathbf{U}_g \cdot \mathbf{g} + S_h \tag{2.3}$$

Species transport equation is

$$\frac{\partial(\rho_g Y_i)}{\partial t} + \nabla \cdot (\rho_g \mathbf{U}_g Y_i) - \nabla \cdot \big(D_{eff} \nabla(\rho_g Y_i)\big) = S_i \tag{2.4}$$

S are the source terms, $\mathbf{R}_g$ is Ronald stress term, h is the enthalpy, K is kinetic energy, Y_i is the mass fraction of the species. α_{eff} and D_{eff} are the effective thermal and mass diffusivity, and "effective" means it include turbulent diffusivity when turbulence model is used.

2.2 Walkthrough of the source code

The `coalChemistryFoam` solver files are located in :

`$FOAM_SOLVERS/lagrangian/coalChemistryFoam`

We are going to have a deep looking into all the files. The walkthrough of the code is a tutorial on how the equations are implemented into code. The solver will start with the main file, which is `coalChemistryFoam.C`. All the other files in this folder are included in the main file. Then the equations files will be explained. We will try to find every terms from the equations in the code. Next is the `createFields.H`, at the same time, the mechanism of runtime selection (RTS) will be briefly introduced. The rest of the files in the directory will be introduced in the `coalChemistryFoam.C` and `createFields.H` section.

```
coalChemistryFoam
|
|---Make
|   |---files
|   |---options
|
|---coalChemistryFoam.C
|---createClouds.H
|---createFieldRefs.H
|---createFields.H
|---EEqn.H
|---pEqn.H
|---rhoEqn.H
|---setDeltaT.H
|---UEqn.H
|---YEqn.H
```

2.2.1 coalChemistryFoam.C

We all know that OpenFOAM is just a C++ code package, and this C file is where the code starts. First, before the main function, there is a block of included the header files.

```
#include "fvCFD.H"
#include "turbulentFluidThermoModel.H"
#include "basicThermoCloud.H"
#include "coalCloud.H"
#include "psiReactionThermo.H"
#include "CombustionModel.H"
#include "fvOptions.H"
#include "radiationModel.H"
#include "SLGThermo.H"
#include "pimpleControl.H"
#include "pressureControl.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"
```

These header files defines the classes and types needed by finite volume method, turbulent model, physical properties, combustion model, PIMPLE algorithm and so on. These are "real" header files, which are just declarations. It will be an endless work if we want to dig into every H file. We can go back to them to find the hierarchy relations of the classes, if we need to figure out the details.

Then we can go into `main()` function. The main function starts with including another block of H files.

```
{
    #include "postProcess.H"
    #include "addCheckCaseOptions.H"
    ...
    #include "initContinuityErrs.H"
```

These are certain code segment instead of "real" header files. We will have a quick look on each file and come back to them when we need details.

```
   #include "postProcess.H"
```

Execute application functionObjects to post-process results.

```
#include "addCheckCaseOptions.H"
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"
```

These four files are located in `$FOAM_SRC/OpenFOAM/include` folder, and they are the basic initial settings for starting a new case.

`addCheckCaseOptions.H` includes predefined arguments to check case set-up.

`setRootCase.H` will create an object from `argList` class to register and output informations according the case settings. `argList` is a very basic class defined in `argList.H`, and its function is to extract command arguments and options from the inputs.

`createTime.H` will create the object `runtime` from the `Time` class. This class is used to control time loops during the calculation.

`createMesh.H` will create the object `mesh` from the `fvMesh` class. `fvMesh` is a multiple inheritence class, and it inherits from `polyMesh` class and other classes related to finite volume method.

```
#include "createControl.H"
```

`createControl.H` is located in `$FOAM_SRC/finiteVolume/cfdTools/general/solutionControl`. In `createControl.H` three different solver controllers are defined (SIMPLE, PISO and PIMPLE). In `coalChemistryFoam`, PIMPLE algorithm is used. The three algorithm control classes are inherited from the `solutionControl` class, while `solutionControl` is inherited from `IOobject` class. These algorithm control class will supply convergence information checks for the calculation loop. The implementation of pressure correction algorithm is usually in the `pEqn.H` file.

```
#include "creatTimeControls.H"
```

`creatTimeControls.H` is located in `$FOAM_SRC/finiteVolume/cfdTools/general/include` Three variables `adjustTimeStep`, `maxCo` and `maxDeltaT` will be created and initialized from the `controlDict`.

```
#include "createFields.H"
```

In this file, the variables are created as fields data object. It is worth to notice that it also includes new H files, and defined two pointer variables which in OpenFOAM are from `autoPrt` template type. The two pointers are `combustion` for combustion model and `turbulent` for turbulence model. The pointer created by New function is using for the "Run Time Selection(RTS)" method, it allows to select the submodel class used in template after compiling of the source code. We will look into more detail in this file in the following sections.

```
#include createFieldRefs.H"
```

This file will create a reference field, if the inertia species are defined.

```
#include initContinuityErrs.H
```

Declare and initialise the cumulative continuity error.

Now the object and variable declaration is almost done. Before the calculation, check the turbulent model by following code.

```
turbulence->validate();
```

Before entering the time loop, set initial time step and calculate Co number.

```
    if (!LTS)
    {
        #include "compressibleCourantNo.H"
        #include "setInitialDeltaT.H"
    }

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

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readTimeControls.H"

        if (LTS)
        {
            #include "setRDeltaT.H"
        }
        else
        {
            #include "compressibleCourantNo.H"
            #include "setDeltaT.H"
        }
        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;
```

If the adjust time step model turns on, each time step during the time loop is determined by the Courant number. LTS is short for local time stepping. When the reactions and radiation model is activated, LTS is usually used. It needs to be specified according to the solver itself, so we can find the file "`setDeltaT.H`" in the solver's directory.

In fact a lot of such universal code segments are made into H files, and are located in `$FOAM_SRC/finiteVolume/` or `$FOAM_SRC/OpenFOAM/` directories. However, in some solvers these files need to be modified. Then the modified H files will be provided in the same folder as the C file, where the compiler will look for firstly. It is good to check whether we need to modify such universal H files, when we doing changes to a solver.

