Cite as: Cappelli, D.: A detailed description of reactngTwoPhaseEulerFoam focussing on the links between mass and heat transfer at the phase interface. 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

A detailed descrption of reactingTwoPhaseEulerFoam focussing on the links between mass and heat transfer at the interface

Developed for OpenFOAM-v1806

Author:
Darren CAPPELLI
University College Dublin
darren.cappelli@ucdconnect.ie

Peer reviewed by:
Mohammad H. Arabnejad
Ashkhen Nalbandyan

December 19, 2018

Learning outcomes

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

The reader will learn:

How to use it:

- How to use the reactingTwoPhaseEulerFoam solver.
- The case example will be a laminar bubble column with air and water from the OpenFOAM tutorials.

The theory of it:

- The theory behind two phase flow will be discussed along with the theory of mass and heat transfer at the interface.
- The equations used by OpenFOAM in this solver will be derived from first principles and analyzed.

How it is implemented:

- The algorithm used in reactingTwoPhaseEulerFoam will be followed in a logical and coherent manner.
- The equations in the code will be compared to the equations in the theory.

How to modify it:

- A new case set up using methanol as an alternative to water will be described.
- A new saturation model will be developed and implemented.

Prerequisites

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

- It is recommended to have a fundamental understanding of the physics behind mass and heat transfer. Chapters 6 and 14 of Fundamentals of heat and mass transfer , Book by Frank.P Incropera, provide a good description of the topics.
- Dimensional groups like Sherwood number,Reynolds,Number,Prandtl Number, Schmidt Number and Lewis Number
- Run standard document tutorials like damBreak tutorial
- The physics of two-phase flows are described in the thesis of, Rusche, H.: Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions, 2002.
- The following report describes the MULES and PIMPLE algorithm employed in OpenFOAM, Manni, A.:An Introduction to twoPhaseEulerFoam with addition of a heat exchange model. In proceedings of CFD with OpenSourceSoftware, 2014.
- The following report describes the mass transfer models employed in the reactingEulerFoam solvers, Phanindra.P.Thummala: Description of reactingTwoPhaseEulerFoam solver with a focus on mass transfer modeling terms. In Proceedings of CFD with OpenSource Software, 2016.
- The following report describes how the multi-phase system is set up in multiphaseEulerFoam, Surya Kaundinya,O.: Implementation of cavitation models into the multiphaseEulerFoam solver. In Proceedings of CFD with OpenSource Software, 2017.

Contents

Chapter 1

Theory

This chapter will explain the theory employed by the reactingTwoPhaseEulerFoam solver. In section 2.1 it was explained that the solver uses a two-fluid Eulerian model to describe the system, where both phases are treated as inter-penetrating continua, Rushe (2002). Below was taken from Rushe's thesis.

The mean properties of the flow for each phase, φ are described by the averaged conservation equations for mass and momentum,

$$\frac{\partial \alpha_\varphi \bar{\mathbf{U}}_\varphi}{\partial t} + \nabla \cdot (\alpha_\varphi \bar{\mathbf{U}}_\varphi \bar{\mathbf{U}}_\varphi) + \nabla \cdot (\alpha_\varphi \bar{\mathbf{R}}_\varphi{}^{eff}) = -\frac{\alpha_\varphi}{\rho_\varphi} \nabla \bar{p} + \alpha_\varphi \mathbf{g} + \frac{\bar{\mathbf{M}}_\varphi}{\rho_\varphi} \tag{1.1}$$

$$\frac{\partial \alpha_\varphi}{\partial t} + \nabla \cdot (\bar{\mathbf{U}}_\varphi \alpha_\varphi) = 0 \tag{1.2}$$

where the subscript φ denotes the phase, α is the phase fraction, $\bar{\mathbf{U}}_\varphi$ is the velocity of the phase, $\bar{\mathbf{R}}_\varphi{}^{eff}$ is the combined Reynolds and viscous stress, ρ_φ is the density of the phase, $\bar{p}$ is the system pressure, g is the gravitation constant and $\bar{\mathbf{M}}_\varphi$ is the averaged inter-phase momentum transfer term.

The averaged inter-phase momentum transfer term is required to ensure conservation of momentum. This is required for multi-phase systems as momentum can transfer between phases. This terms main contributions are due to drag, lift and virtual mass. However, in the case of mass transfer, an extra term is added. This term takes into account the momentum entering and leaving the phase during mass transfer and is defined by the equation below.

$$\bar{\mathbf{M}}_{\mathbf{MT}\varphi} = \dot{m}_{\varphi In} \bar{\mathbf{U}}_{\varphi^{-1}} - \dot{m}_{\varphi Out} \bar{\mathbf{U}}_\varphi \tag{1.3}$$

where the subscripts φ^{-1} denotes the other phase, In denotes into the phase, Out denotes out of the phase, $\dot{m}$ is the mass transfer rate and $\bar{\mathbf{M}}_{\mathbf{MT}\varphi}$ is the mass transfer component of the momentum transfer term.

For the two-phase system described in chapter 2, both the gas and liquid phases will have their own momentum conservation equation (equation 1.1). There will be one value for the inter-phase momentum transfer for the system, however, the sign of the term will vary depending on the direction of momentum transfer.

Weller re-arranged the phase continuity equation 1.2 so that by solving the equation for one phase, the phase fraction α_φ remains bounded between 0 and 1.

$$\frac{\partial \alpha_\varphi}{\partial t} + \nabla \cdot (\bar{\mathbf{U}} \alpha_\varphi) + \nabla \cdot (\bar{\mathbf{U}}_{\mathbf{r}} \alpha_\varphi (1 - \alpha_\varphi)) = 0 \tag{1.4}$$

where $\bar{\mathbf{U}} = \alpha_\varphi \bar{\mathbf{U}_\varphi} + \alpha_{\varphi^{-1}} \bar{\mathbf{U}_{\varphi^{-1}}}$ and $\bar{\mathbf{U_r}} = \bar{\mathbf{U}_\varphi} - \bar{\mathbf{U}_{\varphi^{-1}}}$.

The energy equations for each phase are described below,

$$\frac{\partial \alpha_\varphi \rho_\varphi (\mathbf{H}_\varphi + \mathbf{K}_\varphi)}{\partial t} + \nabla \cdot (\alpha_\varphi \rho_\varphi \mathbf{U}_\varphi (\mathbf{H}_\varphi + \mathbf{K}_\varphi)) - \nabla \cdot (\alpha_\varphi \bar{\alpha_\varphi}^{eff} \nabla \mathbf{H}_\varphi) = \tag{1.5}$$

$$\alpha_\varphi \frac{\partial \bar{p}}{\partial t} + \rho_\varphi \mathbf{g} \cdot \mathbf{U}_\varphi + \alpha_\varphi \dot{\mathbf{Q}_R} + \dot{\mathbf{Q}_{KD}} + \dot{\mathbf{Q}_{TD}}$$

where $\mathbf{H}_\varphi$ denotes the enthalpy of phase φ, $\mathbf{K}_\varphi$ denotes the kinetic energy of phase φ, $\bar{\alpha_\varphi}^{eff}$ denotes the effective thermal diffusivity of the phase, $\dot{\mathbf{Q}_R}$ denotes the rate of heat change due to reaction, $\dot{\mathbf{Q}_{KE}}$ denotes the rate of heat change due to a difference in kinetic energy and $\dot{\mathbf{Q}_{TD}}$ denotes the rate of heat change due to temperature difference. The rate of heat change due to kinetic energy difference and temperature difference are described by the equations below,

$$\dot{\mathbf{Q}_{KE}} = \dot{m_{\varphi In}} (\mathbf{K}_{\varphi^{-1}} - \mathbf{K}_\varphi) \tag{1.6}$$

$$\dot{\mathbf{Q}_{TD}} = h_{i\varphi} (T_f - T_\varphi) \tag{1.7}$$

where $h_{i\varphi}$ denotes the heat transfer coefficient of species i in phase φ, T_f denotes the interface temperature and T_φ denotes the phase temperature.

The above terms take into account the energy effects that mass transfer will have in the system. Transferring mass will carry its own kinetic and heat energy. Equation 1.6 describes the change in kinetic energy for the phase. The heat energy is imparted to the phase through a temperature difference with the interface. The temperature at the interface changes as a result of phase change and depends on the latent heat of the species changing phase. Equation 1.7 describes the change in heat energy for the phase.

The interface temperature is calculated from the assumption that the rate of heat transfer at the interface must equal the latent heat,λ_i consumed at the interface.

$$h_{i\varphi}(T_\varphi - T_f) + h_{i\varphi^{-1}}(T_\varphi^{-1} - T_f) = \dot{m_{\varphi In}} \lambda_i \tag{1.8}$$

As a result of mass transfer occurring in the system, the species transport equation is required to describe concentration of a species within a phase,

$$\frac{\partial \alpha_\varphi C_i}{\partial t} + \nabla \cdot (\alpha_\varphi \bar{\mathbf{U}_\varphi} C_i) - \nabla \alpha_\varphi D_{i\varphi} \nabla (C_i) = \dot{R_i} + \frac{dm_i}{dt} \tag{1.9}$$

where subscript i denotes the species, C_i denotes the concentration of the species in kg/m^3, $D_{i\varphi}$ denotes the mass diffusion coefficient of species i in phase φ, $\dot{R_i}$ denotes the rate of change in species i due to reaction for the phase and $\frac{dm_i}{dt}$ denotes the mass transfer of species i in or out of the phase.

For the two-phase system described in chapter 2, each phase has two components, water and air, therefore, each phase will have two species transport equations. In total there will be four species transport equations. It should be noted that the water species transport equation for both phases will have one value for the mass transfer term, $\frac{dm_i}{dt}$, however, the sign of the term will vary depending on the direction of mass transfer.

The current form of the equation is not valid for compressible flow, this is because the species concentration, C_i is dependent on volume which is evident from the units of kg/m^3. To overcome this, OpenFOAM uses a species concentration, Y_i which is dependent on mass and has units of kg/kg. The two species concentrations are related by the equation below.

$$Y_i = \frac{C_i}{\rho_\varphi} \tag{1.10}$$

The mass diffusion coefficient of the component for the phase, $D_{i\varphi}$, is not computed directly. It is computed through the dimensionless Schmidt number, Sc. The substitution for $D_{i\varphi}$ is outlined below.

$$D_{i\varphi} = \frac{\mu_\varphi}{\rho_\varphi Sc_{i\varphi}} \tag{1.11}$$

The new species transport equation is achieved by multiplying equation 1.9 by $\frac{\rho_\varphi}{\rho_\varphi}$, substituting in equation 1.11 and converting the species concentration to, Y_i, using equation 1.10.

$$\frac{\partial \alpha_\varphi \rho_\varphi Y_i}{\partial t} + \nabla \cdot (\alpha_\varphi \rho_\varphi \bar{\mathbf{U}}_\varphi Y_i) - \nabla \frac{\alpha_\varphi \mu_\varphi}{Sc_{i\varphi}} \nabla(Y_i) = \frac{dm_i}{dt} \tag{1.12}$$

The rate of mass transfer of a component, $\frac{dm_i}{dt}$ is described by the equation below.

$$\frac{dm_i}{dt} = k_{i\varphi} a (C_i^* - C_i) \tag{1.13}$$

Where $k_{i\varphi}$ denotes the mass transfer coefficient of species i in phase φ, a denotes the interfacial area for mass transfer and C_i^* denotes the saturation concentration of species i at the interface of the two phases.

The above equation undergoes a similar manipulation to equation 1.9 and is multiplied by $\frac{\rho_\varphi}{\rho_\varphi}$. The species concentration is then converted to, Y_i, using equation 1.10.

$$\frac{dm_i}{dt} = \rho_\varphi k_{i\varphi} a (Y_i^* - Y_i) \tag{1.14}$$

The saturation concentration C_i^* is dependent on the temperature of the system and can be predicted using saturation models. The saturation model will predict the saturation pressure of a component at the phase interface, p_i^*. This saturation pressure is converted to the saturation concentration using the ideal gas equation.

$$C_i^* = \frac{MW_i}{RT} p_i^* \tag{1.15}$$

The OpenFOAM conversion used to convert saturation pressure to saturation concentration is a modification of the above equation. The density of the phase, ρ_φ is expressed in terms of the ideal gas equation.

$$\rho_\varphi = \frac{MW_\varphi}{RT} p \tag{1.16}$$

Substituting equations 1.15 and 1.16 into equation 1.10 yields the equation OpenFOAM use to convert saturation pressure,p_i^*, to saturation concentration,Y_i^*.

$$Y_i^* = \frac{MW_i}{MW_\varphi} \frac{p_i^*}{p} \tag{1.17}$$

Chapter 2

Tutorial Case

2.1 Introduction

This tutorial describes how to prepare and run a case using the reactingTwoPhaseEulerFoam solver. The method involved in a case set up will be outlined and the case files will be described. The theory behind this solver and the classes employed in it will then be discussed in detail and compared to the code. Finally, modifications to the solver will be developed and implemented.

The reactingTwoPhaseEulerFoam solver uses a two-fluid Eulerian model to describe the system. The Eulerian two-phase system involves two inter-penetrating continua where each phase is its own continuum and is represented by averaged conservation equations. The solver is capable of handling multi-component phases and mass and heat transfer between the two phases.

The tutorial case deals with a laminar bubble column containing air and water. The bubble column geometry will consist of a vertical block filled with water and the remaining head-space will be filled with air. Figure 2.1 shows the phase distribution for the case at time zero, where the blue represents liquid and the red represents gas. Gas consisting of pure air will enter at the bottom of the column, sparge through the water, and exit at the top of the column. The outlet gas will consist of air and water. The gas will enter the column at 350 K and 0.1 m/s. The liquid in the column will also be at 350 K and will have no inflow. The pressure within the column will be 1e5 Pa. Both the liquid and gas in the column are multi-component phases; the components being water and air.

As the simulation progresses, the gas phase will become more saturated with water due to mass transfer as the water changes phase from liquid to gas. This solver contains many phase models to describe the phases; From a basic pure phase model to a more complicated reacting phase model. The solver also allows the user to choose how the phases interact with one another. This can be done by choosing the phase system which will be described later in the tutorial.

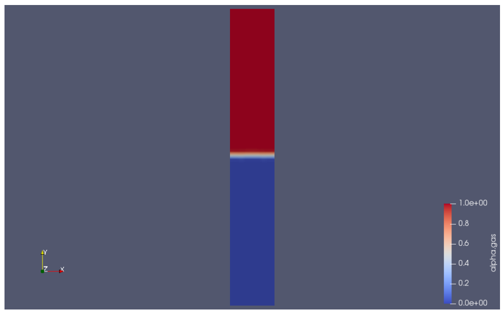

Figure 2.1: Phase fraction of liquid at time 0

2.2 Copy from tutorial directory

Copy the bubbleColumnEvaporatingDissolving tutorial to the run directory.

```
cp -r $FOAM_TUTORIALS/multiphase/reactingTwoPhaseEulerFoam/laminar/
    bubbleColumnEvaporatingDissolving $FOAM_RUN
cd $FOAM_RUN/bubbleColumnEvaporatingDissolving
```

The file structure of the bubbleColumnEvaporatingDissolving case is similar to other OpenFOAM tutorials where the case directory has a `/0`, `/constant` and `/system` directory.