Then, the solver calculates and updates particles' parameter by calling the `evolve()` function in the parcel class, the source terms from the particle to the gas phase equations will also be calculated. `rhoEffLagrangian` and `pDyn` is not used in the calculation. They are only calculated for post-process.

```
        rhoEffLagrangian = coalParcels.rhoEff() + limestoneParcels.rhoEff();
        pDyn = 0.5*rho*magSqr(U);

        coalParcels.evolve();

        limestoneParcels.evolve();
```

Next, solve the governing eqautions for the gas phase, and correct the pressure using PIMPLE algorithm.

```
        #include "rhoEqn.H"

        // --- Pressure-velocity PIMPLE corrector loop
```

```
        while (pimple.loop())
        {
            #include "UEqn.H"
            #include "YEqn.H"
            #include "EEqn.H"

            // --- Pressure corrector loop
            while (pimple.correct())
            {
                #include "pEqn.H"
            }
```

Calculate the effective viscosity and diffusivity by calling the turbulence model's correct function. The "effective" includes turbulent contributions. The laminar viscosity and diffusivity will be calculated first. In OpenFOAM, the laminar model is also united into turbulence models. It means that we should choose the turbulence model as laminar if no turbulence models are used.

```
            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }
```

At last, the density of the gas phase is updated according to the "equation of state" model, and write data before entering the next time step.

```
        rho = thermo.rho();

        runTime.write();
    }
```

The main code ends here with `return 0;`. We know roughly how the solver is designed, now we need to go into more details.

2.3 How the equations are solved

In OpenFOAM the finite volume method is used. We will go into the equation files to see how the governing equations are implemented in the code.

2.3.1 rhoEqn.H

In `rhoEqn.H` the continuity equation is defined as:

```
fvScalarMatrix rhoEqn
(
    fvm::ddt(rho)
  + fvc::div(phi)
    ==
    coalParcels.Srho(rho)
  + fvOptions(rho)
);
```

The variable `phi` in OpenFOAM is usually the flux, and it is a `surfaceScalarField`, which means it is calculated by interpolation from cell centre values. In different solvers, it could be mass flux or volumetric flux. In `createFields.H`, by including `compressibleCreatePhi.H`, `phi` is defined as:

`linearInterpolate(rho*U) & mesh.Sf()`

The code `coalParcels.Srho(rho)` will return the mass source term from the coal parcel, and `fvOptions(rho)` will operate user-defined calculation. If nothing add, this term can be neglect from the equation.

Doing integration of Eq. 2.1 by volume, we get

$$\int_V \left[\frac{\partial \rho_g}{\partial t} + \nabla \cdot (\rho_g \mathbf{U_g})\right] dV = \int_V [S_m]\, dV \tag{2.5}$$

For term $\int_V \nabla \cdot (\rho_g \mathbf{U_g}) dV$, we use Gauss theorem. For one cell, the integration of the surfaces is calculated by the summation of all of the cell faces, we can get

$$\int_V \nabla \cdot (\rho_g \mathbf{U_g}) dV = \oint_{\partial V} (\rho_g \mathbf{U}_g) \cdot d\mathbf{S} = \sum_f \int_{S_f} (\rho_g \mathbf{U}_g) \cdot d\mathbf{S_f} = \sum_f (\rho_g \mathbf{U}_g)_f \cdot \mathbf{S_f} \tag{2.6}$$

So the mass balance equation 2.1 for one cell with a volume of V becomes:

$$\frac{\partial \rho_g}{\partial t} V + \sum_f (\rho_g \mathbf{U_g})_\mathbf{f} \cdot \mathbf{S_f} = S_m V \tag{2.7}$$

Now we get every term in the code corresponding to Eq. 2.7. Calling the `solve()` function to solve the equation, the density `rho` will be updated according to continuity.

```
rhoEqn.solve();

fvOptions.correct(rho);
```

2.3.2 UEqn.H

In `UEqn.H` The momentum equation is defined as:

```
fvVectorMatrix UEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(U)
    ==
    rho()*g
  + coalParcels.SU(U)
  + limestoneParcels.SU(U)
  + fvOptions(rho, U)
);
```

`MRF` is for rotating reference frame, `MRF.DDt` will return the contribution from the rotating reference frame effects. `divDevRhoReff(U)` function is determined by the turbulence model. It will return the source term for the momentum equation. This source term only includes the viscosity term and Reynolds stress term. For example, if a turbulence model except laminar model is used, the function defined in `$FOAM_SRC/TurbulenceModels/turbulenceModels/ReynoldsStress/ReynoldsStress.C` will be called, and if the `couplingFactor` is 0 (in all the OpenFOAM tutorial cases, `couplingFactor` is 0. It will weight the explicit calculation of the Reynold stress term, and details could be found in the definition), the return value of `divDevRhoReff` is defined as:

```
fvc::laplacian
(
    this->alpha_*rho*this->nut(),
```

```
            U,
            "laplacian(nuEff,U)"
        )
      + fvc::div(this->alpha_*rho*R_)
      - fvc::div(this->alpha_*rho*this->nu()*dev2(T(fvc::grad(U))))
      - fvm::laplacian(this->alpha_*rho*this->nuEff(), U)
```

`alpha_` is the phase fraction. In `coalChemistryFoam`, it is not considered. Thereby its value is 1. `nut()` is the turbulent viscosity calculated from turbulence model. `nuEff()` is the summation of laminar viscosity and turbulent viscosity. R is Reynold stress tensor and the calculation of `dev2(T(fvc::grad(U)))` equals to

$$dev2((\nabla \mathbf{U})^T) = (\nabla \mathbf{U})^T - \frac{2}{3} tr((\nabla \mathbf{U})^T)\mathbf{I} = (\nabla \mathbf{U})^T - \frac{2}{3}(\nabla \cdot \mathbf{U})\mathbf{I} \tag{2.8}$$

In OpenFOAM, the operation `fvc` will calculate the results as the source term, so we can regard the `fvc` term as explicit term. However, the `fvm` will return the discrete matrix coefficient, and it make the `fvm` term as implicit term. We can interpret the code of the definition of `divDevRhoReff` into the following equation

$$\begin{aligned}
&[\nabla \cdot (\nu_t \rho \nabla \mathbf{U}) + \nabla \cdot (\rho R) - \nabla \cdot (\nu\rho(\nabla \mathbf{U})^T - \frac{2}{3}\nu\rho(\nabla \cdot \mathbf{U})\mathbf{I})]_{explicit} - [\nabla \cdot ((\nu + \nu_t)\rho \nabla \mathbf{U})]_{implicit} \\
&\quad = -[\nabla \cdot (\nu\rho\nabla \mathbf{U})]_{implicit} - [\nabla \cdot (\nu\rho(\nabla \mathbf{U})^T - \frac{2}{3}\nu\rho(\nabla \cdot \mathbf{U})\mathbf{I})]_{explicit} + [\nabla \cdot (\rho R)]_{explicit} \\
&\qquad + [\nabla \cdot (\nu_t \rho \nabla \mathbf{U})]_{explicit} - [\nabla \cdot (\nu_t \rho \nabla \mathbf{U})]_{implicit}
\end{aligned} \tag{2.9}$$

here ν is laminar viscosity and ν_t is turbulent viscosity. The equation above is just the $\nabla \cdot (\tau_g) + \nabla \cdot (\rho_g \mathbf{R}_g)$ term using Stokes assumption is the momentum equation 2.2. For the convection term $\nabla \cdot (\rho_g \mathbf{U_g} \mathbf{U_g})$, again use Gauss theorem like we did in the mass balance equation above, we get

$$\begin{aligned}
\int_V \nabla \cdot (\rho_g \mathbf{U}_g \mathbf{U}_g) dV = \oint_{\partial V} (\rho_g \mathbf{U}_g \mathbf{U}_g) \cdot d\mathbf{S} &= \sum_f \int_{S_f} (\rho_g \mathbf{U}_g \mathbf{U}_g) \cdot d\mathbf{S_f} \\
&= \sum_f (\rho_g \mathbf{U_g} \mathbf{U_g})_\mathbf{f} \cdot \mathbf{S_f} \approx \sum_\mathbf{f} \mathbf{U_g}((\rho_g \mathbf{U_g})_\mathbf{f} \cdot \mathbf{S_f})
\end{aligned} \tag{2.10}$$