2.3 Boundary and initial conditions

The boundary conditions and initial conditions for the case are set in the `/0` directory. The variables are described in Table 2.1.

Variable Notation	Variable meaning
air.gas	Mass fraction of air in the gas phase
air.liquid	Mass fraction of air in the liquid phase
alpha.gas	phase fraction of gas
alpha.liquid	phase fraction of liquid
p	Pressure
p_rgh	Pressure without hydrostatic pressure
T.gas	Temperature of the gas phase
T.liquid	Temperature of the liquid phase
U.gas	Velocity of gas phase
U.liquid	Velocity of liquid phase
water.gas	Mass fraction of water in the gas phase
water.liquid	Mass fraction of water in the liquid phase

Table 2.1: Boundary condition variables to be set in the /0 directory

2.3.1 Species fractions within phases

As described in the introduction this solver supports multi-component phases. In this case only two components/species are present in each phase however, it would be possible to add more species if required. Air and water are said species. The boundary and initial conditions of the species are set in the /0 directory. The files which define these conditions are `/0/air.gas`, `/0/air.liquid`, `/0/water.gas` and `/0/water.liquid`.

The file `/0/air.gas` which defines the mass fraction of air in the gas phase is shown below.

```
 1 dimensions        [0 0 0 0 0 0 0];
 2
 3 internalField     uniform 1;
 4
 5 boundaryField
 6 {
 7     inlet
 8     {
 9         type            fixedValue;
10         value           $internalField;
11     }
12     outlet
13     {
14         type            inletOutlet;
15         phi             phi.gas;
16         inletValue      $internalField;
17         value           $internalField;
18     }
19     walls
20     {
21         type            zeroGradient;
22     }
23 }
```

In the `/0/air.gas` file, it can be seen that the mass fraction of air in the gas phase is set to 1 throughout the internal field of the column. The mass fraction of air in the gas phase is set to 1 at the inlet boundary condition which means that the inlet gas will be pure air. The outlet boundary condition is of type inletOutlet which is a generic outflow boundary condition for a specified inlet flow. The zero gradient boundary condition is used at the wall. It is evident from line 1 of the above code that the air.gas variable is dimensionless.

The files `/0/air.liquid`, `/0/water.gas` and `/0/water.liquid` follow the same structure as the above file.

2.3.2 Phase fractions

This solver will only work for a two phase system. The boundary and initial conditions of each phase are set in the /0 directory. The files which define these conditions are `/0/alpha.gas` and `/0/alpha.liquid`.

The file `/0/alpha.gas` which defines the gas phase fraction, has a non uniform field which was generated using the setFields utility of OpenFOAM. This will be discussed later in the tutorial. At this stage in the tutorial, it is good practice to delete the old `/0/alpha.gas` file and replace it with the one below.

```
FoamFile
{
    version     2.0;
    format      ascii ;
    class       volScalarField ;
    location    "0";
    object      alpha.gas;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];


internalField   uniform 1;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    outlet
    {
        type            inletOutlet ;
        phi             phi.gas;
        inletValue      uniform 1;
        value           uniform 1;
    }
    walls
    {
        type            zeroGradient;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

The above file has a uniform internal field of 1 for alpha.gas. This can easily be changed later using the setFields utility. The inlet value for alpha.gas is set to 1 which means that only gas is entering the bubble column. The outlet boundary condition is of type inletOutlet. The zero gradient boundary condition is once again used at the wall.

The file `/0/alpha.liquid` for the case will also require replacing with the file below for the same reason as `/0/alpha.gas`.

```
FoamFile
{
    version     2.0;
    format      ascii ;
    class       volScalarField ;
    location    "0";
    object      alpha. liquid ;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];


internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    outlet
    {
        type            inletOutlet ;
        phi             phi. liquid ;
        inletValue      uniform 0;
        value           uniform 0;
    }
    walls
    {
        type            zeroGradient;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

2.3.3 Pressure and hydrostatic pressure

The initial and boundary conditions for the pressure and hydrostaticless pressure are defined in the files /0/p and /0/p_rgh respectively. The file p contains the pressure for the system. This pressure is used for thermodynamic purposes. However, OpenFOAM does not solve for this pressure in the pressure equation. Instead OpenFOAM solves for the hydrstaticless pressure, p_rgh. The thermodynamic pressure, p, is then calculated after p_rgh has been solved using the equation.

$$p = p\_rgh + \rho g h \tag{2.1}$$

The file /0/p is shown below.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions          [1 -1 -2 0 0 0 0];

internalField       uniform 1e5;

boundaryField
{
    inlet
    {
        type                calculated;
        value               $internalField;
    }
    outlet
    {
        type                calculated;
        value               $internalField;
    }
    walls
    {
        type                calculated;
        value               $internalField;
    }
}
```

From the above it is evident that the pressure inside the column is 1e5 Pa. The boundary conditions are calculated to maintain this pressure within the column at initialisation.

The file /0/p_rgh is shown below.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p_rgh;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```

dimensions          [1 -1 -2 0 0 0 0];

internalField       uniform 1e5;

boundaryField
{
    inlet
    {
        type            fixedFluxPressure;
        value           $internalField;
    }
    outlet
    {
        type            prghPressure;
        p               $internalField;
        value           $internalField;
    }
    walls
    {
        type            fixedFluxPressure;
        value           $internalField;
    }
}
```

The inlet pressure is fixed to the internal field value of 1e5 Pa and the outlet pressure is of type prghPressure which will calculate the pressure at the outlet without the hydrostatic effects. This is done through simple manipulation of equation 2.1, yielding the equation below.

$$p\_rgh = p - \rho gh \tag{2.2}$$

2.3.4 Temperature

The initial and boundary conditions for the temperature of the gas and liquid phases are defined in the files `/0/T.gas` and `/0/T.liquid`, respectively. Below is the file `/0/T.gas`.

```
FoamFile
{
    version     2.0;
    format      ascii ;
    class       volScalarField ;
    object      T.gas;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions          [0 0 0 1 0 0 0];

internalField       uniform 350;

boundaryField
{
    walls
    {
        type                zeroGradient;
    }
    outlet
    {
        type                inletOutlet ;
        phi                 phi.gas;
        inletValue          $internalField ;
        value               $internalField ;
    }
    inlet
    {
        type                fixedValue;
        value               $internalField ;
    }
}
```

The gas phase inside the column is at 350 K at time 0. The inlet gas is also at the same temperature as the inlet boundary condition is referencing the internalField temperature. For outlet boundary condition the type inletOutlet is again used along with the zeroGradient boundary condition at the walls. The file structure of `/0/T.liquid` is the same as above.

2.3.5 Velocity

The initial and boundary conditions for the velocity of the gas and liquid phases are defined in the files `/0/U.gas` and `/0/U.liquid` respectively. Below is the file `/0/U.gas`.

```
FoamFile
{
    version     2.0;
    format      binary;
    class       volVectorField;
    object      U.gas;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0.1 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    outlet
    {
        type            pressureInletOutletVelocity;
        phi             phi.gas;
        value           $internalField;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
}
```

From the above code it is evident that the velocity of the gas inside the column and at the inlet is 0.1 m/s and that the direction of velocity is in the positive y-direction. The outlet boundary condition is pressureInletOutletVelocity which will calculate an outlet velocity to satisfy the inlet and internal velocity and pressure and ensure that momentum and mass are conserved. The file `/0/U.liquid` follows the same structure at as above. Verify that the liquid velocity is 0.

2.4 Definition of phase properties

This section will focus on defining the phase system and the phase models for the system. This information is all defined within the file `/constant/phaseProperties`. After defining the phases, it then describes how the phases interact with one another and the models to be used for mass and heat transfer.

2.4.1 Definition of phase system and phase models

```
 1 FoamFile
 2 {
 3     version      2.0;
 4     format       ascii;
 5     class        dictionary;
 6     location     "constant";
 7     object       phaseProperties;
 8 }
 9 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
10 
11 type     interfaceCompositionPhaseChangeTwoPhaseSystem;
12 
13 phases (gas liquid);
14 
15 gas
16 {
17     type            multiComponentPhaseModel;
18     diameterModel   isothermal;
19     isothermalCoeffs
20     {
21         d0              3e−3;
22         p0              1e5;
23     }
24     Sc              0.7;
25 
26     residualAlpha   1e−6;
27 }
28 
29 liquid
30 {
31     type            multiComponentPhaseModel;
32     diameterModel constant;
33     constantCoeffs
34     {
35         d               1e−4;
36     }
37     Sc              0.7;
38 
39     residualAlpha   1e−6;
40 }
```

The above section of the `/constant/phaseProperties` file defines the two phase system and the phase models of each phase. The selection of the two phase system is performed in line 11 of the file. It is evident that the ”type” of two phase system is interfaceCompositionPhaseChangeTwoPhaseSystem. This is one of three available phase system types currently available for the reactingTwoPhaseEulerFoam solver. The three types of two phase systems are described in table 2.2 below.

The phase names are then defined in line 13 of the file. It is evident that in this case those names are gas and liquid, respectively. Next, the phase models for each phase must be defined. The phase models will dictate what the phase continuity equation will be. It can be seen on lines 17 and 31 that both phases in this system are multi-component phase models. The other available phase models which can be used within reactingTwoPhaseEulerFoam are outlined in table 2.3 below.

Lines 18-23 and 32-36 are defining the sauter mean diameter for the gas and liquid phases respectively. The Schmidt numbers for the gas and liquid phases are defined on lines 24 and 37 respectively. The residuals for each phase continuity equation are defined on lines 26 and 39.

Two Phase System	Description
heatAndMomentumTransferTwoPhaseSystem	This system is the most basic system available. It models momentum and heat transfer in a twoPhaseSystem. Drag, virtual mass, lift, wall lubrication and turbulent dispersion are all modelled in this two phase system.
interfaceCompositionPhaseChangeTwoPhaseSystem	This model adds the effect of mass transfer to the basic model. The mass transfer between the phases is calculated according to an interface composition model. The driving force for mass transfer is concentration gradient.
thermalPhaseChangeTwoPhaseSystem	This model adds the effect of mass transfer to the basic model. The mass transfer between the phases is calculated according to a thermal model. The driving force for mass transfer is a temperature gradient. This model should be used for cases involving evaporation and condensation.

Table 2.2: Types of two phase systems available in reactingTwoPhaseEulerFoam

Phase Model	Description
multiComponentPhaseModel	This phase model represents a phase with multiple species.
pureIsothermalPhaseModel	This model represents a pure phase model for which the temperature (strictly energy) remains constant.
purePhaseModel	This model represents a pure phase.
reactingPhaseModel	This model represents a phase with multiple species and volumetric reactions.

Table 2.3: Types of phase models available in reactingTwoPhaseEulerFoam

2.4.2 Blending

Blending is a method used determine the continuous and dispersed phases. As the system becomes more complex with more phases it becomes difficult to differentiate between the continuous and dispersed phases. However, it is also useful for two phase flow. The "FullyContinuous" corresponds to the continuous phase and the "PartlyContinuous" corresponds to the dispersed phase.

```
 1 blending
 2 {
 3     default
 4     {
 5         type                linear ;
 6         minFullyContinuousAlpha.gas 0.7;
 7         minPartlyContinuousAlpha.gas 0.5;
 8         minFullyContinuousAlpha.liquid 0.7;
 9         minPartlyContinuousAlpha.liquid 0.5;
10     }
11 
12     heatTransferModel
13     {
14         type                linear ;
15         minFullyContinuousAlpha.gas 1;
16         minPartlyContinuousAlpha.gas 0;
17         minFullyContinuousAlpha.liquid 1;
18         minPartlyContinuousAlpha.liquid 0;
19     }
20 
21     massTransferModel
22     {
23         type                linear ;
24         minFullyContinuousAlpha.gas 1;
25         minPartlyContinuousAlpha.gas 0;
26         minFullyContinuousAlpha.liquid 1;
27         minPartlyContinuousAlpha.liquid 0;
28     }
29 }
```

The blending factors for the mass transfer and heat transfer models are explicitly described above. Should any other model require a blending coefficient, it will use the coefficients defined in the default blending model (line 3). Linear blending methods are to be used for all models which require it. Other blending methods types include hyperbolic and none.

2.4.3 Interface Composition

The interface composition model will describe how the phases interact at the interface. For the current version of OpenFOAM, only the Saturated model, and Henry model are supported. However, Raoults model and Non Random Two Liquid model exist in the local reactingEulerFoam library, but have not been added to the runtime selectable table as of yet.

```
 1 interfaceComposition
 2 (
 3     (gas in liquid)
 4     {
 5         type Saturated;
 6         species ( water );
 7         Le 1.0;
 8         saturationPressure
 9         {
10             type ArdenBuck;
11         }
12     }
13
14     (liquid in gas)
15     {
16         type Henry;
17         species ( air );
18         k ( 1.492e-2 );
19         Le 1.0;
20     }
21 );
```

The above section of the file `/constant/phaseProperties` describes the interface composition models to be used for a species when it changes phase. The first interface composition model defined will be examined. It is evident from line 5 above that the saturation model used is of type "Saturated" and the species being transported is "water". The saturation pressure is determined using the "ArdenBuck" correlation. Different saturation models can be used to determine the interface composition of a species. These models are outlined in table 2.4 below. The lewis number is the ratio of thermal diffusivity to mass diffusivity and in this case it is given a value of 1.0.

The above section of code defines how much water will transfer from the continuous liquid phase to the dispersed gas phase. The method OpenFOAM uses to evaluate the interface composition will be further examined later in the report.

Saturation Model	Description
Antoine	Uses the Antoine equation to determine vapour pressure in natural log form. The coefficients to be supplied are A, B and C.
Antoine Extended	Uses the Antoine equation to determine vapour pressure in natural log form. The coefficients to be supplied are A, B, C, D and E.
ArdenBuck	Uses the ArdenBuck correlation to determine the vapour pressure. This model is only suitable for a water and air combination.
constant	This model returns a constant saturation temperature and pressure.
function1	This model can only be used to determine a saturation temperature in terms of a saturation pressure.
polynomial	This model can only be used to determine a saturation temperature in terms of a saturation pressure.

Table 2.4: Types of saturation models available in reactingTwoPhaseEulerFoam

2.4.4 Heat transfer

The heat transfer model describes the heat transfer between the phases at the interface. The model calculates the product of the heat transfer coefficient and the surface area. The heat transfer models currently supported by the reactingEulerFoam solvers are the spherical and RanzMarshall models.

```
1  heatTransfer.gas
2  (
3      (gas in liquid)
4      {
5          type spherical;
6          residualAlpha 1e-4;
7      }
8
9      (liquid in gas)
10     {
11         type RanzMarshall;
12         residualAlpha 1e-4;
13     }
14 );
15
16 heatTransfer.liquid
17 (
18     (gas in liquid)
19     {
20         type RanzMarshall;
21         residualAlpha 1e-4;
22     }
23
24     (liquid in gas)
25     {
26         type spherical;
27         residualAlpha 1e-4;
28     }
29 );
```

The heat transfer needs to be defined for both phases. This is evident in lines 1 and 16 of the code.