Now compared with Eq.2.2, we have found every term except the pressure term. How to couple the momentum equation with the pressure correction is always not a easy problem. In `coalChemistryFoam` the PIMPLE algorithm is used. First we predict the velocity and use PIMPLE algorithm to correct the pressure and velocity; it will be explained later. After applying relaxation factor and `fvOptions`, the velocity is predicted for PIMPLE loop.

```
if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));

    fvOptions.correct(U);
    K = 0.5*magSqr(U);
}
```

`K`, the (specific) kinetic energy is also calculated for the energy calculation.

2.3.3 YEqn.H

In the `YEqn.H` file, a `convectionScheme`, which is named `mvConvection`, is defined at the beginning. It is created on free store in random-access memory (RAM). When the number of gas species is large, it enhances the computational efficiency.

```
tmp<fv::convectionScheme<scalar>> mvConvection
(
    fv::convectionScheme<scalar>::New
    (
        mesh,
        fields,
        alphaRhoPhic,
        mesh.divScheme("div(alphaRhoPhi,Yi_h)")
    )
);
```

Before calculate the species transport equation, the gas phase combustion is calculated and the heat effects of the reactions is stored in the parameter `Qdot()`.

```
    combustion->correct();
    Qdot = combustion->Qdot();
```

A variable `Yt` is initialized and used in the calculation loop to make sure all the species mass fraction together is 1. The error is made up by the inert gas. The transport equation is defined as:

```
        fvScalarMatrix YiEqn
        (
            fvm::ddt(rho, Yi)
          + mvConvection->fvmDiv(phi, Yi)
          - fvm::laplacian(turbulence->muEff(), Yi)
            ==
            coalParcels.SYi(i, Yi)
          + combustion->R(Yi)
          + fvOptions(rho, Yi)
        );
```

With the analysis above, it is very strait forword the equation 2.4. It is important to update the boundary condition after solving the equation:

```
        fvOptions.constrain(YiEqn);
```

2.3.4 EEqn.H

The energy equation solves for the enthalpy of the gas species (Eq. 2.3). The kinetic energy is also added in the equation as explicit terms. The enthalpy h could also be internal energy e. The energy transport equation will be similar except for the pressure-volume work term, so there are two way to solve the energy equation. The code of the equation is defined as:

```
    fvScalarMatrix EEqn
    (
        fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
      + fvc::ddt(rho, K) + fvc::div(phi, K)
      + (
            he.name() == "e"
          ? fvc::div
            (
                fvc::absolute(phi/fvc::interpolate(rho), U),
                p,
                "div(phiv,p)"
            )
          : -dpdt
        )
```

```
      - fvm::laplacian(turbulence->alphaEff(), he)
    ==
        rho*(U&g)
      + Qdot
      + coalParcels.Sh(he)
      + limestoneParcels.Sh(he)
      + radiation->Sh(thermo, he)
      + fvOptions(rho, he)
    );
```

`alphaEff()` returns the effective turbulent thermal diffusivity for enthalpy. The code is also quite direct to be implemented from Eq. 2.3. After solving the energy equation, the thermal physical properties will be updated according to the new temperature, and the radiation will be calculated.

```
thermo.correct();
radiation->correct();
```

2.4 Pressure correction

In the velocity equation, the velocity needs to be solved together by the pressure correction. In `coalChemistryFoam` PIMPLE algorithm is used. To have a better understanding of the solver, a brief review of PIMPLE algorithm will be given first, then the implementation in `pEqn.H` will be explained.

2.4.1 PIMPLE algorithm

The PIMPLE Algorithm is a combination of PISO (Pressure Implicit with Splitting of Operator) and SIMPLE (Semi-Implicit Method for Pressure-Linked Equations). It is used for the transient case. The calculation scheme in one time step of PIMPLE Algorithm is:

- From the momentum equation predict the velocity U.
- Discrete momentum equation without the pressure term.

  $$A_P \mathbf{U}_P + \sum A_N \mathbf{U}_N - E = 0, \tag{2.11}$$

  here subscript P is the current cell and N is the neighbour cells, and E includes the source term. A new velocity can be calculated by adding the pressure gradient as:

  $$\mathbf{U}_P^* = \frac{1}{A_P}(\sum A_N \mathbf{U}_N - E - \nabla p) = \mathbf{HbyA}^* - \frac{1}{A_P}\nabla p \tag{2.12}$$

- $\mathbf{U}_P^*$ should satisfy the continuity equation, and get pressure correction. For example if the continuity equation is $\nabla \cdot U = 0$, then

  $$\nabla \cdot \mathbf{U}_P^* = \nabla \cdot (\mathbf{HbyA}^* - \frac{1}{A_P}\nabla p) = 0 \tag{2.13}$$

- Calculate pressure (for collocated mesh, Rhie-Chow interpolation may be used), and correct the velocities with the new pressure field.
- Return to step 2, until velocity is converged.

2.4.2 pEqn.H

The pressure correction is coupled with velocity calculation in `coalChemistryFoam`. First, the density is updated, and the coefficient and the velocity without pressure term `HbyA` is calculated:

```
rho = thermo.rho();

volScalarField rAU(1.0/UEqn.A());
surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
```

In `coalChemistryFoam` the fluid is compressible, so the solver have two ways of calculation. One is density based, the other is pressure based. If pressure based calculation is used, the `pimple.transonic()` value is false. The boundary conditions are applied as:

```
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (
            fvc::flux(rho*HbyA)
          + MRF.zeroFilter(rhorAUf*fvc::ddtCorr(rho, U, phi))
        )
    );

    MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p, rho, U, phiHbyA, rhorAUf, MRF);
```

Then the continuity equation 2.1 is used to derive the pressure equation. The Rhie-Chow interpolation is used, and velocity is interpolated on the surfaces to calculate the flux. By using 2.12, we get

$$\frac{\partial(\rho_g)}{\partial t} + \nabla \cdot (\rho_g \mathbf{HbyA}_f^* - \frac{\rho_g}{A_{P,f}} \nabla p) = S_m \tag{2.14}$$