The first section of code will be evaluated (lines 1 - 14). This defines the heat transfer model for the gas phase. Heat transfer of the gas phase can occur in two different situations. It can occur when the gas is the dispersed phase (line 3) or when the gas is the continuous phase (line 9). The heat transfer model to be used in each situation are defined in lines 5 and 11. The same analysis can be performed on the heat transfer model for the liquid phase.

2.4.5 Mass transfer

The mass transfer model will describe the mass transfer between the phases at the interface. The model calculates the product of the mass transfer coefficient and the surface area. The mass transfer models currently supported by the reactingEulerFoam solvers are the sperical model and the Frossling model.

```
 1 massTransfer.gas
 2 (
 3     (gas in liquid)
 4     {
 5         type spherical;
 6         Le 1.0;
 7     }
 8
 9     (liquid in gas)
10     {
11         type Frossling;
12         Le 1.0;
13     }
14 );
15
16 massTransfer.liquid
17 (
18     (gas in liquid)
19     {
20         type Frossling;
21         Le 1.0;
22     }
23
24     (liquid in gas)
25     {
26         type spherical;
27         Le 1.0;
28     }
29 );
```

The mass transfer within the liquid and gas phases are defined in a similar manner to the heat transfer. That is it is explicitly defined for situations in which each phase is continuous and dispersed.

The mass transfer model for the gas phase will be discussed (lines 1-14). The mass transfer model used to define the gas when it is the dispersed phase (line 3), is the spherical mass transfer model (lines 5). However, when the gas is the continuous phase, the Frossling model (line 11) is used to calculate the mass transfer coefficient. The Lewis number is the ratio of thermal diffusivity to mass diffusivity and is given a value of 1.0 in the mass transfer model. This Le number is used in the calculation of the mass transfer coefficient. The same approach is used to determine the mass transfer model for the liquid phase.

2.4.6 Surface tension and aspect ratio

The surface tension model describes the surface tension at the interface. The solver currently only supports a constant surface tension model. The aspect ratio is refering to the bubble aspect ratio, which will describe the shape of the bubble. A number of aspect ratio models are supported by the reactingEulerFoam solvers. These include, the constant model, the Tomiyama model, the Vakhrushev Efremov model and the Wellek model.

```
 1 surfaceTension
 2 (
 3     (gas and liquid)
 4     {
 5         type            constant;
 6         sigma           0.07;
 7     }
 8 );
 9
10 aspectRatio
11 (
12     (gas in liquid)
13     {
14         type            constant;
15         E0              1.0;
16     }
17
18     (liquid in gas)
19     {
20         type            constant;
21         E0              1.0;
22     }
23 );
```

It should be noted that the surface tension model is valid for a phase pair. It evident from the above file that the phase pair is gas and liquid (line 3). The surface tension model used is of type constant (line 5). This is the only surface tension model supported by reactingTwoPhaseEulerFoam. The value for surface tension is 0.07 N/m. The model used to describe the aspect ratio is of type constant for the situations described on (lines 14 and 20).

2.4.7 Drag and virtual mass

The drag model describes the drag effects the phases impart onto one another. The model calculates the drag coefficients for each phase when it is continuous. The virtual mass model describes the inertial effects the phases have on one another. The model calculates the virtual mass coefficients for each phase.

```
 1 drag
 2 (
 3     (gas in liquid)
 4     {
 5         type            SchillerNaumann;
 6         residualRe      1e-3;
 7         swarmCorrection
 8         {
 9             type        none;
10         }
11     }
12
13     (liquid in gas)
14     {
15         type            SchillerNaumann;
16         residualRe      1e-3;
17         swarmCorrection
18         {
19             type        none;
20         }
21     }
22 );
23
24 virtualMass
25 (
26     (gas in liquid)
27     {
28         type            constantCoefficient;
29         Cvm             0.5;
30     }
31
32     (liquid in gas)
33     {
34         type            constantCoefficient;
35         Cvm             0.5;
36     }
37 );
```

The drag model requires defining when the liquid phase is the continuous phase and the gas phase is dispersed (line 2) and when the phases are reversed (line 13). The drag model is defined on lines 5 and 15. In this case, the SchillerNaumann model is defined for both conditions.

The virtual mass model also requires defining for both phase conditions (lines 26 and 32). The constantCoefficient model is used in this case and the virtual mass coefficient is set to 0.5.

2.5 Definition of thermophysical properties

This section will focus on defining the properties of the species within the individual phases. This information is all defined in the files `/constant/thermophysicalProperties.gas` and `/constant/thermophysicalProperties`. Readers should consult the documentation on thermophysical models in the OpenFOAM user-guide if any further explanation is required.

2.5.1 Thermodynamic properties of the phase

The thermodynamic properties of the gas phase is defined in the first section of the file `/constant/thermophysicalProperties.gas` below.

```
 1 FoamFile
 2 {
 3     version        2.0;
 4     format         ascii ;
 5     class          dictionary;
 6     location       "constant";
 7     object         thermophysicalProperties.gas;
 8 }
 9 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
10 
11 thermoType
12 {
13     type           heRhoThermo;
14     mixture        multiComponentMixture;
15     transport      const;
16     thermo         hConst;
17     equationOfState perfectGas;
18     specie         specie;
19     energy         sensibleInternalEnergy;
20 }
21 
22 dpdt yes;
```

The thermoType model specifies the complete thermophysical model used to describe the system (line 11). It can be seen from (line 19) that the internal energy equation is used for the system. The type model used for the gas phase is heRhoThermo (line 13). This will set up the model calculatioin to be based on internal energy and density. The gas phase is defined as a multiComponentMixture (line 14) with constant transport properties with respect to temperature (line 15). The hConst thermo model is selected in (line 16) which assumes the specific heat capacity of the gas remains constant in the evaluation of the enthalpy of the gas phase. The gas phase is then described as a perfect gas (line 17) and with different species components (line 18). The purpose of (line 22) is to include the pressure term in the energy equation.

2.5.2 Species definition within the phase

The file `/constant/thermophysicalProperties.gas` then continues to define the species components within the gas phase and their properties. It should be noted the variables supplied in this section will depend on the thermoType selections made in the previous subsection 2.5.1.

```
 1 species
 2 (
 3     air
 4     water
 5 );
 6
 7 inertSpecie  air;
 8
 9 "(mixture|air)"
10 {
11     specie
12     {
13         molWeight   28.9;
14     }
15     thermodynamics
16     {
17         Hf          0;
18         Cp          1012.5;
19     }
20     transport
21     {
22         mu          1.84e-05;
23         Pr          0.7;
24     }
25 }
26
27 water
28 {
29     specie
30     {
31         molWeight   18.0153;
32     }
33     thermodynamics
34     {
35         Hf          -1.3435e+07;
36         Cp          1857.8;
37     }
38     transport
39     {
40         mu          1.84e-05;
41         Pr          0.7;
42     }
43 }
```

It is evident from lines (1-5) that the species components of the gas phase are air and water. The species ”air” is then defined as inert (line 7). The mixture is then defined and and the properties of each component in the gas phase is defined. The molecular weight is to be supplied in units of kg/kmol (lines 13 and 31). As a result of selecting the hConst thermo model, the specific heat capacity, c_p, and heat of formation H_f require defining (lines 17, 18, 35 and 36). The units are

J/kg.K and J/kg respectively. It should be noted that condition the heats of formation is evaluated at is 25 C and 1 atm. The enthalpy can then be computed at any temperature using the equation below.

$$Hf(T) = Hf(25\ ^{o}C) + c_p(T - 25) \tag{2.3}$$

Finally the transport variables require defining (lines 22, 23, 40 and 41). The first is the dynamic viscosity, μ, which has units Pa.s. The second is the dimentionless Prandtl number which describes the ratio of momentum diffusivity to thermal diffusivity.

2.6 System set-up

This section will focus on the files in the `/system/` directory. This directory contains dictionaries to control mesh settings and solver settings. The table below outlines the files in this directory.

File name	File description
blockMeshDict	Dictionary which contains instructions on how to generate the mesh structure.
controlDict	Dictionary which contains the run control parameters for the solution procedure.
fvSchemes	Dictionary which contains the discretization schemes used in the solution. This dictionary is run-time selectable.
fvSolution	Dictionary where the equation solvers, tolerances and other algorithm controls are set.
setFieldsDict	Dictionary which contains instructions on generating a non uniform initial conditions in a case.

Table 2.5: Description of files in the system directory

The file `/system/setFieldsDict` will be examined below.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      setFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

defaultFieldValues
(
    volScalarFieldValue alpha.gas 1
    volScalarFieldValue alpha.liquid 0
);

regions
(
    boxToCell
    {
        box (0 0 0) (0.15 0.501 0.1);
        fieldValues
        (
            volScalarFieldValue alpha.gas 0
            volScalarFieldValue alpha.liquid 1
26         );
27     }
28 );
```

The objective of the setFields utility for this tutorial is to add liquid to the column. In section 2.3.2 the boundary and initial conditions were set up to contain no water in the system. The setFieldsDict is informed of this fact in (lines 11 - 15). The utility is then informed that a ”region” (line 17) of type boxToCell (line 19) requires changing. The bottom left and top right points of the rectangular region are then defined (line 21). The new field values within the region are then defined (lines 24 and 25).

2.7 Run the case and visualize in Paraview

The case is nearly ready to run. It is good practice to rename the original 0/ directory before starting with a case because the setFields utility will change the initial conditions.

```
cp −rf 0 0orig
```

The simulation usually takes some time to run completely. In order to decrease execution time of the case the end time of the case should be changed from 100 to 10 and the write interval should be changed from 1 to 0.1. These changes are made in the `system/controlDict` file.

Then generate the mesh using the blockMesh utilty.

```
blockMesh
```

The liquid phase must be added to the initial conditions using the setFields utility.

```
setFields
```

Finally the solver can be run on the case, sending the output to a log file.

```
reactingTwoPhaseEulerFoam >& log&
```

Open the case in paraview.

```
paraFoam
```

It is possible to see the water transferring from the liquid to the gas phase in paraview. To do this select water.gas from the fields tab in the properties section. Click apply. The water.gas field then needs to be selected in the main drop down menu above the viewing window. Press play and watch the mass transfer. Figure 2.2 is a screenshot of the bubble column at the final timestep. The concentration distribution of water in the gas phase across the column is illustrated. Notice how the gas is saturated with water as it passes through the liquid. Figure 2.3 shows the temperature distribution across the column at the final time-step. The gas entering the column is at the same temperature as the liquid in the column, however, there is a clear temperature distribution. This is as a result of water transferring phase. The water is transferring from the liquid to the gas phase which requires energy. The energy expended is in the form of heat energy and heat transfer is clearly happening at the interface between the two phases.

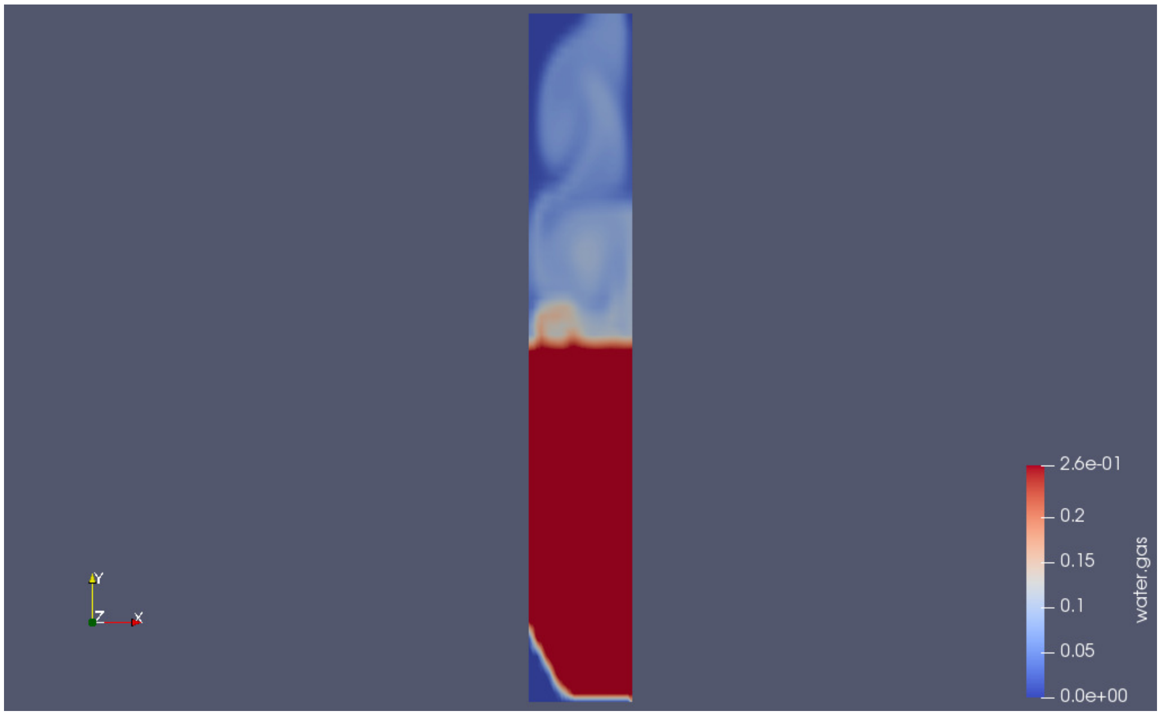

Figure 2.2: Concentration of water in the gas phase at the final timestep

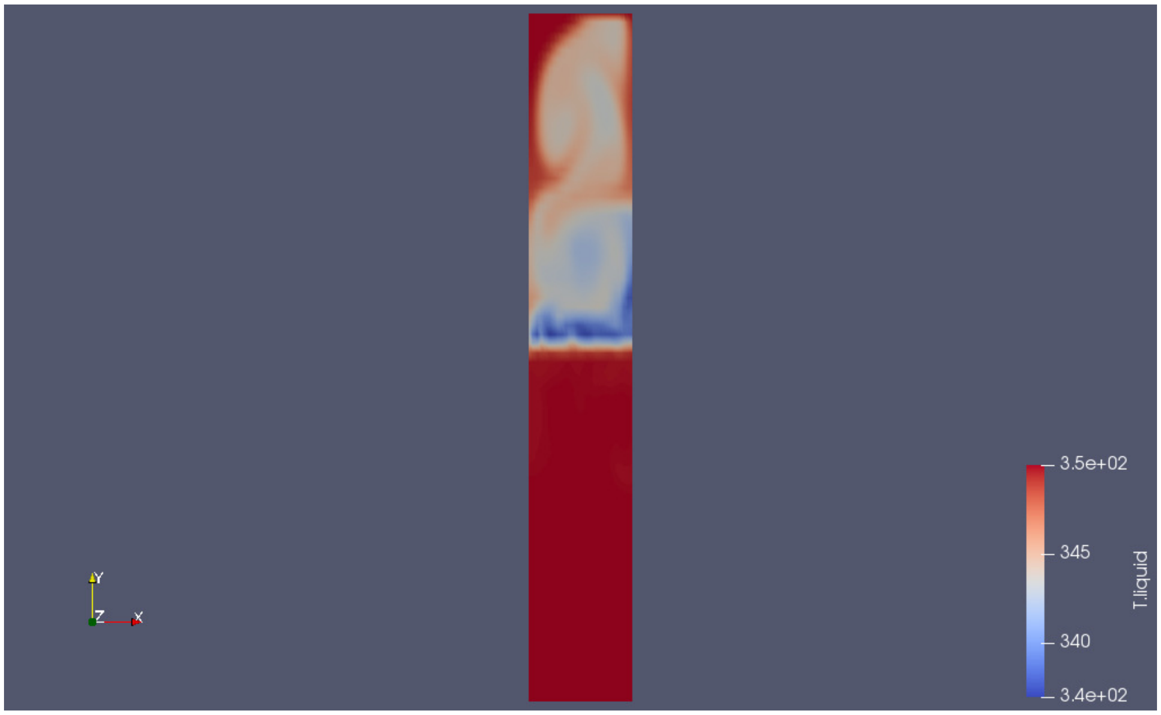

Figure 2.3: Temperature distribution accross column at the final timestep

Chapter 3

Classes

At this point the reader should have an understanding on the general theory employed by the reactingTwoPhaseEulerFoam solver and how to apply the solver to a case. This section will focus on classes which appear in the reactingTwoPhaseEulerFoam solver. The important classes and files for this solver are located in the directory below. All files mentioned in this chapter will be in reference to this directory.
`$FOAM_SOLVERS/multiphase/reactingEulerFoam/reactingTwoPhaseEulerFoam`

The main application file for reactingTwoPhaseEulerFoam is located in
`reactingTwoPhaseEulerFoam.C`.

The application starts by declaring the header files required to perform the finite volume calculations and PIMPLE algorithm.