Rearrange the terms we get

$$\frac{\partial(\rho_g)}{\partial t} + \nabla \cdot (\rho_g \mathbf{HbyA}_f^*) - \nabla \cdot (\frac{\rho_g}{A_{P,f}} \nabla p) = S_m \tag{2.15}$$

The equation above is exactly the same as in the code for constructing p equation

```
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phiHbyA)
          - fvm::laplacian(rhorAUf, p)
            ==
            coalParcels.Srho()
          + fvOptions(psi, p, rho.name())
        );
```

Here, the state equation is used for the compressible fluid:

$$\rho = \psi p \tag{2.16}$$

Then the p equation is solved and the non-orthogonality is corrected, the surface flux is updated.

```
        pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));

        if (pimple.finalNonOrthogonalIter())
        {
         phi = phiHbyA + pEqn.flux();
        }
```

Next, the continuity is constrained by calculating Eq. 2.1 again. The velocity is corrected according to the new calculated pressure, and at last, the kinetic energy and the energy term due to pressure are also updated for the energy calculation.

```
U = HbyA - rAU*fvc::grad(p);
U.correctBoundaryConditions();
fvOptions.correct(U);

K = 0.5*magSqr(U);

if (thermo.dpdt())
{
    dpdt = fvc::ddt(p);
}
```

If the density based calculation is used, the density in the second term of the Eq. 2.15 should also be replaced by the state equation.

2.5 Submodels

We have seen how the equations are solved in `coalChemistryFoam`. To have a better understanding of the code, another important part is essential, which is how the submodels are coupled with the solver. Noticing that in the solvers folder, there are only two files we haven't look at, `createClouds.H` and `createFields.H`. `createClouds.H` is included in `createFields.H`, and it includes the declaration of the particle classes. It is worth to mention that the particles are solved at different levels. The most basic is particle class, then particles with the same properties sharing the same calculation form parcels, and a cloud can include many parcels.

In the definition of submodels object, RTS mechanism is used. In fact, RTS is widely used in OpenFOAM. Here a brief introduction of RTS is described before we go into `createFields.H` file.

RTS makes it possible to construct a derived class using a base class as its interface. It is designed to have the function of "virtual constructor function", because in C++ the constructor function can not be a virtual function. It will be too complicated to explain it in detail, which is also beyond the scope of this tutorial. Since we are trying to understand the code, so the basic step to implement RTS for a set of class is described as

- In the base class, use macro `declareRunTimeSelectionTable` to create a hash table as a static member. In this table, all the elements are pairs of key-value terms. The key is the name of the derived class which is generated by `typeName` function following the OpenFOAM naming rules, and the value is a pointer pointing to the constructor function of the derived class.
- Every time that a derived class is defined from the base class, the derived class will use macro `addToRunTimeSelectionTable` to add a key-value of its name and constructor function pointer to the hash table.
- During compiling, the hashTable will be updated, and will contain all the derived class.

- In the base class a `New` function is defined as a static member, and its type is also a pointer `autoPtr`. It can receive the augments and return the pointer to constructor of the selected derived class from the hash table. Then the RTS is realized, an object of the derived class is created using a base class pointer.

2.5.1 createFields.H

In this H files, it includes several other H files.

```
#include "createRDeltaT.H"
#include "readGravitationalAcceleration.H"
#include "compressibleCreatePhi.H"
#include "createDpdt.H"
#include "createK.H"
#include "createMRF.H"
#include "createClouds.H"
#include "createRadiationModel.H"
#include "createFvOptions.H"
```

They are just code segments and by the name of the H files we can easily tell the function of these files. For example, `createRDeltaT.H` is used to create reciprocal local face time-step field for LTS model, and `compressibleCreatePhi.H` is the flux times the density by interpolation method. Both of them are already mention above.

The variables are defined, and some are defined with the argument `IOobject`. `IOobject` is more convenient to control the input and output of the variables. There are also three pointers that are defined as `autoPtr`.

```
Info<< "Reading thermophysical properties\n" << endl;
autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));
...
Info<< "Creating turbulence model\n" << endl;
autoPtr<compressible::turbulenceModel> turbulence
(
    compressible::turbulenceModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
);

Info<< "Creating combustion model\n" << endl;
autoPtr<CombustionModel<psiReactionThermo>> combustion
(
    CombustionModel<psiReactionThermo>::New(thermo, turbulence())
);
```

They are all initialized with a `New` functions. The RTS mention above is used here. The thermophysical model, turbulence model and combustion model are then coupled into the solver.

Chapter 3

coalChemistryAlphaFoam

In this chapter a solver based on `coalChemistryFoam` is developed. The solid volume fraction is considered by adding the gas phase volume fraction α to the governing equations in the new solver. First, the new governing equations will be given, then the changes to the code will be made to solve the new equations.

3.1 New governing euqations

According to the theory background in chapter 1, the equation for fluid-particle system have two different model in formulation. The difference lies in the decomposition of the particle-fluid interaction force. Please notice that the different drag force in OpenFOAM have different forms. Basically if the force models are changed the modified solver may not be strictly valid in the governing equations. In this case, the governing equations for the gas phase need to be adjust. Here we use the settings in the tutorial case for `coalChemistryFoam`, where the particle force is set as:

```
particleForces
{
    sphereDrag;
    gravity;
}
```

The source codes to calculate parcel force are in the `$FOAM_SRC/lagrangian/intermediate/submodels/Kinematic/ParticleForces` folder. The drag force is coupled to the Eulerian governing equations while the gravity is uncoupled. The buoyancy force is also included in the `gravity`. "Uncoupled" means it does not have any contribution in the fluid momentum equation. It will be easier to use model A to implement. The $\mathbf{f}_i^{''}$ term in 1.2 is neglected, which is acceptable. While for the particle, the buoyancy force is part of $\mathbf{f}_{\nabla p,i}$, and the rest force is also neglected. It is also reasonable, because the drag force and gravity are dominating the particle motion. Then we can get the set of the new governing equations.