```
#include ”fvCFD.H”
#include ”twoPhaseSystem.H”
#include ”phaseCompressibleTurbulenceModel.H”
#include ”pimpleControl.H”
#include ”localEulerDdtScheme.H”
#include ”fvcSmooth.H”
```

It is assumed that the reader already has an understanding of finite volume analysis in OpenFOAM and the PIMPLE procedure for two-phase flows. A good explanation of the PIMPLE procedure for two-phase flow and the MULES procedure used to solve the phase continuity equation is provided by Manni (2014). The focus of this report is how OpenFOAM calculates the mass and heat transfer at the interface.

3.1 twoPhaseSystem class

The two phase system class is responsible for the heat and mass transfer at the interface. The header file `twoPhaseSystem.H` is located in the directory `twoPhaseSystem/`.

3.1.1 twoPhaseSystem.H

```
#ifndef twoPhaseSystem_H
#define twoPhaseSystem_H

#include "phaseSystem.H"

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

namespace Foam
{

class dragModel;
class virtualMassModel;
```

The file starts by declaring phaseSystem.H and forward declaring the dragModel and virtualMass-Model classes.

```
class twoPhaseSystem
:
    public phaseSystem
```

The twoPhaseSystem class exhibits inheritance from the phaseSystem class.

```
protected:

    // Protected data

        //- Phase model 1
        phaseModel& phase1_;

        //- Phase model 2
        phaseModel& phase2_;
```

The phases are protected data and are of type phaseModel.

```
    // Selectors

        static autoPtr<twoPhaseSystem> New
        (
            const fvMesh& mesh
        );
```

The selector "New" is declared. This static function is used to create the two-phase system in the case and will be explained in section 3.1.4.

```
{
   public:

    //− Runtime type information
    TypeName(”twoPhaseSystem”);

    // Declare runtime construction

        declareRunTimeSelectionTable
        (
            autoPtr,
            twoPhaseSystem,
            dictionary,
            (
                const fvMesh& mesh
            ),
            (mesh)
        );
```

A runtime construction is created allowing the type of twoPhaseSystem to be selected by the user at runtime in the case set up. The procedure behind the generation of this runtime selection table is outlined in section 3.1.3.

```
 1          //− Return the momentum transfer matrices
 2          virtual autoPtr<momentumTransferTable> momentumTransfer() const = 0;
 3
 4          //− Return the heat transfer matrices
 5          virtual autoPtr<heatTransferTable> heatTransfer() const = 0;
 6
 7          //− Return the mass transfer matrices
 8          virtual autoPtr<massTransferTable> massTransfer() const = 0;
 9
10          //− Return true if there is mass transfer
11          bool transfersMass() const;
12
13          //− Return the interfacial mass flow rate
14          tmp<volScalarField> dmdt() const;
15
16          //− Solve for the phase fractions
17          virtual void solve();
```

Important member functions of the twoPhaseSystem class are declared above. The first three functions momentumTransfer, heatTransfer and MassTransfer (lines 2, 5 and 8) are virtual functions which will be called by reactingTwoPhaseEulerFoam and return pointers to matrices. They appear in the momentum, heat and mass conservation equations respectively. On (line 11) a boolean switch called transferMass is declared which is used to determine if mass transfer will occur in the system. The solve function declared on (line 17) is used to solve the continuity equation. It will be examined in section 3.1.2.

3.1.2 twoPhaseSystem.C

```
 1 #include ”twoPhaseSystem.H”
 2 #include ”dragModel.H”
 3 #include ”virtualMassModel.H”
 4
 5 #include ”MULES.H”
 6 #include ”subCycle.H”
 7
 8 #include ”fvcDdt.H”
 9 #include ”fvcDiv.H”
10 #include ”fvcSnGrad.H”
11 #include ”fvcFlux.H”
12 #include ”fvcSup.H”
13
14 #include ”fvmDdt.H”
15 #include ”fvmLaplacian.H”
16 #include ”fvmSup.H”
```

The header files to be used in `twoPhaseSystem.C` are listed above. The drag and virtualMass models were declared in section 3.1.1 and their header files are now included in `twoPhaseSystem.C` (lines 2 and 3). It is evident that the MULES procedure will be employed in this file (line 5) and that finite volume calculations will be performed (lines 8 - 16).

```
 1 Foam::twoPhaseSystem::twoPhaseSystem
 2 (
 3     const fvMesh& mesh
 4 )
 5 :
 6     phaseSystem(mesh),
 7     phase1_(phaseModels_[0]),
 8     phase2_(phaseModels_[1])
 9 {
10     phase2_.volScalarField :: operator=(scalar(1) − phase1_);
11
12     volScalarField& alpha1 = phase1_;
13     mesh.setFluxRequired(alpha1.name());
14 }
```

The twoPhaseSystem constructor is then defined. It is evident from (line 3) that the two-phase system takes the object mesh as an argument. phaseSystem is the parent class which is initialized with the argument mesh (line 6). The object phase1 is initialized with the first argument of the phaseModels object (line 7) and phase2 is initialized with the second argument of the phaseModels object (line 8). It is evident from (line 10) that phase2 is calculated using the equation.

$$\alpha_2 = 1 - \alpha_1 \tag{3.1}$$

```
1 void Foam::twoPhaseSystem::solve()
2 {
3     const Time& runTime = mesh_.time();
4
5     volScalarField& alpha1 = phase1_;
6     volScalarField& alpha2 = phase2_;
```

The solve function is defined in `twoPhaseSystem.C` and is used to solve the phase continuity equation 1.4. It is a convoluted function which will not be analysed in detail. The OF course report of Manni (2014), should be consulted for a detailed description of how the solver employs the MULES algorithm to solve the phase continuity equation.

```
            MULES::explicitSolve
            (
                geometricOneField(),
                alpha1,
                phi_,
                alphaPhi1,
                Sp,
                Su,
                phase1_.alphaMax(),
                0
            );
```

It is evident from the above section of the `solve` function, that the MULES algorithm is employed to solve for α_1.

```
        Info<< alpha1.name() << " volume fraction = "
            << alpha1.weightedAverage(mesh_.V()).value()
            << " Min(alpha1) = " << min(alpha1).value()
            << " Max(alpha1) = " << max(alpha1).value()
            << endl;

        // Ensure the phase-fractions are bounded
        alpha1.maxMin(0, 1);

        // Update the phase-fraction of the other phase
        alpha2 = scalar(1) - alpha1;
    }
}
```

The `solve` function ends by outputting the values calculated for α_1. Boundedness of α_1 is then verified and finally α_2 is calculated using equation 3.1.

3.1.3 twoPhaseSystems.C

```
1 #include ”addToRunTimeSelectionTable.H”
2
3 #include ”phaseSystem.H”
4 #include ”twoPhaseSystem.H”
5 #include ”MomentumTransferPhaseSystem.H”
6 #include ”HeatTransferPhaseSystem.H”
7 #include ”InterfaceCompositionPhaseChangePhaseSystem.H”
8 #include ”ThermalPhaseChangePhaseSystem.H”
```

The declaration files of the two-phase systems available are on lines 5-8. It is evident that the phase systems described are MomentumTransferPhaseSystem, HeatTransferPhaseSystem, InterfaceCompositionPhaseChangePhaseSystem and ThermalPhaseChangePhaseSystem.

```
namespace Foam
{
    typedef
        HeatTransferPhaseSystem
        <
            MomentumTransferPhaseSystem<twoPhaseSystem>
        >
        heatAndMomentumTransferTwoPhaseSystem;

    addNamedToRunTimeSelectionTable
    (
        twoPhaseSystem,
        heatAndMomentumTransferTwoPhaseSystem,
        dictionary,
        heatAndMomentumTransferTwoPhaseSystem
    );

    typedef
        InterfaceCompositionPhaseChangePhaseSystem
        <
            MomentumTransferPhaseSystem<twoPhaseSystem>
        >
        interfaceCompositionPhaseChangeTwoPhaseSystem;

    addNamedToRunTimeSelectionTable
    (
        twoPhaseSystem,
        interfaceCompositionPhaseChangeTwoPhaseSystem,
        dictionary,
        interfaceCompositionPhaseChangeTwoPhaseSystem
    );

    typedef
        ThermalPhaseChangePhaseSystem
        <
            MomentumTransferPhaseSystem<twoPhaseSystem>
        >
        thermalPhaseChangeTwoPhaseSystem;

    addNamedToRunTimeSelectionTable
41     (
42         twoPhaseSystem,
43         thermalPhaseChangeTwoPhaseSystem,
44         dictionary,
45         thermalPhaseChangeTwoPhaseSystem
46     );
47 }
```

The phase systems require input arguments and are layered in a manner which is not user friendly. Type definitions are used to simplify the process of inputting a two-phase system at runtime.

The interfaceCompositionPhaseChangeTwoPhaseSystem typedef will be analysed. This is typedef in which the twoPhaseSystem class is an argument of the MomentumTransferPhaseSystem (line 21). This is then passed as an argument to the InterfaceCompositionPhaseChangePhaseSystem (line 19). The two-phase system created is then passed to a runtime selection table for the twoPhase system (lines 25 - 31).

It should be noted that the MomentumTransferPhaseSystem is present in all the two-phase systems in reactingTwoPhaseEulerFoam. This is because it is the system which solves the momentum equations. Note that the two-phase systems created in the above file are the ones described in table 2.2.

3.1.4 newTwoPhaseSystem.C

The file below contains the definition of static function used to select the two-phase system.

```
 1 #include "twoPhaseSystem.H"
 2 
 3 // * * * * * * * * * * * * * * * * Selector  * * * * * * * * * * * * * * * * //
 4 
 5 Foam::autoPtr<Foam::twoPhaseSystem> Foam::twoPhaseSystem::New
 6 (
 7     const fvMesh& mesh
 8 )
 9 {
10     const word systemType
11     (
12         IOdictionary
13         (
14             IOobject
15             (
16                 propertiesName,
17                 mesh.time().constant(),
18                 mesh,
19                 IOobject::MUST_READ_IF_MODIFIED,
20                 IOobject::NO_WRITE,
21                 false
22             )
23         ).lookup("type")
24     );
25     Info<< "Selecting twoPhaseSystem " << systemType << endl;
26 
27     auto cstrIter = dictionaryConstructorTablePtr_->cfind(systemType);
28 
29     if (!cstrIter.found())
30     {
31         FatalErrorInFunction
32             << "Unknown twoPhaseSystem type "
33             << systemType << nl << nl
34             << "Valid twoPhaseSystem types :" << endl
35             << dictionaryConstructorTablePtr_->sortedToc()
36             << exit(FatalError);
37     }
38 
39     return cstrIter()(mesh);
40 }
```

The header file `twoPhaseSystem.H` is declared on line 1 because this is where the declaration of the "New" selector is located. It can be seen that the mesh argument is passed to the function on line 7 of the file. The two phase system is then read from the propertiesName dictionary (line 16). The type of two-phase system is found by searching for word "type" in the dictionary (line 23). The system defined in the propertiesName dictionary is assigned the variable name "systemType" (line 10).
The function then searches for "systemType" from the runtime selection table created in section 3.1.3. If the "systemType" is not in the table it returns the error described in (lines 31-36). Otherwise the systemType is returned with the object mesh as an argument (line 30).

3.2 phaseModel class

The phases in the two-phase system were objects of the class phaseModel. The files which construct the phaseModel class are located in the directory `../phaseSystems/phaseModel/phaseModel/`. Note that the directory `phaseSystems` is located in the directory `reactingEulerFoam`. This directory contains classes which are templated and are also used in the `reactingMultiphaseEulerFoam` solver. The classes are created in such a way that they work for an unlimited number of phases. This class will contain the momentum and species fraction equations derived in chapter 1.

3.2.1 phaseModel.H

```
#include ”dictionary.H”
#include ”dimensionedScalar.H”
#include ”volFields.H”
#include ”surfaceFields.H”
#include ”fvMatricesFwd.H”
#include ”rhoThermo.H”
#include ”phaseCompressibleTurbulenceModelFwd.H”
#include ”runTimeSelectionTables.H”
```

The header files to be used by the phaseModel class are declared. Notice the rhoThermo.H header file.

```
namespace Foam
{

class phaseSystem;
class diameterModel;
```

The forward declaration of phaseSystem and diameterModel classes.

```
1  class phaseModel
2  :
3      public volScalarField
4  {
5      // Private data
6
7          //- Reference to the phaseSystem to which this phase belongs
8          const phaseSystem& fluid_;
9
10         //- Name of phase
11         word name_;
12
13         //- Index of phase
14         label index_;
```

The phaseModel class is inherited from the volScalarField class (line 3). An object called fluid is declared and it is of the class phaseSystem. The name of the phase is also declared (line 11) and is given an index (line 14).

```
    // Declare runtime construction

        declareRunTimeSelectionTable
        (
            autoPtr,
            phaseModel,
            phaseSystem,
            (
                const phaseSystem& fluid,
                const word& phaseName,
                const label index
            ),
            (fluid , phaseName, index)
        );
```

A runtime construction is declared allowing the user to select the phase model for each phase in the case set up. The procedure behind the generation of this runtime selection table is outlined in section 3.2.3.

```
            static autoPtr<phaseModel> New
        (
            const phaseSystem& fluid,
            const word& phaseName,
            const label index
        );
```

The selector ”New” is declared. This static function is used to create the phaseModels in the case and is explained in section 3.2.4.

```
        //- Return the momentum equation
        virtual tmp<fvVectorMatrix> UEqn() = 0;

        //- Return the enthalpy equation
        virtual tmp<fvScalarMatrix> heEqn() = 0;

        //- Return the species fraction equation
        virtual tmp<fvScalarMatrix> YiEqn(volScalarField& Yi) = 0;
```

The following virtual member functions refer to the momentum, enthalpy and species fraction equations without momentum transfer, heat transfer and mass transfer respectively between the phases.

3.2.2 phaseModel.C

```
 1  Foam::phaseModel::phaseModel
 2 (
 3     const phaseSystem& fluid,
 4     const word& phaseName,
 5     const label index
 6 )
 7 :
 8     volScalarField
 9     (
10         IOobject
11         (
12             IOobject::groupName("alpha", phaseName),
13             fluid .mesh().time().timeName(),
14             fluid .mesh(),
15             IOobject::READ_IF_PRESENT,
16             IOobject::AUTO_WRITE
17         ),
18         fluid .mesh(),
19         dimensionedScalar(dimless, Zero)
20     ),
21 
22     fluid_ ( fluid ),
23     name_(phaseName),
24     index_(index),
```

The constructor of the class phaseModel is defined in the file `phaseModel.C`. The arguments fluid, phaseName and index are passed to the phaseModel class during construction (lines 3-5). The parent class volVectorVield is then constructed using "alpha.phaseName" as the scalar (line 12). The class member data, fluid, name and index are then initialized with the arguments passed to the class during construction. (lines 22-24).

3.2.3 phaseModels.C

```
1  #include ”addToRunTimeSelectionTable.H”
2
3  #include ”rhoThermo.H”
4  #include ”rhoReactionThermo.H”
5
6  #include ”CombustionModel.H”
7
8  #include ”phaseModel.H”
9  #include ”ThermoPhaseModel.H”
10 #include ”IsothermalPhaseModel.H”
11 #include ”AnisothermalPhaseModel.H”
12 #include ”PurePhaseModel.H”
13 #include ”MultiComponentPhaseModel.H”
14 #include ”InertPhaseModel.H”
15 #include ”ReactingPhaseModel.H”
16 #include ”MovingPhaseModel.H”
```

The declaration files of partial phase models are available are on lines (9-16). rhoThermo.H, rhoReactionThermo.H and CombustionModel.H are the declaration files of classes which describe the thermodynamics of the individual phase. The classes declared above are combined in such a way to generate a complete phase model.

```
namespace Foam
{
    typedef
        AnisothermalPhaseModel
        <
            PurePhaseModel
            <
                InertPhaseModel
                <
                    MovingPhaseModel
                    <
                        ThermoPhaseModel<phaseModel, rhoThermo>
                    >
                >
            >
        >
        purePhaseModel;

    addNamedToRunTimeSelectionTable
    (
        phaseModel,
        purePhaseModel,
        phaseSystem,
        purePhaseModel
    );
    typedef
        IsothermalPhaseModel
        <
            PurePhaseModel
            <
                InertPhaseModel
                <
                    MovingPhaseModel
                    <
                        ThermoPhaseModel<phaseModel, rhoThermo>
                    >
                >
            >
        >
        pureIsothermalPhaseModel;

    addNamedToRunTimeSelectionTable
    (
        phaseModel,
        pureIsothermalPhaseModel,
        phaseSystem,
        pureIsothermalPhaseModel
    );

    typedef
        AnisothermalPhaseModel
        <
            MultiComponentPhaseModel
            <
                InertPhaseModel
                <
                    MovingPhaseModel
                    <
                        ThermoPhaseModel<phaseModel, rhoReactionThermo>
                    >
                >
            >
        >
        multiComponentPhaseModel;

    addNamedToRunTimeSelectionTable
    (
        phaseModel,
        multiComponentPhaseModel,
        phaseSystem,
        multiComponentPhaseModel
    );
    typedef
        AnisothermalPhaseModel
        <
            MultiComponentPhaseModel
            <
                ReactingPhaseModel
                <
                    MovingPhaseModel
                    <
                        ThermoPhaseModel<phaseModel, rhoReactionThermo>
                    >,
                    CombustionModel<rhoReactionThermo>
                >
```

```
86            >
87        >
88        reactingPhaseModel;
89
90    addNamedToRunTimeSelectionTable
91    (
92        phaseModel,
93        reactingPhaseModel,
94        phaseSystem,
95        reactingPhaseModel
96    );
97 }
```

Similiarly to the twoPhaseSystems, the phaseModels require type definitions to simplify the process of inputting a phase model at runtime.

The reactingPhaseModel typedef will be analysed as it is the most complicated(line 88). This is the typedef in which the ThermoPhaseModel has phaseModel, and rhoReactionThermo as the input arguments. This model is then used as the input argument to the MovingPhaseModel. This model is then input as one of the two argument to the ReactingPhaseModel. The other argument required for the ReactingPhaseModel is the CombustionModel. The reactingPhaseModel is then used as the input argument to the MultiComponentPhaseModel. Finally this model is the input argument to the AnisothermalPhaseModel. The name given to this convoluted typedef is reactingPhaseModel which is then added to the runtime selection table (line 90).

Note that the phase model typedefs created in the above file are the ones described in table 2.3.

3.2.4 newPhaseModel.C

```
1  Foam::autoPtr<Foam::phaseModel> Foam::phaseModel::New
2  (
3      const phaseSystem& fluid,
4      const word& phaseName,
5      const label index
6  )
7  {
8      const word modelType(fluid.subDict(phaseName).lookup(”type”));
9
10     Info<< ”Selecting phaseModel for ”
11         << phaseName << ”: ” << modelType << endl;
12
13     auto cstrIter = phaseSystemConstructorTablePtr_->cfind(modelType);
14
15     if (! cstrIter .found())
16     {
17         FatalErrorInFunction
18             << ”Unknown phaseModel type ”
19             << modelType << nl << nl
20             << ”Valid phaseModel types :” << endl
21             << phaseSystemConstructorTablePtr_->sortedToc()
22             << exit(FatalError);
23     }
24
25     return cstrIter ()( fluid , phaseName, index);
26 }
```

The phase model selector ”New” takes three arguments, fluid which is an phaseSystem object (line 3), phaseName (line 4) and index (line 5). The phaseModel is selected by searching the word, ”type” in the subdictionary of the phase system. This is then assigned the variable name modelType (line 8). The function then searches for the modelType in the runtime selectable table created in section 3.2.3. If the modelType is not found the function will output and error (lines 17-22), otherwise the function will return the phaseModel with the fluid, phaseName and index arguments passed to it.

3.3 InterfaceCompositionPhaseChangePhaseSystem

This class is responsible for calculating the amount of mass transfer occuring in the system. The interface temperature will also vary as a result. The functions employed to determine the mass transfer and interface temperatures are massTransfer and correctThermo respectively.

3.3.1 InterfaceCompositionPhaseChangePhaseSystem.H

```
1  #include "HeatAndMassTransferPhaseSystem.H"
2
3  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
4
5  namespace Foam
6  {
7
8  class interfaceCompositionModel;
9
10 /*---------------------------------------------------------------------------*\
11                 Class InterfaceCompositionPhaseChangePhaseSystem Declaration
12 \*---------------------------------------------------------------------------*/
13
14 template<class BasePhaseSystem>
15 class InterfaceCompositionPhaseChangePhaseSystem
16 :
17     public HeatAndMassTransferPhaseSystem<BasePhaseSystem>
18 {
```

It is evident from the declaration of the interfaceCompositionPhaseChangePhaseSystem class that the class exhibits inheritance from the HeatAndMassTransferPhaseSystem class (line 17). There is a forward declaration of the class interfaceCompositionModel on line 8. This class contains all the models to evaluate the properties at the interface.

```
1  protected:
2
3      // Protected typedefs
4
5          typedef HashTable
6          <
7              autoPtr<interfaceCompositionModel>,
8              phasePairKey,
9              phasePairKey::hash
10         > interfaceCompositionModelTable;
11
12
13     // Protected data
14
15         // Sub Models
16
17             //- Interface composition models
18             interfaceCompositionModelTable interfaceCompositionModels_;
```

A hash table typedef called interfaceCompositionModelTable (line 10) is then created for the interface composition models. The object interfaceCompositionModels_ is then created from the typedef class interfaceCompositionModelTable (line 18). This object will contain all the interfaceComposition models for the system.

3.3.2 InterfaceCompositionPhaseChangePhaseSystem.C

```
 1 #include ”InterfaceCompositionPhaseChangePhaseSystem.H”
 2 #include ”interfaceCompositionModel.H”
 3
 4
 5 // * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
 6
 7 template<class BasePhaseSystem>
 8 Foam::InterfaceCompositionPhaseChangePhaseSystem<BasePhaseSystem>::
 9 InterfaceCompositionPhaseChangePhaseSystem
10 (
11     const fvMesh& mesh
12 )
13 :
14     HeatAndMassTransferPhaseSystem<BasePhaseSystem>(mesh)
15 {
16     this->generatePairsAndSubModels
17     (
18         ”interfaceComposition”,
19         interfaceCompositionModels_
20     );
21 }
```

This file opens up with two declaration files. Those are the declaration file for InterfaceCompositionPhaseChangePhaseSystem (line 1) and the declaration file for interfaceCompositionModels (line 2).

The constructor for the InterfaceCompositionPhaseChangePhaseSystem is then defined. This class is templated for a ”BasePhaseSystem” (line 7), which is amtwoPhaseSystem for this report. The class requires the mesh object as an argument. The parent class, HeatAndMassTransferPhaseSystem is also construced with argument mesh (line 14). The interfaceCompositionModels_ object is initialised using the generatePairsAndSubModels function. This function is declared and defined in `phaseSystems/phaseSystem/phaseSystem.H`.

Two important functions are located in this file. This first function is massTransfer() which will be described in 3.3.2, and the second is correctThermo() which will be described in 3.3.2.

MassTransfer Function

```
 1 template<class BasePhaseSystem>
 2 Foam::autoPtr<Foam::phaseSystem::massTransferTable>
 3 Foam::InterfaceCompositionPhaseChangePhaseSystem<BasePhaseSystem>::
 4 massTransfer() const
 5 {
 6     // Create a mass transfer matrix for each species of each phase
 7     auto eqnsPtr = autoPtr<phaseSystem::massTransferTable>::New();
 8     auto& eqns = *eqnsPtr;
 9 
10     for (const phaseModel& phase : this->phaseModels_)
11     {
12         const PtrList<volScalarField>& Yi = phase.Y();
13 
14         forAll(Yi, i)
15         {
16             eqns.set
17             (
18                 Yi[i].name(),
19                 new fvScalarMatrix(Yi[i], dimMass/dimTime)
20             );
21         }
22     }
```

The above section of the mass transfer function creates a new mass transfer matrix for each species of each phase. The "New" function belonging to the class autoPtr creates an empty mass transfer matrix. The class autoPtr uses the template massTransferTable which is a typedef of a HashPtrTable defined in the file `/phaseSystems/phaseSystem/phaseSystem.H`. The pointer value is then assigned to the variable eqns.

The for loop (line 10) is where the species for each phase is generated. The function will look through all of the phaseModels in the phase object. For this case there are two phases in the phase object. The first line in the for loop (line 12) determines which species are in current phase in the loop and stores them in the volScalarField `Yi`. The nested forAll loop will then create a new volume scalar matrix within the eqns matrix for each component in the phase. The species mass fraction is defined as the unknown quantity in the matrix (line 18). This matrix represents the mass transfer of this component in the phase. This is evident because of units used on line 19.

```
 1     // Sum up the contribution from each interface composition model
 2     forAllConstIters
 3     (
 4         interfaceCompositionModels_,
 5         interfaceCompositionModelIter
 6     )
 7     {
 8         const phasePair& pair =
 9             *(this->phasePairs_[interfaceCompositionModelIter.key()]);
10 
11         const interfaceCompositionModel& compositionModel =
12             *(interfaceCompositionModelIter.object());
13 
14         const phaseModel& phase = pair.phase1();
15         const phaseModel& otherPhase = pair.phase2();
```

```
16        const phasePairKey key(phase.name(), otherPhase.name());
17
18        const volScalarField& Tf(*this->Tf_[key]);
19
20        volScalarField& dmdtExplicit(*this->dmdtExplicit_[key]);
21        volScalarField& dmdt(*this->dmdt_[key]);
22
23        scalar  dmdtSign(Pair<word>::compare(this->dmdt_.find(key).key(), key));
24
25        const volScalarField  K
26        (
27            this->massTransferModels_[key][phase.name()]->K()
28        );
```

The effect of mass transfer is then evaluated for every composition model in the system. In the case of the two phase system described in section 2. The interface composition models were ”Saturated” for when the gas was present in the liquid, and ”Henry” for when the liquid was present in the gas (line 1). The Saturation model will be explored.

Line 18 defines the interface temperature, which is crucial information as mass transfer is temperature dependent. dmdtExplicit refers to the explicit part of the mass transfer term and dmdt refers to the implicit part of the term (lines 20 and 21 respectively). dmdtSign defines the direction of mass transfer for the phase pair.

The mass transfer coefficient is defined on lines 25-28. The mass transfer model used is defined at runtime in the phaseProperties file, see section 2.4.5. The method OpenFOAM uses to determine K is outlined in Prasad (2016).

It should be noted that the ”key” referred to in this file is the key assigned to the phase pair in the hash table. For cases with multiple phases this phase pair key makes it easier to choose and assign variables.

```
1            forAllConstIter
2         (
3             hashedWordList,
4             compositionModel.species(),
5             memberIter
6         )
7         {
8             const word& member = *memberIter;
9
10            const word name
11            (
12                IOobject::groupName(member, phase.name())
13            );
14
15            const word otherName
16            (
17                IOobject::groupName(member, otherPhase.name())
18            );
19
20            const volScalarField  KD
21            (
22                K*compositionModel.D(member)
23             );
24 
25             const volScalarField Yf
26             (
27                 compositionModel.Yf(member, Tf)
28             );
```

The second forAllConstIter loop is nested within the interfaceCompositionModel forAllConstIter loop. This loop will evaluate the effect of mass transfer for every compositionModel for each species involved. The KD object created on line 20 is calculated by multiplying the mass transfer coefficient defined in the previous section by the mass diffusivity, D for the component in the phase. The function to determine the mass diffusivity is located in
`interfacialCompositionModels/interfaceCompositionModels/InterfaceCompositionModels/`
`.../InterfaceCompositionModel.C`

The species saturation concentration, `Y_f` is defined on line 25. The Yf function of the compositionModel object is used on line 27. This value is very important in the calculation of the mass transfer for the system as it is the primary driving force for phase change. This function will be discussed in section 3.5.