The continuity equation is

$$\frac{\partial(\alpha\rho_g)}{\partial t} + \nabla\cdot(\alpha\rho_g\mathbf{U}_g) = S_m \tag{3.1}$$

The momentum equation

$$\frac{\partial(\alpha\rho_g\mathbf{U}_g)}{\partial t} + \nabla\cdot(\alpha\rho_g\mathbf{U}_g\mathbf{U}_g) - \nabla\cdot(\alpha\tau_g) - \nabla\cdot(\alpha\rho_g\mathbf{R}_g) = -\alpha\nabla p + \alpha\rho_g\mathbf{g} + S_\mathbf{U} \tag{3.2}$$

The energy transport equation is

$$\frac{\partial(\alpha\rho_g(h+K))}{\partial t} + \nabla\cdot(\alpha\rho_g\mathbf{U}_g(h+K)) - \nabla\cdot\big(\alpha\alpha_{eff}\nabla(\mathrm{h})\big) = \alpha\nabla p + \alpha\rho_g\mathbf{U}_g\cdot\mathbf{g} + S_h \tag{3.3}$$

Species transport equation is

$$\frac{\partial(\alpha\rho_g Y_i)}{\partial t} + \nabla\cdot(\alpha\rho_g \mathbf{U}_g Y_i) - \nabla\cdot\left(\alpha D_{eff}\nabla(\rho_g Y_i)\right) = S_i \tag{3.4}$$

From the chapter 2 we have seen how the equations are implemented into the code. For the new solver, the surface flux needs to be modified by multiplying the phase volume fraction. There are two main tasks. The first one is the Laplace diffusion terms is calculated from the turbulence model, and it means a new turbulence models is required to take the volume fraction of the gas phase into account. The second one is that the pressure correction needs to be specified according to the new equations.

3.2 Implementation

The new solver `coalChemistryAlphaFoam` is modified on `coalChemistryFoam`. The implementation follows the step below:

- Create the new fields variables.
- Create the new turbulence model.
- Update the calculation of the equations

Before beginning, we should prepare the solver. Go to the user directory (or the directory preferred), copy `coalChemistryFoam` to the folder and rename it to `coalChemistryAlphaFoam`.

```
cp -r $FOAM_APP/solvers/lagrangian/coalChemistryFoam .
mv coalChemistryFoam/ coalChemistryAlphaFoam/
cd coalChemistryAlphaFoam/
```

Rename the C file, and redirect the compiling to user `bin` folder

```
mv coalChemistryFoam.C coalChemistryAlphaFoam.C
sed -i s/coalChemistryFoam/coalChemistryAlphaFoam/g Make/files
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
```

3.2.1 Alpha fields

In the `createFields.H` file, replace the code `#include "compressibleCreatePhi.H"` with the following code:

```
surfaceScalarField phi
(
    IOobject
    (
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(U) & mesh.Sf()
);
```

The old H file will create the mass surface flux, but we need volumetric flux in the new solver. Then add the following code after `#include "createClouds.H"`:

```
volScalarField alpha
(
    IOobject
    (
        "alpha",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar(dimless, Zero)
);

scalar alphaMin
(
    1.0
  - readScalar
    (
        coalParcels.particleProperties()
        .subDict("constantProperties").lookup("alphaMax")
    )
);

alpha = max(1.0 - coalParcels.theta() - limestoneParcels.theta(), alphaMin);
alpha.correctBoundaryConditions();

surfaceScalarField alphaf("alphaf", fvc::interpolate(alpha));
Info<<"alphaf: "<<alphacf<<endl;

surfaceScalarField rhoPhi
(
    IOobject
    (
        "rhoPhi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    fvc::interpolate(rho)*phi
);

surfaceScalarField alphaRhoPhi
(
    IOobject
    (
        "alphaRhoPhi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    alphaf*rhoPhi
```

```
);
```

The alpha fields is defined, as well as the phase fraction weighted surface flux is. Notes that a minimum value of alpha is needed. This value should be provided in the `coalParcels` dictionary, in the `constantProperties` subDict, by name of `alphaMax`, the value should be `(1 - alphaMin)`.

3.2.2 coalChemistryTurbulenceModel

The turbulent model used in `coalChemistryFoam` is defined in the `createClouds.H` file:

```
autoPtr<compressible::turbulenceModel> turbulence
(
    compressible::turbulenceModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
)
```

The `compressible::turbulenceModel` is defined in the file `$FOAM_SRC/TurbulenceModels/compressible/turbulentFluidThermoModels/turbulentFluidThermoModel.H`

```
    namespace compressible
    {
        typedef ThermalDiffusivity<CompressibleTurbulenceModel<fluidThermo>>
            turbulenceModel;
...
```

This model is in the library `libcompressibleTurbulenceModels.so`. From the code we can see that the class `fluidThermo` is used as the type of transport model. The turbulence model is inherited from `CompressibleTurbulenceModel`, while we prefer to use `PhaseCompressibleTurbulenceModel`. The problem is that there is no library that instantiated the template `PhaseCompressibleTurbulenceModel` with class `fluidThermo`, so we need to compile a new library first. (To make such a library, thy way to make similar libraries in `multiphase` solvers can be used as references).

First, at the new solver directory create a sub-directory for the library of turbulence Model. In terminal using the following command:

```
mkdir coalChemistryTurbulenceModels
```

Then create three files to use macro to instantiate the template class.

```
touch coalChemistryTurbulenceModels/coalChemistryTurbulenceModel.H
touch coalChemistryTurbulenceModels/coalChemistryTurbulenceModelFwd.H
touch coalChemistryTurbulenceModels/coalChemistryTurbulenceModels.C
```

Add the following code to `coalChemistryTurbulenceModel.H` file

```
#ifndef coalChemistryTurbulenceModel_H
#define coalChemistryTurbulenceModel_H

#include "coalChemistryTurbulenceModelFwd.H"
#include "PhaseCompressibleTurbulenceModel.H"
#include "ThermalDiffusivity.H"
#include "fluidThermo.H"
```

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

namespace Foam
{
typedef ThermalDiffusivity<PhaseCompressibleTurbulenceModel<fluidThermo>>
coalChemistryTurbulenceModel;
}

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

#endif
```

Add the following code to `coalChemistryTurbulenceModelFwd.H` file

```
#ifndef coalChemistryTurbulenceModelFwd_H
#define coalChemistryTurbulenceModelFwd_H

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

namespace Foam
{
    class fluidThermo;

    template<class TransportModel>
    class PhaseCompressibleTurbulenceModel;

    template<class BasicTurbulenceModel>
    class ThermalDiffusivity;

    typedef ThermalDiffusivity<PhaseCompressibleTurbulenceModel<fluidThermo>>
        coalChemistryTurbulenceModel;
}

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

#endif
```

However, this file is not necessary, it is also OK to do not have this file. It just makes the code follow the structure in OpenFOAM.

Add the following code to `coalChemistryTurbulenceModels.C` file

```
#include "PhaseCompressibleTurbulenceModel.H"
#include "fluidThermo.H"
#include "addToRunTimeSelectionTable.H"
#include "makeTurbulenceModel.H"

#include "ThermalDiffusivity.H"
#include "EddyDiffusivity.H"

#include "laminarModel.H"
#include "RASModel.H"
#include "LESModel.H"

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

```
makeTurbulenceModelTypes
(
    volScalarField,
    volScalarField,
    compressibleTurbulenceModel,
    PhaseCompressibleTurbulenceModel,
    ThermalDiffusivity,
    fluidThermo
);

makeBaseTurbulenceModel
(
    volScalarField,
    volScalarField,
    compressibleTurbulenceModel,
    PhaseCompressibleTurbulenceModel,
    ThermalDiffusivity,
    fluidThermo
);

#define makeLaminarModel(Type)                                                \
    makeTemplatedLaminarModel                                                 \
    (fluidThermoPhaseCompressibleTurbulenceModel, laminar, Type)

#define makeRASModel(Type)                                                    \
    makeTemplatedTurbulenceModel                                              \
    (fluidThermoPhaseCompressibleTurbulenceModel, RAS, Type)

#define makeLESModel(Type)                                                    \
    makeTemplatedTurbulenceModel                                              \
    (fluidThermoPhaseCompressibleTurbulenceModel, LES, Type)

#include "Stokes.H"
makeLaminarModel(Stokes);

#include "kEpsilon.H"
makeRASModel(kEpsilon);

#include "kOmegaSST.H"
makeRASModel(kOmegaSST);

#include "Smagorinsky.H"
makeLESModel(Smagorinsky);

#include "kEqn.H"
makeLESModel(kEqn);
```

Only five turbulence models are added here. If other models are preferred, just add the new model using `makeLaminar/RAS/LESModel` macros.

Then make rules to compile the library. In terminal using command:

```
mkdir coalChemistryTurbulenceModels/Make
touch coalChemistryTurbulenceModels/Make/files
touch coalChemistryTurbulenceModels/Make/options
```

In `files` add:

```
coalChemistryTurbulenceModels.C

LIB = $(FOAM_USER_LIBBIN)/libcoalChemistryTurbulenceModels
```

In `options` add:

```
EXE_INC = \
    -I$(LIB_SRC)/transportModels/compressible/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/transportModel \
    -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/phaseCompressible/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lcompressibleTransportModels \
    -lfluidThermophysicalModels \
    -lspecie \
    -lturbulenceModels \
    -lcompressibleTurbulenceModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools
```

Since the new library will be used, we should add the new library into the `options` file of the new solver. After `EXE_INC = \`, add:

```
    -I../coalChemistryAlphaFoam/coalChemistryTurbulenceModels \
```

After `-I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \`, add:

```
    -I$(LIB_SRC)/TurbulenceModels/phaseCompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
```

After `EXE_LIBS = \`, add:

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

After `-lphaseCompressibleTurbulenceModels \`, add:

```
    -lcoalChemistryTurbulenceModels \
```

Now we can use the new turbulence model in the new solver. Again in the `createClouds.H` file, replace the declaration of the turbulence pointer with the following code:

```
autoPtr<coalChemistryTurbulenceModel> turbulence
(
    coalChemistryTurbulenceModel::New
    (
        alpha,
        rho,
        U,
        alphaRhoPhi,
        phi,
        thermo
    )
);
```

Then the new turbulence model can take the phase volume fraction into account.

3.2.3 Modification of the equations in the code

The modification of the equations in the code is quite straightforward to the new governing equations. For the mass balance equation in the `rhoEqn.H`, the code should be changed into:

```
    fvScalarMatrix rhoEqn
    (
        fvm::ddt(alpha, rho)
      + fvc::div(alphaRhoPhi)
        ==
        coalParcels.Srho(rho)
      + fvOptions(rho)
    );
```

For the `UEqn.H`, the code should be changed into:

```
fvVectorMatrix UEqn(U, rho.dimensions()*U.dimensions()*dimVol/dimTime);

    UEqn =
    (
        fvm::ddt(alpha, rho, U) + fvm::div(alphaRhoPhi, U)
      + MRF.DDt(alpha*rho, U)
      + turbulence->divDevRhoReff(U)
        ==
        alpha*rho()*g
      + coalParcels.SU(U)
      + limestoneParcels.SU(U)
      + fvOptions(rho, U)
    );
```

While inside the `momentumPredictor`, the pressure should be multiply by the phase fraction

```
        solve(UEqn == -alpha*fvc::grad(p));
```

For the `YEqn.H`, firstly, the surface mass flux should be corrected by the phase fraction:

```
tmp<fv::convectionScheme<scalar>> mvConvection
(
     fv::convectionScheme<scalar>::New
     (
          mesh,
          fields,
          alphaRhoPhi,
          mesh.divScheme("div(alphaRhoPhi,Yi_h)")
     )
);
```

Then, according to Eq. 3.4, the governing equation should be modified as

```
    const volScalarField muEff(turbulence->muEff());

    forAll(Y, i)
    {
        if (i != inertIndex && composition.active(i))
        {
            volScalarField& Yi = Y[i];
```

```
            fvScalarMatrix YiEqn
            (
                fvm::ddt(alpha, rho, Yi)
              + mvConvection->fvmDiv(alphaRhoPhi, Yi)
              - fvm::laplacian
                (
                    fvc::interpolate(alpha)
                   *fvc::interpolate(muEff),
                    Yi
                )
                ==
                coalParcels.SYi(i, Yi)
              + combustion->R(Yi)
              + fvOptions(rho, Yi)
            );
...
```

The `EEqn.H`, according to Eq. 3.3, should be modified as

```
    const volScalarField alphaEff(turbulence->alphaEff());

    fvScalarMatrix EEqn
    (
        fvm::ddt(alpha, rho, he) + mvConvection->fvmDiv(alphaRhoPhi, he)
      + fvc::ddt(alpha, rho, K) + fvc::div(alphaRhoPhi, K)
      + (
            he.name() == "e"
          ? fvc::div
            (
                fvc::absolute(alphaRhoPhi/fvc::interpolate(rho), U),
                p,
                "div(phiv,p)"
             )
          : -dpdt*alpha
        )
      - fvm::laplacian
        (
            fvc::interpolate(alpha)
           *fvc::interpolate(alphaEff),
            he
        )
        ==
        alpha*rhoc*(U&g)
      + Qdot
      + coalParcels.Sh(he)
      + limestoneParcels.Sh(he)
      + radiation->Sh(thermo, he)
      + fvOptions(rho, he)
    );
```

As for pressure correction, the pressure equation is derived from the continuity equation 3.1. For the new solver the equation should be:

$$\frac{\partial(\alpha\rho_g)}{\partial t} + \nabla\cdot(\alpha\rho_g \mathbf{HbyA}_f^*) - \nabla\cdot(\frac{\alpha\rho_g}{A_{P,f}}\alpha\nabla p) = S_m \tag{3.5}$$