```
 1             // Implicit transport through the phase
 2             *eqns[name] +=
 3                 phase.rho()*KD*Yf
 4               - fvm::Sp(phase.rho()*KD, eqns[name]->psi());
 5 
 6             // Sum the mass transfer rate
 7             dmdtExplicit += dmdtSign*phase.rho()*KD*Yf;
 8             dmdt -= dmdtSign*phase.rho()*KD*eqns[name]->psi();
 9 
10             // Explicit transport out of the other phase
11             if (eqns.found(otherName))
12             {
13                 *eqns[otherName] -=
14                     otherPhase.rho()*KD*compositionModel.dY(member, Tf);
15             }
16         }
17     }
18 
19     return eqnsPtr;
20 }
```

This section of the code is where the mass transfer term for the component in a phase is calculated. Multiplying out equation 1.14 yields the following equation

$$\frac{dm_i}{dt} = \rho_\varphi k_{i\varphi} a Y_i^* - \rho_\varphi k_{i\varphi} a Y_i \tag{3.2}$$

Where $k_{i\varphi}a$ is KD in OpenFOAM.

This is the equation shown on lines 2-4. The function `fvm::Sp` on line 4 is used to make the source term implicit. The function $->psi()$ is a reference to an unknown field. Line 4 is therefore, making an implicit source term based on the unknown variable in the eqns object, which will be Y_i. The massTransfer function ends by returning the eqnsPtr which is a pointer to a location which contains all the mass transfer terms for each component in each phase.

correctThermo Function

```
1  template<class BasePhaseSystem>
2  void Foam::InterfaceCompositionPhaseChangePhaseSystem<BasePhaseSystem>::
3  correctThermo()
4  {
5      BasePhaseSystem::correctThermo();
6  
7      // This loop solves for the interface temperatures, Tf, and updates the
8      // interface composition models.
9      //
10     // The rate of heat transfer to the interface must equal the latent heat
11     // consumed at the interface, i.e.:
12     //
13     // H1*(T1 - Tf) + H2*(T2 - Tf) == mDotL
14     //                             == K*rho*(Yfi - Yi)*Li
15     //
16     // Yfi is likely to be a strong non-linear (typically exponential) function
17     // of Tf, so the solution for the temperature is newton-accelerated
18 
19     forAllConstIters(this->phasePairs_, phasePairIter)
20     {
21         const phasePair& pair = *(phasePairIter.object());
22 
23         if (pair.ordered())
24         {
25             continue;
26         }
```

The idea behind the correctThermo function is described in the comments of the function. The mass transfer occurring is resulting in a phase change which will result in a latent heat. The above section describes how the interface temperature will be affected by the latent heat. The forAllConstIter loop will solve for the interface temperature between all phase pairs in the system (line 19).

```
        const phasePairKey key12(pair.first(), pair.second(), true);
        const phasePairKey key21(pair.second(), pair.first(), true);

        volScalarField H1(this->heatTransferModels_[pair][pair.first()]->K());
        volScalarField H2(this->heatTransferModels_[pair][pair.second()]->K());
        dimensionedScalar HSmall("small", heatTransferModel::dimK, SMALL);

        volScalarField mDotL
        (
            IOobject
            (
                "mDotL",
                this->mesh().time().timeName(),
                this->mesh()
            ),
            this->mesh(),
            dimensionedScalar(dimEnergy/dimVolume/dimTime, Zero)
        );
        volScalarField mDotLPrime
        (
21                IOobject
22                (
23                    "mDotLPrime",
24                    this->mesh().time().timeName(),
25                    this->mesh()
26                ),
27                this->mesh(),
28                dimensionedScalar(mDotL.dimensions()/dimTemperature, Zero)
29            );
30
31            volScalarField& Tf = *this->Tf_[pair];
```

The objects key12 and key21 (lines 1 and 2) are used to define the direction of heat transfer. key 12 denotes transfer from phase 1 to phase 2. The objects H1 and H2 (lines 4 and 5) are the heat transfer coefficients for phase 1 and 2 respectively. HSmall (line 6) is an object used to ensure that there is no division by zero. The mDotL object (line 8) energy released by latent heat due to mass transfer. It is the term on the right hand side of equation 1.8. The mDotLPrime object denotes the derivative of mDotL with respect to interface temperature, T_f. This term is employed in newtons method to calculate T_f.

```
        // Add latent heats from forward and backward models
        if (this->interfaceCompositionModels_.found(key12))
        {
            this->interfaceCompositionModels_[key12]->addMDotL
            (
                this->massTransferModels_[pair][pair.first()]->K(),
                Tf,
                mDotL,
                mDotLPrime
            );
        }
        if (this->interfaceCompositionModels_.found(key21))
        {
            this->interfaceCompositionModels_[key21]->addMDotL
            (
                this->massTransferModels_[pair][pair.second()]->K(),
                Tf,
                mDotL,
                mDotLPrime
            );
        }
```

The addMDotL object is used if there is more than one species transferring phase. This variable will sum up the individual mDotL values and store them.

```
        // Update the interface temperature by applying one step of newton's
        // method to the interface relation
        Tf -=
            (
                H1*(Tf - pair.phase1().thermo().T())
              + H2*(Tf - pair.phase2().thermo().T())
              + mDotL
            )
           /(
                max(H1 + H2 + mDotLPrime, HSmall)
            );
```

```
12
13         Tf.correctBoundaryConditions();
14
15         Info<< "Tf." << pair.name()
16             << ": min = " << min(Tf.primitiveField())
17             << ", mean = " << average(Tf.primitiveField())
18             << ", max = " << max(Tf.primitiveField())
19             << endl;
20
21         // Update the interface compositions
22         if (this->interfaceCompositionModels_.found(key12))
23         {
24             this->interfaceCompositionModels_[key12]->update(Tf);
25         }
26         if (this->interfaceCompositionModels_.found(key21))
27         {
28             this->interfaceCompositionModels_[key21]->update(Tf);
29         }
30     }
31 }
```

The interface temperature is then calculated using one step of newtons method on equation 1.8. The boundary conditions are corrected based on the new Tf value (line 13). The value for Tf is then outputted from the solver (line 15).

3.4 HeatAndMassTransferPhaseSystem

This section will discuss the HeatAndMassTransferPhaseSystem class. This class describes how the mass transfer which was calculated in section 3.3.2 affects the momentum and energy equations.

3.4.1 HeatAndMassTransferPhaseSystem.H

```
1  #ifndef HeatAndMassTransferPhaseSystem_H
2  #define HeatAndMassTransferPhaseSystem_H
3
4  #include "phaseSystem.H"
5
6  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
7
8  namespace Foam
9  {
10
11 template<class modelType>
12 class BlendedInterfacialModel;
13
14 class blendingMethod;
15 class heatTransferModel;
16 class massTransferModel;
17
18 /*---------------------------------------------------------------------------
19                    Class HeatAndMassTransferPhaseSystem Declaration
20 \*---------------------------------------------------------------------------
21
22 template<class BasePhaseSystem>
23 class HeatAndMassTransferPhaseSystem
24 :
25     public BasePhaseSystem
26 {
```

The declaration file begins by declaring the phaseSystem header file and declaring the classes of BlendedInterfacialModel, blendingMethod, heatTransferModel and massTransferModel. It is evident from line 25 that the HeatAndMassTransferPhaseSystem exhibits inheritance from the BasePhaseSystem (line 25) which is the templated class (line 22).

```
protected:

    // Protected typedefs

        typedef HashTable
        <
            HashTable
            <
                autoPtr<BlendedInterfacialModel<heatTransferModel>>
            >,
            phasePairKey,
            phasePairKey::hash
        > heatTransferModelTable;
```

```
14
15        typedef HashTable
16        <
17            HashTable
18            <
19                autoPtr<BlendedInterfacialModel<massTransferModel>>
20            >,
21            phasePairKey,
22            phasePairKey::hash
23        > massTransferModelTable;
24
25
26    // Protected data
27
28        //- Mass transfer rate
29        HashPtrTable<volScalarField, phasePairKey, phasePairKey::hash>
30            dmdt_;
31
32        //- Explicit part of the mass transfer rate
33        HashPtrTable<volScalarField, phasePairKey, phasePairKey::hash>
34            dmdtExplicit_;
35
36        //- Interface temperatures
37        HashPtrTable<volScalarField, phasePairKey, phasePairKey::hash> Tf_;
38
39        // Sub Models
40
41            //- Heat transfer models
42            heatTransferModelTable heatTransferModels_;
43
44            //- Mass transfer models
45            massTransferModelTable massTransferModels_;
```

The heat transfer and mass transfer hash tables for the phase pairs are then defined and assigned the type definition of heatTransferModelTable and massTransferModelTable respectively (lines 5-23). The heat transfer and mass transfer model objects are then of declared from the typedefs (lines 42 and 45). The objects of Tf, dmdt and dmdtExplicit are declared as hash pointer tables (lines 29 - 37).

```
    // Member Functions

        //- Return true if there is mass transfer for phase
        virtual bool transfersMass(const phaseModel& phase) const;

        //- Return the interfacial mass flow rate
        virtual tmp<volScalarField> dmdt(const phasePairKey& key) const;

        //- Return the total interfacial mass transfer rate for phase
        virtual tmp<volScalarField> dmdt(const phaseModel& phase) const;

        //- Return the momentum transfer matrices
        virtual autoPtr<phaseSystem::momentumTransferTable>
            momentumTransfer() const;

        //- Return the heat transfer matrices
        virtual  autoPtr<phaseSystem::heatTransferTable>
            heatTransfer() const;

        //− Return the mass transfer matrices
        virtual  autoPtr<phaseSystem::massTransferTable>
            massTransfer() const = 0;

        //− Correct the thermodynamics
        virtual  void correctThermo() = 0;

        //− Read base phaseProperties dictionary
        virtual  bool read();
```

Important member functions are then decalared. Note that the massTransfer and correctThermo functions were described in section 3.3.2. The momentumTransfer and heatTransfer functions are responsible for calculating the effect of mass transfer on the momentum and heat equations and are described in section 3.4.2.

3.4.2 HeatAndMassTransferPhaseSystem.C

```
#include "HeatAndMassTransferPhaseSystem.H"

#include "BlendedInterfacialModel.H"
#include "heatTransferModel.H"
#include "massTransferModel.H"

#include "HashPtrTable.H"

#include "fvcDiv.H"
#include "fvmSup.H"
#include "fvMatrix.H"
#include "zeroGradientFvPatchFields.H"

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

template<class BasePhaseSystem>
Foam::HeatAndMassTransferPhaseSystem<BasePhaseSystem>::
HeatAndMassTransferPhaseSystem
(
    const fvMesh& mesh
)
:
    BasePhaseSystem(mesh)
{
    this−>generatePairsAndSubModels
    (
        "heatTransfer",
        heatTransferModels_
    );

    this−>generatePairsAndSubModels
    (
        "massTransfer",
34          massTransferModels_
35      );
```

The constructor for the HeatAndMassTransferPhaseSystem starts by initializing the parent BasePhaseSystem class (line 23). The heat transfer and mass transfer models are then generated for the system on lines 25 and 31.

```
    forAllConstIters(this->phasePairs_, phasePairIter)
    {
        const phasePair& pair = *(phasePairIter.object());

        if (pair.ordered())
        {
            continue;
        }

        // Initially assume no mass transfer

        dmdt_.set
        (
            pair,
            new volScalarField
            (
                IOobject
                (
                    IOobject::groupName(”dmdt”, pair.name()),
                    this->mesh().time().timeName(),
                    this->mesh(),
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
                ),
                this->mesh(),
                dimensionedScalar(dimDensity/dimTime, Zero)
            )
        );

        dmdtExplicit_.set
        (
            pair,
            new volScalarField
            (
                IOobject
                (
                    IOobject::groupName(”dmdtExplicit”, pair.name()),
                    this->mesh().time().timeName(),
                    this->mesh()
                ),
                this->mesh(),
                dimensionedScalar(dimDensity/dimTime, Zero)
            )
        );

        volScalarField H1(heatTransferModels_[pair][pair.first()]->K());
        volScalarField H2(heatTransferModels_[pair][pair.second()]->K());
```

```
48
49          Tf_.set
50          (
51              pair,
52              new volScalarField
53              (
54                  IOobject
55                  (
56                      IOobject::groupName("Tf", pair.name()),
57                      this->mesh().time().timeName(),
58                      this->mesh(),
59                      IOobject::NO_READ,
60                      IOobject::AUTO_WRITE
61                  ),
62                  (
63                      H1*pair.phase1().thermo().T()
64                    + H2*pair.phase2().thermo().T()
65                  )
66                 /max
67                  (
68                      H1 + H2,
69                      dimensionedScalar("small", heatTransferModel::dimK, SMALL)
70                  ),
71                  zeroGradientFvPatchScalarField::typeName
72              )
73          );
74          Tf_[pair]->correctBoundaryConditions();
75      }
76  }
```

The dmdt, dmdtExplicit and Tf objects are initialized on lines 12, 30 and 49 respectively. dmdt and dmdtExplicit are initially set to zero and are updated by the massTransfer function described in section 3.3.2. The heat transfer coefficients, H_φ are then computed by calling the K function of the heatTransferModels object for each phase (lines 46 and 47). The interface temperature, Tf, is then initialized by manipulating the convection equation to isolate Tf, yielding the equation below.

$$T_f = \frac{H_\varphi T_\varphi + H_{\varphi^{-1}} T_{\varphi^{-1}}}{H_\varphi + H_{\varphi^{-1}}} \tag{3.3}$$

Where φ denotes the phase, and φ^{-1} denotes the other phase.

Momentum Transfer Function

```
template<class BasePhaseSystem>
Foam::autoPtr<Foam::phaseSystem::momentumTransferTable>
Foam::HeatAndMassTransferPhaseSystem<BasePhaseSystem>::momentumTransfer() const
{
    autoPtr<phaseSystem::momentumTransferTable>
        eqnsPtr(BasePhaseSystem::momentumTransfer());

    phaseSystem::momentumTransferTable& eqns = eqnsPtr();

    // Source term due to mass transfer
    forAllConstIters(this->phasePairs_, phasePairIter)
    {
```

```
13         const phasePair& pair = *(phasePairIter.object());
14
15         if (pair.ordered())
16         {
17             continue;
18         }
19
20         const volVectorField& U1(pair.phase1().U());
21         const volVectorField& U2(pair.phase2().U());
22
23         const volScalarField dmdt(this->dmdt(pair));
24         const volScalarField dmdt21(posPart(dmdt));
25         const volScalarField dmdt12(negPart(dmdt));
26
27         *eqns[pair.phase1().name()] += dmdt21*U2 - fvm::Sp(dmdt21, U1);
28         *eqns[pair.phase2().name()] -= dmdt12*U1 - fvm::Sp(dmdt12, U2);
29     }
30
31     return eqnsPtr;
32 }
```

The momentum transfer function is responsible for calculating the momentum transferred between the phases as a result of mass transfer. The function starts by initializing a pointer to a momentum transfer table (line 6). The reference object, eqns, is then created by referencing the pointer (line 8). Lines 20 and 21 create the velocity objects and lines 24 and 25 create the objects which contain the direction of mass transfer. It should be noted that for dmdt21, the mass transfer direction is from phase two to phase one and dmdt12 is the reciprocal. The momentum change as a result of mass transfer is then calculated in lines 27 and 28 and stored in the eqns object for the phases. It is evident that the the solver is employing equation 1.3 to evaluate the mass transfer effect on momentum.

Heat Transfer Function

```
1  template<class BasePhaseSystem>
2  Foam::autoPtr<Foam::phaseSystem::heatTransferTable>
3  Foam::HeatAndMassTransferPhaseSystem<BasePhaseSystem>::heatTransfer() const
4  {
5      auto eqnsPtr = autoPtr<phaseSystem::heatTransferTable>::New();
6      auto& eqns = *eqnsPtr;
7
8      for (const phaseModel& phase : this->phaseModels_)
9      {
10         eqns.set
11         (
12             phase.name(),
13             new fvScalarMatrix(phase.thermo().he(), dimEnergy/dimTime)
14         );
15     }
```

The heat transfer function is responsible for calculating the energy transferred between the phases as a result of mass transfer. Mass transfer results in phase change which will change the temperature at the interface. Therefore, energy will transfer between phases in two different ways, the first is due to a temperature difference, the second is due to kinetic energy contained in the mass being transferred. The heat transfer function begins by initializing a new pointer to a heat transfer table (line 3). A reference object, eqns, is then created by referencing the pointer.