For the modified code, when `pimple.transonic()` value is false,the code should be like:

```
else
{
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (
            fvc::flux(rho*HbyA)
          + MRF.zeroFilter(alphaf*rhorAUf*fvc::ddtCorr(rho, U, rhoPhi))
        )
    );

    if (p.needReference())
    {
        adjustPhi(phiHbyA, U, p);
    }

    MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p, rho, U, phiHbyA, rhorAUf, MRF);

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(alpha, psi, p)
          + fvc::div(alphaf*phiHbyA)
          - fvm::laplacian(alphaf*alphaf*rhorAUf, p)
            ==
            coalParcels.Srho()
          + fvOptions(psi, p, rho.name())
        );

        pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));

        if (pimple.finalNonOrthogonalIter())
        {
            rhoPhi = phiHbyA + pEqn.flux()/alphaf;
            phi = rhoPhi/fvc::interpolate(rho);
        }
    }
}
```

And when the case `pimple.transonic()` is true, which is very unlikely for coal combustion case, the code should be

```
    surfaceScalarField phid
    (
        "phid",
        fvc::interpolate(psi)
        *(
            fvc::flux(HbyA)
```

```
      + MRF.zeroFilter
        (
            rhorAUf*fvc::ddtCorr(rho, U, rhoPhi)/fvc::interpolate(rho)
        )
    )
);

MRF.makeRelative(fvc::interpolate(psi), phid);

while (pimple.correctNonOrthogonal())
{
    fvScalarMatrix pEqn
    (
        fvm::ddt(alpha, psi, p)
      + fvm::div(alphaf*phid, p)
      - fvm::laplacian(alphaf*alphaf*rhorAUf, p)
        ==
        coalParcels.Srho()
      + fvOptions(psi, p, rho.name())
    );

    pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));

    if (pimple.finalNonOrthogonalIter())
    {
        rhoPhi = pEqn.flux()/alphaf;
        phi = rhoPhi/fvc::interpolate(rho);
    }
}
```

At the ending, the surface flux need to be updated, and the velocity is calculated from the corrected pressure.

```
alphaRhoPhi = alphaf*rhoPhi;
U = HbyA - rAU*alpha*fvc::grad(p);
```

3.2.4 Completion of the solver

Here are some final steps to complete the modification of the solver. The First is to include the turbulence model in the main file, and to update the phase fraction in every time step. After `#include "turbulent FluidThermoModel.H"` , add the H file

```
#include "coalChemistryTurbulenceModel.H"
```

After calculating the parcels "`limestoneParcels.evolve();`", add the following code before "`#inclu de "rhoEqn.H"`"to update the phase fraction

```
alpha = max(1.0 - coalParcel.theta(), alphaMin);
alpha.correctBoundaryConditions();
alphaf = fvc::interpolate(alpha);
alphaRhoPhi = alphaf*rhoPhi;
```

Because we changed the flux `phi` from mass flux into volume flux, we need to modify some of the H files included from general source code. For the existing `setRDeltaT.H` file, change the code calculated with `phi` (line 61-65)into:

```
        rDeltaT.ref() =
        (
            fvc::surfaceSum(mag(phi))()()
            /((2*maxCo)*mesh.V())
        );
```

Copy two files to the solver folder. In terminal using command:

```
cp $FOAM_SRC/finiteVolume/cfdTools/compressible/compressibleContinuityErrs.H .
cp $FOAM_SRC/finiteVolume/cfdTools/compressible/compressibleCourantNo.H .
```

In the file `compressibleContinuityErrs.H` , change line 33 as:

```
volScalarField contErr(fvc::ddt(alpha, rho) + fvc::div(alphaRhoPhi) -
↪  coalParcels.Srho(rho)) ;
```

And in file `compressibleCourantNo.H`, change the calculation of `sumPhi` into the following code:

```
scalarField sumPhi
(
    fvc::surfaceSum(mag(rhoPhi))().primitiveField()/rho.primitiveField()
);
```

The code modification is finished. Compile the `coalChemistryTurbulenceModels` first, and then compile the solver. Use the command `coalChemistryAlphaFoam` to run the solver.

Chapter 4

Sample case

The new solver just changed the coupling calculations between the particles and the gas phase, so the tutorial case for `coalChemistryFoam`, `simplifiedSiwek`, could also be used as the sample case. However, some modification should be made for the settings.

4.1 Preparing the case

First go to the directory where is preferred to run the case, for example `$FOAM_RUN`. Copy the `simplifiedSiwek` case to the current folder, and rename it to `simplifiedAlphaSiwek` , using the commands

```
cp -r $FOAM_TUTORIALS/lagrangian/coalChemistryFoam/simplifiedSiwek .
mv simplifiedSiwek simplifiedAlphaSiwek
```

In the file `constant/coalCloud1Properties`, add the maximum solid phase fraction in the `constantProperties` subDict, after `"constantVolume  true;"`:

```
    alphaMax        0.99;
```

In the file `system/controlDict`, change the application name into `coalChemistryAlphaFoam`.

In the file `system/fvSchemes`, replace the `divSchemes` into the following code:

```
{
    default         none;

    div(alphaRhoPhi,U)      Gauss upwind;
    div((alphaf*phid),p)     Gauss upwind;
    div(alphaRhoPhi,K)      Gauss linear;
    div(alphaRhoPhi,h)     Gauss upwind;
    div(alphaRhoPhi,k)      Gauss upwind;
    div(alphaRhoPhi,epsilon) Gauss upwind;
    div(U)          Gauss linear;
    div((((alpha*rho)*nuEff)*dev2(T(grad(U))))) Gauss linear;
    div(alphaRhoPhi,Yi_h)   Gauss upwind;
}
```

In the original tutorial case, the solid volume fraction is too small. In order to have a more suitable test case, the diameters of the limestone particles are increased by 10 times, and at the same time the density is reduced by 100 times. It is of course not practical, and it is just used to test the solver. In the `constant/limestoneCloud1Properties` file, the code is modified as

```
constantProperties
{
    parcelTypeId    2;

    rho0            25;
...
}

subModels
{
    ...

    injectionModels
    {
        ...
        model1
        {
            ...
            sizeDistribution
            {
                type        RosinRammler;
                RosinRammlerDistribution
                {
                    minValue        5e-05;
                    maxValue        0.00565;
                    d               4.8e-04;
                    n               0.5;
                }
            }
        }
    }
    ...
}
```

The number of injected limestone particles is also increased by 3 times by add more injecting position. Another 36 positions shown as follows are added to the `constant/limestonePositions` file.

```
(0.0075 0.50 0.05)
(0.0125 0.50 0.05)
(0.0175 0.50 0.05)
(0.0225 0.50 0.05)
(0.0275 0.50 0.05)
(0.0325 0.50 0.05)
(0.0375 0.50 0.05)
(0.0425 0.50 0.05)
(0.0475 0.50 0.05)
(0.0075 0.40 0.05)
(0.0125 0.40 0.05)
(0.0175 0.40 0.05)
(0.0225 0.40 0.05)
(0.0275 0.40 0.05)
(0.0325 0.40 0.05)
(0.0375 0.40 0.05)
(0.0425 0.40 0.05)
(0.0475 0.40 0.05)
```

```
(0.0075 0.52 0.05)
(0.0125 0.52 0.05)
(0.0175 0.52 0.05)
(0.0225 0.52 0.05)
(0.0275 0.52 0.05)
(0.0325 0.52 0.05)
(0.0375 0.52 0.05)
(0.0425 0.52 0.05)
(0.0475 0.52 0.05)
(0.0075 0.47 0.05)
(0.0125 0.47 0.05)
(0.0175 0.47 0.05)
(0.0225 0.47 0.05)
(0.0275 0.47 0.05)
(0.0325 0.47 0.05)
(0.0375 0.47 0.05)
(0.0425 0.47 0.05)
(0.0475 0.47 0.05)
```

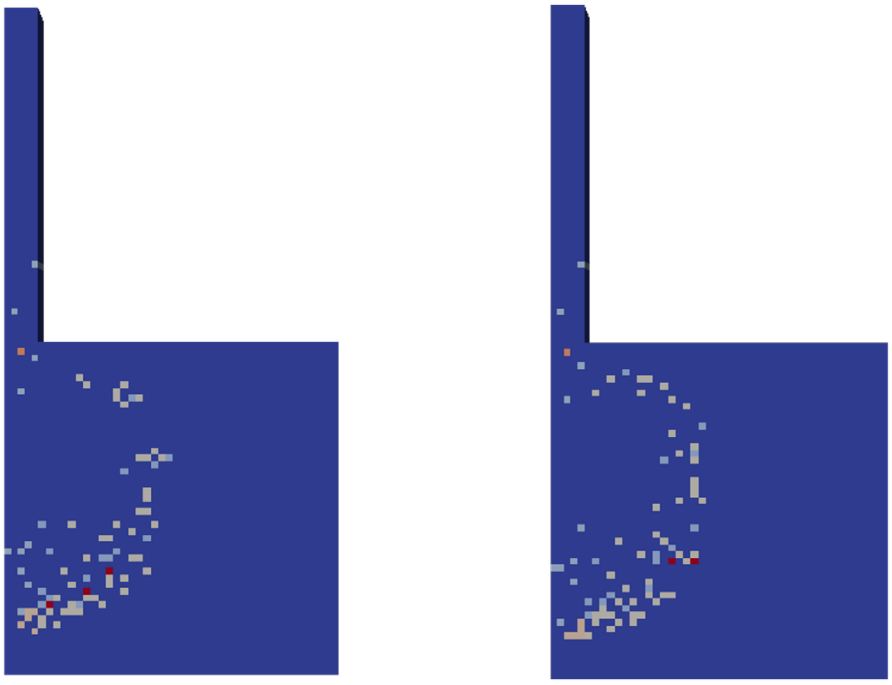
4.2 Results

Run the script `Allrun`, in terminal using:

```
./Allrun
```

The same case is also run by the `coalChemistryFoam` solver as a comparison.

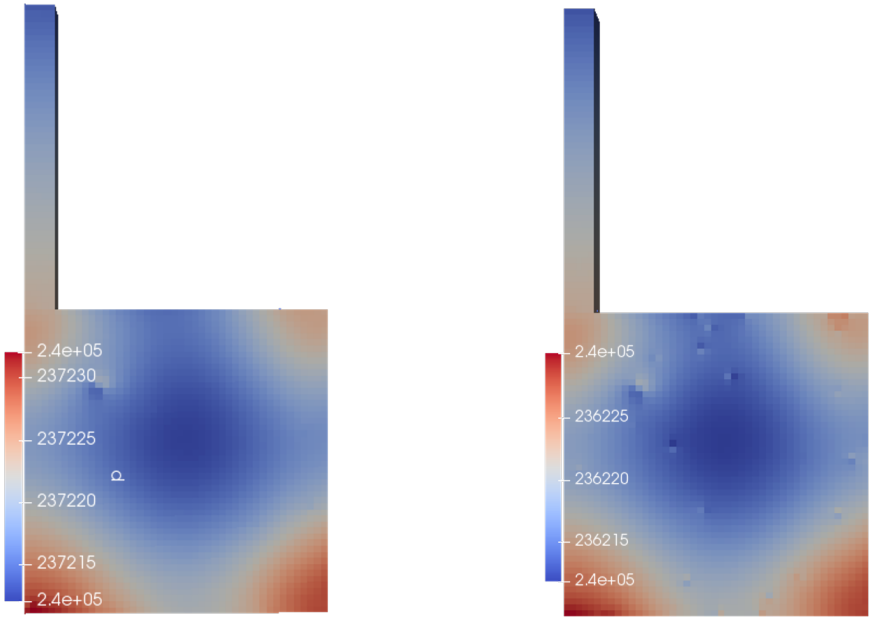
`rhoEffLagrangian`, which shows the particles concentration for the first written time step in figure below shows that the movement of the particle is already different at beginning. It shows that velocity fields predicted by the two solvers have notable difference.



(a) coalChemistryFoam (b) coalChemistryAlphaFoam

Figure 4.1: rhoEffLagrangian at first written time step

Compared together with the gas phase volume fraction distribution and the pressure profile, it shows that `coalChemistryAlphaFoam` is more likely to predict a relatively higher pressure gradient at high solid volume fraction region. It is as expected that the local pressure field around the particles will be more uneven when considering the phase volume fraction effects.



(a) coalChemistryFoam (b) coalChemistryAlphaFoam

Figure 4.2: Pressure at 0.5s

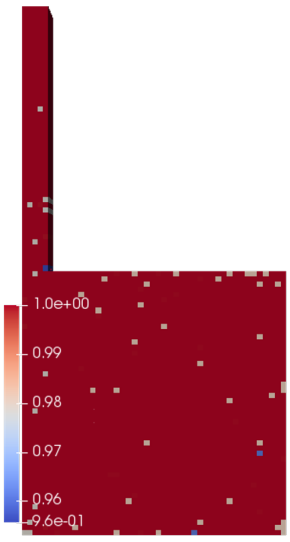


Figure 4.3: Gas phase volume fraction

References

[1] Zhou, Z. Y., Kuang, S. B., Chu, K. W., & Yu, A. B. (2010). Discrete particle simulation of particle–fluid flow: model formulations and their applicability. Journal of Fluid Mechanics, 661, 482-510.

[2] Leboreiro, J. (2008). Influence of drag laws on segregation and bubbling behavior in gas-fluidized beds (Doctoral dissertation, Ph. D. Thesis, University of Colorado, Boulder, USA).

Study questions

- 1. How does the solver get the physical properties like the viscosity of the fluid?
- 2. Why we have to set a minimum fluid volume fraction?
- 3. In one time step, where is the velocity calculated?
- 4. Why should we make a library when we want to use `PhaseCompressibleTurbulenceModel`?