```
1  // Heat transfer with the interface
2  forAllConstIters(heatTransferModels_, heatTransferModelIter)
3  {
4      const phasePair& pair =
5          *(this->phasePairs_[heatTransferModelIter.key()]);
6
7      const phaseModel* phase = &pair.phase1();
8      const phaseModel* otherPhase = &pair.phase2();
9
10     const volScalarField& Tf(*Tf_[pair]);
11
12     const volScalarField K1
13     (
14         heatTransferModelIter()[pair. first ()]->K()
15     );
16     const volScalarField K2
17     (
18         heatTransferModelIter()[pair.second()]->K()
19     );
20     const volScalarField KEff
21     (
22         K1*K2
23        /max
24         (
25             K1 + K2,
26             dimensionedScalar("small", heatTransferModel::dimK, SMALL)
27         )
28     );
29
30     const volScalarField* K = &K1;
31     const volScalarField* otherK = &K2;
32
33             forAllConstIter(phasePair, pair, iter)
34     {
35         const volScalarField& he(phase->thermo().he());
36         volScalarField Cpv(phase->thermo().Cpv());
37
38         *eqns[phase->name()] +=
39             (*K)*(Tf - phase->thermo().T())
40           + KEff/Cpv*he - fvm::Sp(KEff/Cpv, he);
41
42         Swap(phase, otherPhase);
43         Swap(K, otherK);
44     }
45 }
```

The phases and interface temperature are defined on lines 7-10. The heat transfer coefficients, K1 and K2 are then defined by calling the K function of the heat transfer model. The energy transferred as a result of temperature difference is then calculated on lines 39-40 and added to the heat transfer table for the phase. The equation can be compared to equation 1.7. The terms of the equation on line 40 are used to increase diagonal dominance in the energy equation to help convergence.

```
1  // Source term due to mass transfer
2  forAllConstIters(this->phasePairs_, phasePairIter)
```

```
 3     {
 4         const phasePair& pair = *(phasePairIter.object());
 5
 6         if (pair.ordered())
 7         {
 8             continue;
 9         }
10
11         const phaseModel& phase1 = pair.phase1();
12         const phaseModel& phase2 = pair.phase2();
13
14         const volScalarField& he1(phase1.thermo().he());
15         const volScalarField& he2(phase2.thermo().he());
16
17         const volScalarField& K1(phase1.K());
18         const volScalarField& K2(phase2.K());
19
20         const volScalarField dmdt(this->dmdt(pair));
21         const volScalarField dmdt21(posPart(dmdt));
22         const volScalarField dmdt12(negPart(dmdt));
23         const volScalarField& Tf(*Tf_[pair]);
24
25         *eqns[phase1.name()] +=
26             dmdt21*(phase1.thermo().he(phase1.thermo().p(), Tf))
27           - fvm::Sp(dmdt21, he1)
28           + dmdt21*(K2 - K1);
29
30         *eqns[phase2.name()] -=
31             dmdt12*(phase2.thermo().he(phase2.thermo().p(), Tf))
32           - fvm::Sp(dmdt12, he2)
33           + dmdt12*(K1 - K2);
34     }
35
36     return eqnsPtr;
37 }
```

The phase, enthalpy and kinetic energy are defined on lines 11-18. The direction of mass transfer is then defined on lines 21 and 22. Note that for dmdt21, the mass transfer direction is from phase two to phase one and dmdt12 is the opposite. The kinetic energy change as a result of mass transfer is then calculated and added to the heat transfer table for each phase. The equation can be compared to equation 1.6. The first two terms of the equation on lines 26, 27, 31 and 32 are used to increase diagonal dominance in the energy equation and help convergence.

3.5 Saturation Model Class

Saturation models employed in reactingTwoPhaseEulerFoam are used to determine saturation concentrations or temperatures at an interface. The saturated model is an interface composition model which assumes the rate of mass transfer is based on a concentration difference between the saturated concentration at the interface and the actual concentration. A saturation model is then applied which will determine the saturation pressure at the interface. This pressure is then converted to a concentration and used to calculcate mass transfer. The files for the Saturated class are located in the directory `interfacialCompositionModels/interfaceCompositionModels/Saturated/`.

Saturated.H

```
#include "InterfaceCompositionModel.H"
#include "saturationModel.H"

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

namespace Foam
{

class phasePair;

namespace interfaceCompositionModels
{
protected:

    // Private data

        //- Saturated species name
        word saturatedName_;

        //- Saturated species index
        label saturatedIndex_;

        //- Saturation pressure model
        autoPtr<saturationModel> saturationModel_;
```

The file begins with the declaration files for IterfaceCompositionModel and saturationModel. The class phasePair is forward declared and a namespace "interfaceCompositionModels" is created. The object `saturationModel_` is a pointer of the class saturationModel.

Saturated.C

```
#include "Saturated.H"

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

template<class Thermo, class OtherThermo>
Foam::tmp<Foam::volScalarField>
Foam::interfaceCompositionModels::Saturated<Thermo, OtherThermo>::wRatioByP() const
{
    const dimensionedScalar Wi
    (
        "W",
12          dimMass/dimMoles,
13          this->thermo_.composition().W(saturatedIndex_)
14      );
15
16      return Wi/this->thermo_.W()/this->thermo_.p();
```

The member function wRatioByP is the conversion factor used to convert saturation pressure to saturation concentration. This conversion was described in section 1 and can be seen in equation 1.17. Wi is the moleculat weight of the species (line 10). The `thermo.W()` function returns the molecular weight of the species. The `this->` pointer is refering to the current phase. Therefore `this->thermo.W()` is referring to the molecular weight of the phase.

```
template<class Thermo, class OtherThermo>
Foam::tmp<Foam::volScalarField>
Foam::interfaceCompositionModels::Saturated<Thermo, OtherThermo>::Yf
(
    const word& speciesName,
    const volScalarField& Tf
) const
{
    if (saturatedName_ == speciesName)
    {
        return wRatioByP()*saturationModel_->pSat(Tf);
    }
```

The Yf function determines the saturation concentration at the interface and is an important variable in determining mass transfer. It is evident from the above section of the file that `Yf` is determined by converting a saturation pressure to a saturation concentration. The pSat function belongs to the saturationModel class which is chosen at runtime.

3.5.1 Arden Buck saturation model

This section will describe the Arden Buck saturation model which was the saturation model selected for the tutorial case in section 2. The directory containing the saturation models in reactingTwoPhaseEulerFoam is `interfacialCompositionModels/saturationModels/`. Descriptions of the saturation models can be found in table 2.4. The Arden Buck equation is used to correlate the saturated moisture concentration of air to temperature.

$$P_s(T) = 611.21exp((18.678 - \frac{T}{234.5})(\frac{T}{257.14 + T})) \tag{3.4}$$

Where $P_s(T)$ is the saturation pressure in Pa and T is the temperature in ^oC

ArdenBuck.C

```
1  namespace Foam
2  {
3  namespace saturationModels
4  {
5      defineTypeNameAndDebug(ArdenBuck, 0);
6      addToRunTimeSelectionTable(saturationModel, ArdenBuck, dictionary);
7  }
8  }
9
10 static const Foam::dimensionedScalar zeroC("", Foam::dimTemperature, 273.15);
11 static const Foam::dimensionedScalar A("", Foam::dimPressure, 611.21);
12 static const Foam::dimensionedScalar B("", Foam::dimless, 18.678);
13 static const Foam::dimensionedScalar C("", Foam::dimTemperature, 234.5);
14 static const Foam::dimensionedScalar D("", Foam::dimTemperature, 257.14);
```

The Arden Buck equation is added as a run time selectable on line 6 of the above section of the file. Lines 10 to 14 then define the objects to be used in the ArdenBuck class.

```
Foam::tmp<Foam::volScalarField>
Foam::saturationModels::ArdenBuck::pSat
(
    const volScalarField& T
) const
{
    volScalarField TC(T - zeroC);

    return A*exp(TC*xByTC(TC));
}
```

The pSat function of the ArdenBuck class is shown above. Inputting the variables into the above equation yields equation 3.4. The pSat function is then employed by the Yf function to determine the saturation concentration at the interface as described in section 3.5.

Chapter 4

Algorithm

This section will focus on how the ReactingTwoPhaseEulerFoam solver employs the classes described in chapter 3 to solve the twoPhaseSystem equations.

4.1 createFields.H

```
Info<< "Creating phaseSystem\n" << endl;

autoPtr<twoPhaseSystem> fluidPtr
(
    twoPhaseSystem::New(mesh)
);
twoPhaseSystem& fluid = fluidPtr();
```

The twoPhaseSystem is constructed and is assigned the name fluid. The fluid object for the tutorial case described in section 2 will have be a two phase system of type interfaceCompositionPhaseChangeTwoPhaseSystem. The individual phase models for the phase system are of type multiComponentPhaseModel.

```
dimensionedScalar pMin
(
    "pMin",
    dimPressure,
    fluid
);

#include "gh.H"

volScalarField& p = fluid.phase1().thermo().p();

Info<< "Reading field p_rgh\n" << endl;
volScalarField p_rgh
(
    IOobject
    (
        "p_rgh",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

The file then creates the pressure and hydrostaticless pressure and gravity terms.

It should be noted that the object ”fluid” now contains all the information about the two-phase system and its individual phases. The reactingTwoPhaseEulerFoam solver is very convoluted and the classes are layered on top of one another. This complicates navigation through the solver files. The table below contains a list of the locations of important objects used to solve the twoPhaseSystem for the case described in chapter 2. The file locations are given from the directory
`$FOAM_SOLVERS/multiphase/reactingEulerFoam`

Object	Location
UEqn()	phaseSystems/phaseModel/MovingPhaseModel/MovingPhaseModel.C
heEqn()	phaseSystems/phaseModel/AnisothermalPhaseModel/AnisothermalPhaseModel.C
YiEqn(Yi)	phaseSystems/phaseModel/MultiComponentPhaseModel/MultiComponentPhaseModel.C
momentumTransfer()	phaseSystems/PhaseSystems/MomentumTransferPhaseSystem/..
	../MomentumTransferPhaseSystem.C
heatTransfer()	phaseSystems/PhaseSystems/HeatAndMassTransferPhaseSystem/..
	../HeatAndMassTransferPhaseSystem.C
massTransfer()	phaseSystems/PhaseSystems/InterfaceCompositionPhaseChangePhaseSystem/..
	../InterfaceCompositionPhaseChangePhaseSystem.C

Table 4.1: The definition locations of important Variables in the interfaceCompositionPhaseChangeTwoPhaseSystem and multiComponentPhaseModel classes

4.2 createFieldRefs.H

The following file creates variables from the fluid object created in section 4.1.

```
 1 phaseModel& phase1 = fluid.phase1();
 2 phaseModel& phase2 = fluid.phase2();
 3
 4 volScalarField& alpha1 = phase1;
 5 volScalarField& alpha2 = phase2;
 6
 7 volVectorField& U1 = phase1.U();
 8 surfaceScalarField& phi1 = phase1.phi();
 9 surfaceScalarField& alphaPhi1 = phase1.alphaPhi();
10 surfaceScalarField& alphaRhoPhi1 = phase1.alphaRhoPhi();
11
12 volVectorField& U2 = phase2.U();
13 surfaceScalarField& phi2 = phase2.phi();
14 surfaceScalarField& alphaPhi2 = phase2.alphaPhi();
15 surfaceScalarField& alphaRhoPhi2 = phase2.alphaRhoPhi();
16 surfaceScalarField& phi = fluid.phi();
17
18 rhoThermo& thermo1 = phase1.thermo();
19 rhoThermo& thermo2 = phase2.thermo();
20
21 volScalarField& rho1 = thermo1.rho();
22 const volScalarField& psi1 = thermo1.psi();
23
24 volScalarField& rho2 = thermo2.rho();
25 const volScalarField& psi2 = thermo2.psi();
26
27 const IOMRFZoneList& MRF = fluid.MRF();
28 fv::options& fvOptions = fluid.fvOptions();
```

The phase models are assigned on lines 1 and 2. The phase fractions are assigned on lines 4 and 5. The phase velocities and fluxes are assigned on lines 7 - 16. The thermodynamic models are assigned on lines 18 and 19. The density and compressibilty are assigned on lines 21 - 25. Note that all of the above information was located within the fluid object and that this file is assigning variables data to the make the information easier to access.

4.3 reactingTwoPhaseEulerFoam.C

The reactingTwoPhaseEulerFoam file starts by creating the objects required to solve the finite volume calculations and the including declaration file for the two-phase system. See chapter 4 introduction.

```
1  int main(int argc, char *argv[])
2  {
3      #include ”postProcess.H”
4
5      #include ”addCheckCaseOptions.H”
6      #include ”setRootCase.H”
7      #include ”createTime.H”
8      #include ”createMesh.H”
9      #include ”createControl.H”
10     #include ”createTimeControls.H”
11     #include ”createFields.H”
12     #include ”createFieldRefs.H”
13
14     if (!LTS)
15     {
16         #include ”CourantNo.H”
17         #include ”setInitialDeltaT.H”
18     }
19
20     bool faceMomentum
21     (
22         pimple.dict().lookupOrDefault(”faceMomentum”, false)
23     );
24
25     bool implicitPhasePressure
26     (
27         mesh.solverDict(alpha1.name()).lookupOrDefault
28         (
29             ”implicitPhasePressure”, false
30         )
31     );
```

The application starts on line 1. Important declaration files which set up the case, post processing utilities and the time object are included on lines 3-12. Line 14 checks if local time stepping is occuring and will include files if it is set to not occur.

Line 20 is a boolean which checks if faceMomentum is occurring. faceMomentum can be set at runtime. Weller stated in the OpenFOAM-Dev release notes on github that ”face momentum” is beneficial for the Euler-Euler multiphase equations because it allows all forces to be treated in a consistent manner at the cell faces which will increase the calculation stability, consistancy and accuracy. However, the momentum transport terms must also be interpolated to the cell-faces which will reduce the accuracy of the momentum transport terms. The user must decide if accuracy of the momentum transport terms are more important than the accuracy of the force balance. The face momentum is switched off by default (line 22).

An option is available to solve for the phase 1 pressure implicitly should it be required. (line 25).

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

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

    int nEnergyCorrectors
    (
        pimple.dict().lookupOrDefault<int>("nEnergyCorrectors", 1)
    );

    if (LTS)
    {
        #include "setRDeltaT.H"
        if (faceMomentum)
        {
            #include "setRDeltaTf.H"
        }
    }
    else
    {
        #include "CourantNos.H"
        #include "setDeltaT.H"
    }

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

This section is all about setting up the time step and starting the time loop in the algorithm.

```
  // --- Pressure-velocity PIMPLE corrector loop
    while (pimple.loop())
    {
        fluid.solve();
        fluid.correct();

        #include "YEqns.H"

        if (faceMomentum)
        {
            #include "pUf/UEqns.H"
            #include "EEqns.H"
            #include "pUf/pEqn.H"
            #include "pUf/DDtU.H"
        }
        else
        {
            #include "pU/UEqns.H"
            #include "EEqns.H"
            #include "pU/pEqn.H"
        }

        fluid.correctKinematics();
```

```
24
25              if (pimple.turbCorr())
26              {
27                  fluid .correctTurbulence();
28              }
29          }
30
31          runTime.write();
32
33          runTime.printExecutionTime(Info);
34      }
35
36      Info<< "End\n" << endl;
```

The pressure-veloctity PIMPLE corrector loop is set up in this section of the file (line 2). Within the loop the first equation to be solved is the phase continuity equation 1.4. Line 4 calls the solve function of the twoPhaseSystem object called fluid. This function is defined in section 3.1.2 and employs the MULES algorithm. Line 5 then calls the correct function of the fluid object. The correct function is inherited from the phaseSystem class. Its function is to correct the fluid properties other than the thermo and turbulence for the system.

Line 7 should not be confused as a header file. The `YEqns.H` file contains the section of the application which solves the species fraction equations within each phase and calculates the mass transfer in the system, see equation 1.9. This file will be further explained in section 4.4.

The files containing the forces and momentum equation will vary depending on whether facemomentum is selected. The case where face momentum is not selected will be examined. Line 18 then calls the file `pU/UEqns.H`. This file solves the momentum equations described in equation 1.1 without the effects of pressure, turbulent drag and the explicit part of the drag term. The momentum transfer function described in section 3.4.2 is called in this file. This function calculates the momentum change in the phase as a result of mass transfer.

Line 19 calls the file `pU/EEqns.H`. This file solves the energy equation for the system and includes the heat transfer function described in section 3.4.2 and the correct thermo function described in section 3.3.2. The heatTransfer function calculates the effect of mass transfer on the energy equation. The change in energy in the phase is due to a kinetic energy difference and temperature difference. The correctThermo function calculates the new temperature at the phase interface after mass transfer has occurred.

Line 20 calls the file `pU/pEqn.H`. This file solves the pressure equation of the system and the terms omitted from `pU/UEqns.H` are included here. This is where the pressure-velocity linking occurs in the system which is pivotal for the PIMPLE algorithm. `Rusche(2002)` should be consulted for more information on the theory behind the pressure-velocity linking in the PISO algorithm.

The kinematics of the fluid are then corrected on line 23 and the turbulence properties are corrected on line 27. The file ends by writing the specified data to runtime (line 31), and printing the time elapsed to the terminal window (line 33).

4.4 YEqns.H

```
autoPtr<phaseSystem::massTransferTable>
    massTransferPtr(fluid.massTransfer());

phaseSystem::massTransferTable&
    massTransfer(massTransferPtr());

PtrList<volScalarField>& Y1 = phase1.Y();
PtrList<volScalarField>& Y2 = phase2.Y();
```

The file starts by defining a pointer to a mass transfer table called massTransferPtr. The pointer is then initialized using the massTransfer function of the fluid object which is described in section 3.3.2. The object massTransfer is then initialised as a reference to the pointer. This process of calling a function to return a pointer and then using a variable to reference the pointer is used bacause the mass transfer table generated by the mass transfer function is large. It would slow the application down to copy the object so in the interest of execution time the pointers are outputted from the function.

Y1 refers to the species in phase 1 and Y2 refers to the species in phase 2. For the tutorial case described section 2, both Y1 and Y2 will have two species, gas and liquid.

```
 1  forAll(Y1, i)
 2  {
 3      tmp<fvScalarMatrix> Y1iEqn(phase1.YiEqn(Y1[i]));
 4
 5      if (Y1iEqn.valid())
 6      {
 7          Y1iEqn =
 8          (
 9              Y1iEqn
10           ==
11              *massTransfer[Y1[i].name()]
12            + fvOptions(alpha1, rho1, Y1[i])
13          );
14
15          Y1iEqn->relax();
16          Y1iEqn->solve(mesh.solver("Yi"));
17      }
18  }
```

The application continues and executes a for loop which will loop through all of the species in phase 1. An fvScalarMatrix called Y1iEqn is then initialized with the species conservation equation for the species in the current iteration of the for loop (line 3). YiEqn function is defined in the phase model of each phase. Since both phases in the case were multiComponentPhaseModels, the YiEqn function is defined in the file `phaseSystems/phaseModel/MultiComponentPhaseModelMultiComponentPhaseModel.C`. See section 4.4.1. Y1[i] is referring to the species in the current iteration of the for loop.

The YiEqn is then rewritten to include the mass transfer term in the species conservation equation. The species equation now matches equation 1.12. The equation is relaxed and then solved for the species concentration. The procedure will loop for every component in phase 1. The procedure then occurs for phase 2.

4.4.1 MultiComponentPhaseModel.C

```
template<class BasePhaseModel>
Foam::tmp<Foam::fvScalarMatrix>
Foam::MultiComponentPhaseModel<BasePhaseModel>::YiEqn
(
    volScalarField& Yi
)
{

return
    (
        fvm::ddt(alpha, rho, Yi)
      + fvm::div(alphaRhoPhi, Yi, "div(" + alphaRhoPhi.name() + ",Yi)")
      - fvm::Sp(this->continuityError(), Yi)

      - fvm::laplacian
        (
            fvc::interpolate(alpha)
           *fvc::interpolate(this->turbulence().muEff()/Sc_),
            Yi
        )
```

The above equation is identical to the left hand side of equation 1.12.

Chapter 5

Case Modification

This chapter will look at using methanol as an alternative to water. In order for this to happen changes will have to be made to the following files in the case set up.

File name
0/water.gas
0/water.liquid
0/T.gas
0/T.liquid
constant/phaseProperties
constant/thermophysicalProperties.gas
constant/thermophysicalProperties.liquid

Table 5.1: Case files requiring changing

The only composition model in the system which requires changing is the saturation model as it is only valid for humid air. This model will be changed to the Antoine saturation model. However, the current Antoine saturation model available in reactingTwoPhaseEulerFoam relates pressure to temperature via an exponential scale. For the antoine coefficients obtained for methanol, a power scale is required. Therefore a new saturation model must be created.Before proceeding the reactingEulerFoam directory must be copied to the user directory.

The current Antoine saturation model available in reactingEulerFoam is outlined below.

$$\log(P_s(T)) = A + \frac{B}{C+T} \tag{5.1}$$

Where $P_s(T)$ is the saturation pressure in bar, T is the temperature in K and A, B and C are the species coefficients which have units of dimensionless, K and K, respectively.

The new Antoine Equation is shown below.

$$\log_{10}(P_s(T)) = A - \frac{B}{C+T} \tag{5.2}$$

Manipulations and calculus are performed on the above equations to explicitly describe different variables. Member functions are called which calculate and return the variables.

5.1 Antoine Equation Manipulations

Isolating the saturation pressure, P_s, in equation 5.2 yields the equation below. This equation is used to calculate the saturation concentration in the manner described in section 3.5.

$$P_s = 10^{A-\frac{B}{C+T}} \tag{5.3}$$

Equation 5.3 is then differentiated with respect to temperature and yields the equation below,

$$\frac{dP_s}{dT} = \frac{\log 10 * B}{(C+T)^2} P_s \tag{5.4}$$

Equation 5.2 is once again manipulated and the saturation temperature, T_s, is isolated. This term is used as part of the driving force for mass transfer in the thermal phase change model.

$$T_s = \frac{B}{A - \log_{10} P} - C \tag{5.5}$$

5.2 Copying solver to the user directory

```
foam
cp −r −−parents applications/solvers/multiphase/reactingEulerFoam/
    $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/applications/solvers/multiphase/reactingEulerFoam
rm −rf reactingMultiphaseEulerFoam
mv reactingTwoPhaseEulerFoam/ myReactingTwoPhaseEulerFoam/
sed −i s/reactingTwoPhaseEulerFoam/myReactingTwoPhaseEulerFoam/g Allw∗
sed −i /"reactingMultiphaseEulerFoam\/Allwclean"/d Allwclean
sed −i /"reactingMultiphaseEulerFoam\/Allwmake\ \$targetType\ \$∗"/d Allwmake
sed −i s/reactingTwoPhaseEulerFoam/myReactingTwoPhaseEulerFoam/g
    myReactingTwoPhaseEulerFoam/Make/files
sed −i s/FOAM_APPBIN/FOAM_USER_APPBIN/g myReactingTwoPhaseEulerFoam/Make/files
mv myReactingTwoPhaseEulerFoam/reactingTwoPhaseEulerFoam.C
    myReactingTwoPhaseEulerFoam/myReactingTwoPhaseEulerFoam.C
./Allwclean
./Allwmake
```

5.3 Creating a new saturation model

The most efficient way to create the new antoine model is to copy the old model and modify the member functions using the equations described in section 5.1.

```
cp −rf interfacialCompositionModels/saturationModels/Antoine interfacialCompositionModels/
    saturationModels/AntoineConventional
sed −i s/Antoine/AntoineConventional/g interfacialCompositionModels/saturationModels/
    AntoineConventional/∗
mv interfacialCompositionModels/saturationModels/AntoineConventional/Antoine.C
    interfacialCompositionModels/saturationModels/AntoineConventional/AntoineConventional.
    C
```

```
mv interfacialCompositionModels/saturationModels/AntoineConventional/Antoine.H
    interfacialCompositionModels/saturationModels/AntoineConventional/AntoineConventional.
    H
```

Next it is necessary to tell the solver that there is another saturation model available. Edit the file `interfacialCompositionModels/Make/files` and include the link to the AntoineConventional model below the Antoine model in the file. The interfacialCompositionModels directory should then be compiled.

```
wclean interfacialCompositionModels
wmake interfacialCompositionModels
```

Replace the *AntoineConventional.C* file with the code below.

```
#include "AntoineConventional.H"
#include "addToRunTimeSelectionTable.H"
#include "function1.H"

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

namespace Foam
{
namespace saturationModels
{
    defineTypeNameAndDebug(AntoineConventional, 0);
    addToRunTimeSelectionTable(saturationModel, AntoineConventional, dictionary);
}
}


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

Foam::saturationModels::AntoineConventional::AntoineConventional(const dictionary& dict)
:
    saturationModel(),
    A_("A", dimless, dict),
    B_("B", dimTemperature, dict),
    C_("C", dimTemperature, dict)
{}


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

Foam::saturationModels::AntoineConventional::~AntoineConventional()
{}


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

Foam::tmp<Foam::volScalarField>
Foam::saturationModels::AntoineConventional::pSat
(
    const volScalarField& T
) const
{
    return
```

```
        dimensionedScalar("one", dimPressure, 1)
       *100000*pow(10,(A_ - B_/(C_ + T)));
}


Foam::tmp<Foam::volScalarField>
Foam::saturationModels::AntoineConventional::pSatPrime
(
    const volScalarField& T
) const
{
    return  pSat(T)*B_*log(dimensionedScalar("one", dimless, 1)*10)
            /pow((C_ + T),2);
}


Foam::tmp<Foam::volScalarField>
Foam::saturationModels::AntoineConventional::lnPSat
(
    const volScalarField& T
) const
{
    return log(pSat(T));
}


Foam::tmp<Foam::volScalarField>
Foam::saturationModels::AntoineConventional::Tsat
(
    const volScalarField& p
) const
{
    return
        B_/(A_ - log10(p*dimensionedScalar("one", dimless/dimPressure, 1)))
      - C_;
}
```

5.4 New Case Setup

Copy the original case described in section 2 and remove all the time files and other files which are not required.

```
cp -rf bubbleColumnEvaporatingDissolving methanolColumn
cd methanolColumn
rm -rf 0 0.* [1-9]* constant/polyMesh
```

The files which require changing are outlined in table 5.1. The first two files can be quickly changed as illustrated below.

```
mv water.gas methanol.gas
mv water.liquid methanol.liquid
```

The files in the constant folder will need to be replace with the files below.

5.4.1 constant/thermophysicalProperties.gas

```
FoamFile
{
    version       2.0;
    format        ascii ;
    class         dictionary;
    location      "constant";
    object        thermophysicalProperties.gas;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

thermoType
{
    type              heRhoThermo;
    mixture           multiComponentMixture;
    transport         const;
    thermo            hConst;
    equationOfState perfectGas;
    specie            specie;
    energy            sensibleInternalEnergy;
}

dpdt yes;

species
(
    air
    methanol
);

inertSpecie  air;

"(mixture|air)"
{
    specie
    {
        molWeight   28.9;
    }
    thermodynamics
    {
        Hf          0;
        Cp          1012.5;
    }
    transport
    {
        mu          1.84e-05;
        Pr          0.7;
    }
}

methanol
{
    specie
    {
        molWeight   32.01;
    }
    thermodynamics
    {
        Hf          −6.2938e+06;
        Cp          1380.6;
    }
    transport
    {
        mu          1.84e−05;
        Pr          0.7;
    }
}
```

5.4.2 constant/thermophysicalProperties.liquid

```
FoamFile
{
    version     2.0;
    format      ascii ;
    class       dictionary ;
    location    ”constant”;
    object      thermophysicalProperties.liquid ;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

thermoType
{
    type            heRhoThermo;
    mixture         multiComponentMixture;
    transport       const ;
    thermo          hConst;
    equationOfState perfectFluid;
    specie          specie ;
    energy          sensibleInternalEnergy ;
}

dpdt yes;

species
(
    air
    methanol
) ;

inertSpecie  methanol;

”(mixture|methanol)”
{
    specie
    {
        molWeight   32.01;
    }
    equationOfState
    {
        R           3000;
        rho0        764; //Methanol at 55 C from Engineering ToolBox
    }
    thermodynamics
    {
        Hf          −7.4467e+06;
        Cp          2686; //Methanol at 55 C from Engineering ToolBox
    }
    transport
    {
        mu          3.99e−4; //Methanol at 55 C from Engineering ToolBox
        Pr           5.3;  //k = 0.202 W/m/K thermal conductivity. Pr = mu Cp / k
    }
}

air
{
    specie
    {
        molWeight   28.9;
    }
    equationOfState
    {
        R           3000;
        rho0        764; //Methanol at 55 C from Engineering ToolBox
    }
    thermodynamics
    {
        Hf          0;
        Cp          2686; //Methanol at 55 C from Engineering ToolBox
    }
    transport
    {
        mu          3.99e−4; //Methanol at 55 C from Engineering ToolBox
        Pr           5.3;  //k = 0.202 W/m/K thermal conductivity. Pr = mu Cp / k
    }
}
```

5.4.3 constant/phaseProperties

File very long. To be downloaded.

5.5 Run new case

The case can now be run in the same manner as in section 2.7. generate the mesh using the blockMesh utilty.

```
cp −rf 0orig 0
blockMesh
setFields
reactingTwoPhaseEulerFoam >& log&
paraFoam
```

It is possible to see the methanol transferring from the liquid to the gas phase in paraview. To do this select methanol.gas from the fields tab in the properties section. Click apply. The methanol.gas field then needs to be selected in the main drop down menu above the viewing window. Press play and watch the mass transfer.

Study questions

1. In what file is the twoPhaseSystem for a case set?
2. What type of twoPhaseSystem would be used to simulate a condensor?
3. In which file are the typedefs of the twoPhaseSystem located?
4. In which file are the typedefs of the phaseModels located?